비공식 한국어 번역

원서: Jeff Erickson, *Algorithms*, 1st edition (2019), `http://algorithms.wtf`. © 2019 Jeff Erickson. 원서는 Creative Commons Attribution 4.0 International(CC BY 4.0)로 공개되었습니다.

이 번역본은 원서 PDF에서 영어 원고를 복원한 뒤 한국어로 옮긴 개작물이며, 같은 CC BY 4.0으로 제공합니다. 원서의 색인 3종과 그림 1.25는 싣지 않았고, 그림 안의 영어 문구는 원서 그대로 두었습니다. 저자가 검토하거나 승인한 번역이 아닙니다.

영어 원고는 AI 모델의 도움으로 복원했으며, 번역은 yeokja와 함께 Anthropic 사의 `claude-opus-5-5` 모델을 활용하여 번역되었으며 학습을 모두 비허용한 상태로 작업하였습니다.

Incipit prologus in libro alghoarismi de practica arismetrice.

— Ioannis Hispalensis [세비야의 요한?],
Liber algorismi de pratica arismetrice (1135년경)

벗이여, 그대가 그것을 어떻게 이해하게 될지 알려 드리겠습니다.
가서 그것에 관한 책을 쓰십시오.

— 헨리 홈, 케임스 경(1696–1782),
길버트 엘리엇 경에게 보낸 편지에서

개인은 언제나 잘못 생각합니다. 그는 많은 일을 계획했고, 다른 이들을 조력자로 끌어들였으며, 그중 일부 혹은 모두와 다투었고, 숱한 실수를 저질렀지만, 그래도 무언가는 이루어집니다. 모두가 조금씩 나아가지만, 개인은 언제나 잘못 생각합니다. 결과는 다소 새로우며, 그가 스스로에게 약속했던 것과는 사뭇 다릅니다.

— 랠프 월도 에머슨, "경험", *에세이 제2집* (1844)

위에서 제가 개괄한 것은 한 권의 책의 내용입니다. 그 기본 구상을 실현하고 세부 사항을 모두 담아내는 일은 아마도 불가능할 것입니다. 제가 쓴 것은 이 책의 예비판의 두 번째 또는 세 번째 초고입니다

— 마이클 스피박(Michael Spivak),
Differential Geometry, *Volume I* 초판 서문 (1970)

서문

이 책에 대하여

이 교재는 제가 어바나-샴페인 소재 일리노이 대학교에서 여러 알고리즘 강의를 위해 작성한 강의 노트 모음에서 비롯되었습니다. 저는 이 강의들을 1999년 1월부터 대략 1년에 한 번씩 맡아 왔습니다. 학부 이론 교육과정이 바뀐 것을 계기로 저는 2016년에 노트를 대대적으로 개정했습니다. 이 책은 개정한 노트 가운데 가장 기초적인 강의 내용을 다룬 일부로 이루어져 있으며, 주로 새로 개설된 3학년 필수 이론 과목의 알고리즘 내용을 반영합니다.

선수 지식

제가 일리노이 대학교에서 가르치는 알고리즘 강의에는 중요한 선수 과목이 두 가지 있습니다. 하나는 이산수학 과목이고, 다른 하나는 기초 자료구조 과목입니다. 따라서 이 교재는 대부분의 학생에게 자료구조와 알고리즘을 처음 배우는 *첫* 강의로는 적합

하지 않을 것입니다. 특히 저는 독자가 적어도 다음의 구체적인 주제들을 어느 정도는 알고 있다고 가정합니다.

- **이산수학:** 고등학교 수준의 대수학, 로그 항등식, 소박한 집합론, 불 대수, 1차 술어 논리, 집합, 함수, 동치 관계, 부분 순서, 모듈러 산술, 재귀적 정의, 트리(자료구조가 아닌 추상적 대상으로서의 트리), 그래프(함수 그래프가 아닌 정점과 간선으로 이루어진 그래프).

- **증명 기법:** 직접 증명, 간접 증명, 귀류법, 철저한 경우 분석, 그리고 귀납법(특히 "강한" 귀납법과 "구조적" 귀납법)입니다. 0장은 귀납법을 사용하며, $n-1$장이 귀납법을 사용할 때마다 n장도 귀납법을 사용합니다.

- **반복적 프로그래밍 개념:** 변수, 조건문, 반복문, 레코드, 간접 참조(주소/포인터/참조), 서브루틴, 재귀. 특정 프로그래밍 언어에 능숙할 것을 전제하지는 않지만, 간접 참조와 재귀를 모두 지원하는 언어를 적어도 하나는 다뤄 본 경험이 있다고 전제합니다.

- **기본 추상 자료형:** 스칼라, 시퀀스, 벡터, 집합, 스택, 큐, 맵/딕셔너리, 순서 있는 맵/딕셔너리, 우선순위 큐.

- **기본 자료구조:** 배열, 연결 리스트(단일 및 이중, 선형 및 원형), 이진 탐색 트리, *균형* 이진 탐색 트리 중 적어도 한 가지(AVL 트리, 레드-블랙 트리, 트립, 스킵 리스트, 스플레이 트리 등), 해시 테이블, 이진 힙, 그리고 무엇보다 중요한 것으로 이 목록과 앞의 목록의 차이점입니다.

- **기본적인 계산 문제:** 기초 산술, 정렬, 탐색, 열거, 트리 순회(전위, 중위, 후위, 레벨 순서 순회 등)가 있습니다.

- **기본 알고리즘:** 초등 산술 계산법(algorism), 순차 탐색, 이진 탐색, 정렬(선택, 삽입, 병합, 힙, 퀵, 기수 정렬 등), (최소한 이진) 트리에서의 너비 우선 탐색과 깊이 우선 탐색, 그리고 무엇보다도 이 목록과 앞의 목록 사이의 차이점입니다.

- **기초 알고리즘 분석:** 점근 표기법(o, O, Θ, Ω, ω), 반복문을 합으로 바꾸고 재귀 호출을 점화식으로 바꾸는 방법, 간단한 합과 점화식을 계산하는 방법을 다룹니다.

- **수학적 성숙도:** 추상화, 형식적(특히 재귀적) 정의, (특히 귀납적) 증명을 능숙하게 다루는 능력; 수학적 논증을 작성하고 따라가는 능력; 구문적, 의미적 및/또는 논리적 난센스를 알아보고 피하는 능력입니다.

이 책에서는 이러한 선수 지식 중 일부를 맥락상 필요할 때 *간략히* 다루지만, 좋은 입문이라기보다는 상기시켜 주는 정도에 가깝습니다. 더 철저한 개관을 원한다면 무료로 이용할 수 있는 다음 참고 자료를 강력히 추천합니다:

- Margaret M. Fleck. *Building Blocks for Theoretical Computer Science*. 버전 1.3(2013년 1월) 또는 그 이후 버전은 http://mfleck.cs.illinois.edu/building-blocks/에서 구할 수 있습니다.

- Eric Lehman, F. Thomson Leighton, Albert R. Meyer. *Mathematics for Computer Science*. 2018년 6월 개정판은 https://courses.csail.mit.edu/6.042/spring18/에서 구할 수 있습니다. (가장 최신 개정판을 검색해 보시기를 강력히 권합니다.)
- Pat Morin. *Open Data Structures (오픈 자료구조)*. 0.1Gβ판(2016년 1월) 또는 그 이후 판을 http://opendatastructures.org/에서 구할 수 있습니다.
- Don Sheehy. *A Course in Data Structures and Object-Oriented Design (자료구조와 객체 지향 설계 강좌)*. 2019년 2월판 또는 이후 개정판은 https://donsheehy.github.io/datastructures/에서 구할 수 있습니다.

추가 참고 문헌

이 책이든 다른 어떤 책이든, 단 하나의 참고 자료에만 스스로를 한정하지 마십시오. 저자와 독자는 어떤 지적 자료를 대하든 저마다의 관점을 가지고 옵니다. 어떤 교수자도 모든 학생과, 심지어 아주 뛰어난 모든 학생과도 "잘 통하지"는 못합니다. *저자의* 직관을 *여러분의* 머릿속에 가장 효과적으로 전달해 주는 저자를 찾는 데는 어느 정도 노력이 들지만, 그 노력은 장기적으로 충분히 보답받습니다.

다음 참고 문헌들은 직관, 예제, 연습 문제, 영감을 얻는 데 특히 귀중한 원천이 되었습니다. 다만 이 목록이 모든 문헌을 망라하지는 않습니다.

- Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *컴퓨터 알고리즘의 설계와 분석*. Addison-Wesley, 1974. (저는 라이스 대학교 학부생 시절에 이 교재를 사용했고, UC 어바인 석사 과정 학생 시절에 다시 사용했습니다.)
- Boaz Barak. *Introduction to Theoretical Computer Science*. 교재 초고이며, 가장 최근 개정은 2019년 6월입니다. (할아버지 세대의 이론 전산학 교재와는 다르며, 바로 그 덕분에 훨씬 좋습니다. 무료라는 점은 반가운 덤입니다.)
- Thomas Cormen, Charles Leiserson, Ron Rivest, Cliff Stein. *Introduction to Algorithms (알고리즘 입문)*, 제3판. MIT Press/McGraw-Hill, 2009. (저는 버클리에서 조교로 일할 때 초판을 사용했습니다.)
- Sanjoy Dasgupta, Christos H. Papadimitriou, Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2006. (내용 면에서는 아마 이 책과 가장 가깝지만, 서술은 훨씬 간결합니다.)
- Jeff Edmonds. *알고리즘에 대해 생각하는 법(How to Think about Algorithms)*. Cambridge University Press, 2008.
- Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- Michael T. Goodrich, Roberto Tamassia. *알고리즘 설계: 기초, 분석, 인터넷 예제(Algorithm Design: Foundations, Analysis, and Internet Examples)*. John Wiley & Sons, 2002.

- Jon Kleinberg, Éva Tardos. *알고리즘 설계*. Addison-Wesley, 2005. 가능하면 도서관에서 빌리십시오.
- Donald Knuth. *The Art of Computer Programming*, 1–4A권. Addison-Wesley, 1997년 및 2011년. (제가 14살 때 부모님께서 크리스마스 선물로 처음 세 권을 주셨습니다. 안타깝게도 실제로 그 책들을 읽은 것은 한참 뒤였습니다.)
- Udi Manber. *알고리즘 입문: 창의적 접근(Introduction to Algorithms: A Creative Approach)*. Addison-Wesley, 1989. (저는 버클리에서 조교로 일할 때 이 교재를 사용했습니다.)
- Ian Parberry. *Problems on Algorithms(알고리즘에 관한 문제)*. Prentice-Hall, 1995 (절판). 소액의 자선 기부를 하기로 동의하면 https://larc.unt.edu/ian/books/free/license.html에서 내려받을 수 있습니다. 동의한 약속을 꼭 지키십시오.
- Robert Sedgewick, Kevin Wayne. *Algorithms*. Addison-Wesley, 2011.
- Robert Endre Tarjan. *자료구조와 네트워크 알고리즘*. SIAM, 1983.
- 제가 버클리에서 직접 들은 알고리즘 수업, 특히 딕 카프(Dick Karp)와 라이문트 자이델(Raimund Seidel)이 가르친 수업의 강의 노트입니다.
- 전 세계의 수많은 동료가 웹에 무료로 공개한 강의 노트, 슬라이드, 과제, 시험, 동영상 강의, 연구 논문, 블로그 글, StackExchange 질문과 답변, 팟캐스트, 그리고 완전한 형태의 MOOC.

연습문제에 관하여

각 장은 여러 개의 연습 문제로 끝나며, 그중 대부분은 제가 과제, 토론/실습 시간 또는 시험에서 적어도 한 번은 사용한 것입니다. 연습 문제는 난이도가 높아지는 순서로 배열되어 있지 *않으며*, (대체로) 공통된 기법이나 주제별로 묶여 있습니다. 일부 문제에는 다음과 같은 기호가 표시되어 있습니다.

- ♥빨간 하트는 특히 어려운 문제를 나타냅니다. 이 중 상당수는 일리노이 대학교 박사 과정 학생들의 자격시험에 출제된 적이 있습니다. *정말로* 어려운 소수의 문제에는 ♥큰 하트가 표시되어 있습니다.
- ♦파란색 다이아몬드는 뒤에 나오는 장의 내용을 알아야 풀 수 있지만 주제상으로는 현재 위치에 속하는 문제를 나타냅니다. 그러나 *앞에서* 다룬 내용을 알아야 하는 문제는 따로 표시하지 않습니다. 이 책도 인생과 마찬가지로 앞의 것이 차곡차곡 쌓여 가기 때문입니다.
- ♣초록색 클로버는 유한 상태 기계, 선형대수학, 확률, 평면 그래프처럼 이 책의 범위를 벗어나는 내용에 익숙해야 풀 수 있는 문제를 나타냅니다. 이러한 문제는 드뭅니다.

- ♠검은 스페이드는 상당한 양의 단순 반복 작업이나 코딩, 또는 둘 다가 필요한 문제를 나타냅니다. 이런 문제는 드뭅니다.
- ⋆주황색 별은 여러분이 1998년 이전에 제조된 럭키 참스를 먹고 있다는 뜻입니다. 으윽.

이 연습문제들은 연습의 기회로 마련된 것이며, 그 자체를 위한 대상으로 마련된 것이 아닙니다. 각 문제의 목표는 그 특정 문제를 푸는 것이 아니라, 특정한 기술들을 연마하거나 특정 *유형*의 문제를 푸는 연습을 하는 것입니다. 부분적으로는 이러한 이유로 저는 연습문제의 풀이를 제공하지 않습니다. 풀이가 핵심이 아니기 때문입니다. 특히 "강사용 매뉴얼"은 없습니다. 어떤 문제를 직접 풀 수 없다면 아마 그 문제를 학생들에게 과제로 내지 않는 것이 *좋을* 것입니다. 그렇긴 하지만, 제가 *이번* 학기에 낸 숙제 문제의 풀이는 아마 제가 가르치고 있는 강좌의 웹 페이지에서 찾을 수 있을 것입니다. 그리고 *여러분*이 강사용 매뉴얼을 쓰는 것을 막는 것은 아무것도 없습니다!

이 책을 훔치십시오!

이 책은 원래 출처를 밝히기만 하면 여러분이 *저의 허락 없이도* 그 내용을 사용, 재배포, 개작, 리믹스할 수 있도록 허용하는 크리에이티브 커먼즈 라이선스에 따라 출판되었습니다. 이 책의 전체 전자판은 다음 위치 중 어느 곳에서나 무료로 이용하실 수 있습니다:

- 이 책의 웹사이트: http://jeffe.cs.illinois.edu/teaching/algorithms/
- 기억하기 쉬운 단축 주소: http://algorithms.wtf
- 버그 제보 사이트: https://github.com/jeffgerickson/algorithms
- 인터넷 아카이브: https://archive.org/details/Algorithms-Jeff-Erickson

이 책의 웹사이트에는 관련 주제와 더 심화된 주제를 다루는 수백 쪽 분량의 추가 강의 노트가 실려 있으며, 지난 숙제, 시험, 토론/실습 문제를 비롯한 여러 교육 자료를 거의 빠짐없이 모아 둔 아카이브도 있습니다. 저는 알고리즘 수업을 가르칠 때마다 교육 자료를 수정하고 갱신하며 때로는 일부를 덜어 내므로, 제가 현재 가르치고 있는 강의의 웹 페이지에서 더 최근에 개정된 판을 찾으실 수 있습니다.

학생이든 강사이든, 제 허락을 구하지 않고도 이 교재나 제 다른 강의 노트의 어느 부분이든 여러분의 수업에서 얼마든지 사용하셔도 좋습니다. 바로 그래서 이 자료들을 웹에 올려 두었습니다! 다만 책 제목을 밝히거나 http://algorithms.wtf로 연결되는 링크를 달아 이 책을 인용해 주시기 바랍니다. 여러분이 학생이고 숙제를 하는 데 제 강의 자료를 활용한다면 이 점은 *특히* 중요합니다. (담당 강사에게도 확인하십시오.)

하지만 강의하시는 분이라면 *직접 작성한* 추가 자료로 이 자료들을 보충하시기를 강력히 권합니다. 자료를 직접 작성하면 그 내용을 더 깊이 익히고 수업에서 더 잘 설명하게 되며, 이는 다시 학생들이 내용을 더 깊이 익히는 데 도움이 됩니다. 또한 이 책에서 마음에 들지 않는 부분 때문에 겪는 좌절감도 넘어설 수 있습니다. 모든 교과서는 ~~crap~~ 불완전하며, 이 책도 예외가 아닙니다.

마지막으로, **여러분이 작성한 것은 무엇이든 오픈 웹에 공개하여 전 세계 누구나 자유롭고 쉽게 이용할 수 있도록 해 주십시오**—학습 관리 시스템의 관문이나 그 밖의 유료 장벽 뒤에 숨겨 두지 마십시오—그래야 다른 곳의 학생과 교수자도 여러분의 독창적인 통찰로부터 도움을 받을 수 있습니다. 특히 슬라이드, 동영상, 해답집처럼 이 교재를 직접 보완하는 유용한 자료를 만드신다면 저에게 알려 주십시오. 그러면 책 웹사이트에 여러분의 자료로 연결되는 링크를 추가할 수 있습니다.

감사의 글

이 교과서는 헤아릴 수 없이 많은 알고리즘 학생, 교사, 연구자의 기여에 크게 힘입었습니다. 특히 제 강의 노트를 주요 참고 자료로 사용하고, 유용한(때로는 뼈아픈) 비판을 해 주었으며, 정말 형편없었던 초기 원고를 견뎌 준 3천 명이 넘는 일리노이 학생들에게 깊이 감사드립니다. 또한 이 노트를 자신의 수업에서 사용하고 도움이 되는 의견과 버그 리포트를 보내 준 전 세계의 많은 동료와 학생들에게도 감사드립니다.

특히 훌륭한 조교들이 보내 준 피드백과 기여(특히 연습 문제)에 깊이 감사드립니다:

> Aditya Ramani, Akash Gautam, Alex Steiger, Alina Ene, Amir Nayyeri, Asha Seetharam, Ashish Vulimiri, Ben Moseley, Brad Sturt, Brian Ensink, Chao Xu, Charlie Carlson, Chris Neihengen, Connor Clark, Dan Bullok, Dan Cranston, Daniel Khashabi, David Morrison, Ekta Manaktala, Erin Wolf Chambers, Gail Steitz, Gio Kao, Grant Czajkowski, Hsien-Chih Chang, Igor Gammer, Jacob Laurel, John Lee, Johnathon Fischer, Junqing Deng, Kent Quanrud, Kevin Milans, Kevin Small, Konstantinos Koiliaris, Kyle Fox, Kyle Jao, Lan Chen, Mark Idleman, Michael Bond, Mitch Harris, Naveen Arivazhagen, Nick Bachmair, Nick Hurlburt, Nirman Kumar, Nitish Korula, Patrick Lin, Phillip Shih, Rachit Agarwal, Reza Zamani-Nasab, Rishi Talreja, Rob McCann, Sahand Mozaffari, Shalan Naqvi, Shripad Thite, Spencer Gordon, Srihita Vatsavaya, Subhro Roy, Tana Wattanawaroon, Umang Mathur, Vipul Goyal, Yasu Furakawa, Yipu Wang.

또한 일리노이의 동료 교수들인 Alexandra Kolla, Cinda Heeren, Edgar Ramos, Herbert Edelsbrunner, Jason Zych, Kim Whittlesey, Lenny Pitt, Madhu Parasarathy, Mahesh Viswanathan, Margaret Fleck, Shang-Hua Teng, Steve LaValle, 그리고 특히 Chandra Chekuri, Ed Reingold, Sariel Har-Peled와 나눈 많은 토론에서도 큰 도움을 받았습니다.

물론 이 책은 애초에 저에게 이 알고리즘이라는 것을 가르쳐 주신 분들께 큰 빚을 지고 있습니다. 라이스 대학교의 밥 빅스비(Bob Bixby)와 마이클 펄먼(Michael Pearlman), 어바인의 데이비드 엡스타인(David Eppstein), 댄 허슈버그(Dan Hirschberg), 조지 루커(George Lueker), 그리고 버클리의 아비람 라나데(Abhiram Ranade), 딕 카프(Dick Karp), 마누엘 블룸(Manuel Blum), 마이크 루비(Mike Luby), 라이문트 자이델(Raimund Seidel)이 그분들입니다.

전체 강의 구조의 첫 번째 반복과, 애초에 제 강의 노트를 직접 작성하겠다는 생각은 Herbert Edelsbrunner에게서, 제 노트의 일부를 책으로 엮겠다는 생각은 Steve LaValle에게서, 그리고 책 디자인의 여러 요소는 Robert Ghrist에게서 훔쳐 왔습니다.

독자여, 주의하십시오!

물론 그 결과로 나온 이 책의 어떤 결함에 대해서도 그분들을 탓해서는 안 됩니다. 여러 차례 수정과 편집을 거쳤지만, 이 책에는 여전히 실수, 버그, 실언, 누락, 대혼란, 땜질식 해법, 오타, 수식 오타, 문법 오타, 생각 오타, 머릿속 헛발질, 잘못된 설계 결정, 역사적 부정확성, 시대착오, 비일관성, 과장, 우유부단함, 헛소리, 왜곡, 지나친 단순화, 중복, 장황함, 허튼소리, 쓰레기, 군더더기, 잡동사니, 그리고 새빨간 거짓말이 여럿 들어 있으며, **이 모든 것은 전적으로 스티브 스키에나의 잘못입니다.**

저는 https://github.com/jeffgerickson/algorithms에서 이슈 트래커를 운영하고 있으며, 여러분과 같은 독자는 이곳에서 버그 신고, 기능 요청, 그리고 이 책에 대한 전반적인 의견을 제출할 수 있습니다. 수학적 오류든, 문법적 오류든, 역사적 오류든, 오탈자든, 문화적 오류든, 그 밖의 어떤 종류의 오류든, 그리고 본문이든 연습문제든 제 다른 강의 자료든 상관없이 오류를 발견하시면 알려 주십시오. (스티브는 별로 신경 쓰지 않을 것입니다.) 물론 그 밖의 모든 의견도 환영합니다!

즐겁게 읽으십시오!

— Jeff

남아 있는 모든 결함에 대해 저자가 너그럽게 책임을 지는 것이 관례입니다. 저는 그러지 않습니다. 이 책의 모든 오류, 결함, 문제는 다른 누군가의 잘못이지만, 누구의 책임인지 가려낼 수 있도록 알려 주신다면 감사하겠습니다.

— Steven S. Skiena, *알고리즘 설계 매뉴얼* (1997)

틀림없이 이 발언 뒤에는 모든 교재를 주석과 함께 나열한 목록과, 각 교재가 왜 형편없는지에 대한 설명이 이어질 것입니다.

— Adam Contini, MetaFilter, 2010년 1월 4일

차례

서문 **ii**
이 책에 대하여 . . . ii
선수 지식 . . . ii
추가 참고 문헌 . . . iv
연습문제에 관하여 . . . v
이 책을 훔치십시오! . . . vi
감사의 글 . . . vii
독자여, 주의하십시오! . . . viii

0 소개 **1**
0.1 알고리즘이란 무엇입니까? . . . 1
0.2 곱셈 . . . 3
격자 곱셈 . . . 3
배가와 반분 . . . 4
컴퍼스와 눈금 없는 자 . . . 6
0.3 의회 의석 배분 . . . 8
0.4 나쁜 예 . . . 9
0.5 알고리즘 기술하기 . . . 10
문제 명세하기 . . . 11
알고리즘 설명하기 . . . 12
0.6 알고리즘 분석 . . . 13
정확성 . . . 13
실행 시간 . . . 13
연습문제 . . . 16

1 재귀 **19**
1.1 환원 . . . 19
1.2 단순화하고 위임하기 . . . 20
1.3 하노이의 탑 . . . 21
1.4 병합 정렬 . . . 24
정확성 . . . 25

분석 26
1.5 퀵정렬 26
정확성 27
분석 28
1.6 패턴 29
1.7 재귀 트리 29
♥바닥 함수와 천장 함수를 무시해도 괜찮습니다, 정말입니다 32
1.8 ♥선형 시간 선택 33
퀵셀렉트 33
좋은 피벗 34
분석 35
타당성 검사 36
1.9 빠른 곱셈 37
1.10 거듭제곱 39
연습문제 41

2 백트래킹 66
2.1 N 퀸 문제 66
2.2 게임 트리 68
2.3 부분집합 합 71
정확성 72
분석 72
변형 73
2.4 일반적인 패턴 73
2.5 텍스트 분할(*Interpunctio Verborum*) 75
인덱스 정식화 78
♥분석 79
변형 80
2.6 최장 증가 부분 수열 80
2.7 최장 증가 부분 수열, 두 번째 접근 83
2.8 최적 이진 탐색 트리 84
♥분석 86
연습문제 87

3 동적 계획법 91
3.1 마트라브리타(Mātrāvṛtta) 91
백트래킹은 느릴 수 있습니다 92
메모(라)이제이션: 모든 것을 기억하기 93
동적 계획법: 신중하게 채우기 94
결국 모든 것을 다 기억하지는 마십시오 96
3.2 ♥여담: 훨씬 더 빠른 피보나치 수 97
워워! 그렇게 서두르지 마십시오! 98

3.3 *Interpunctio Verborum Redux* 98
3.4 패턴: 똑똑한 재귀 99
3.5 경고: 탐욕은 어리석습니다 101
3.6 최장 증가 부분 수열 102
첫 번째 점화식: 이것이 다음입니까? 102
두 번째 점화식: 다음은 무엇입니까? 103
3.7 편집 거리 104
재귀 구조 105
점화식 105
동적 계획법 106
3.8 부분집합 합 109
3.9 최적 이진 탐색 트리 110
3.10 트리에서의 동적 계획법 113
연습문제 115

4 탐욕 알고리즘 **148**
4.1 테이프에 파일 저장하기 148
4.2 강의 일정 짜기 150
4.3 일반적인 패턴 153
4.4 허프만 코드 153
4.5 안정 매칭 157
몇 가지 나쁜 아이디어 159
보스턴 풀 알고리즘과 Gale-Shapley 알고리즘 160
실행 시간 162
정확성 162
최적성! 163
연습문제 164

5 기본 그래프 알고리즘 **175**
5.1 소개 및 역사 175
5.2 기본 정의 178
5.3 표현과 예시 180
5.4 자료구조 183
인접 리스트 183
인접 행렬 184
비교 185
5.5 아무거나 우선 탐색 186
분석 188
5.6 주요 변형 189
스택: 깊이 우선 189
큐: 너비 우선 189
우선순위 큐: 최선 우선 189

연결되지 않은 그래프 190
방향 그래프 192
5.7 그래프 환원: 플러드 필(Flood Fill) 192
연습문제 194

6 깊이 우선 탐색 **211**
6.1 전위 순회와 후위 순회 213
정점과 간선의 분류 214
6.2 사이클 탐지 216
6.3 위상 정렬 217
암시적 위상 정렬 218
6.4 메모이제이션과 동적 계획법 220
DAG에서의 동적 계획법 221
6.5 강한 연결성 222
6.6 선형 시간에 강한 연결 요소 찾기 224
코사라주와 샤리르의 알고리즘 225
♥Tarjan 알고리즘 226
연습문제 229

7 최소 신장 트리 **241**
7.1 서로 다른 간선 가중치 241
7.2 유일한 최소 신장 트리 알고리즘 243
7.3 보루프카 알고리즘 245
여러분이 원하는 최소 신장 트리 알고리즘은 바로 이것입니다 246
7.4 야르니크("Prim") 알고리즘 247
♥야르니크 알고리즘 개선하기 248
7.5 크러스컬 알고리즘 249
연습문제 251

8 최단 경로 **257**
8.1 최단 경로 트리 258
8.2 ♥음수 간선 259
8.3 유일한 SSSP 알고리즘 260
8.4 가중치 없는 그래프: 너비 우선 탐색 262
8.5 방향 비순환 그래프: 깊이 우선 탐색 265
8.6 최선 우선 탐색: 데이크스트라 알고리즘 268
음수 간선이 없는 경우 269
♥음수 간선 271
8.7 모든 간선을 완화하기: 벨먼-포드 272
Moore의 개선 275
동적 계획법 정식화 276
연습문제 279

9 모든 쌍 최단 경로 **291**
9.1 서론 . . . 291
9.2 수많은 단일 출발점 . . . 292
9.3 가중치 재조정 . . . 293
9.4 존슨 알고리즘 . . . 294
9.5 동적 계획법 . . . 294
9.6 분할 정복 . . . 296
9.7 기묘한 행렬 곱셈 . . . 298
9.8 (클리니-로이-)플로이드-워셜(-잉거먼) . . . 299
연습문제 . . . 301

10 최대 흐름 & 최소 컷 **308**
10.1 흐름 . . . 309
10.2 컷 . . . 310
10.3 최대 흐름-최소 컷 정리 . . . 311
10.4 포드와 풀커슨의 증가 경로 알고리즘 . . . 314
♥무리수 용량 . . . 315
10.5 흐름의 결합과 분해 . . . 317
10.6 에드먼즈와 카프의 알고리즘 . . . 320
가장 굵은 증가 경로 . . . 320
최단 증가 경로 . . . 321
10.7 추가 진전 . . . 322
연습문제 . . . 324

11 흐름과 컷의 응용 **332**
11.1 간선 서로소 경로 . . . 332
11.2 정점 용량과 정점 서로소 경로 . . . 333
11.3 이분 매칭 . . . 333
11.4 튜플 선택 . . . 335
시험 일정 잡기 . . . 337
11.5 서로소 경로 덮개 . . . 339
최소 교수 채용 . . . 340
11.6 야구 탈락 문제 . . . 342
11.7 프로젝트 선택 . . . 344
연습문제 . . . 346

12 NP-난해성 **357**
12.1 이길 수 없는 게임 . . . 357
12.2 P 대 NP . . . 359
12.3 NP-난해, NP-쉬움, NP-완전 . . . 360
12.4 ♥형식적 정의 (*HC SVNT DRACONES*) . . . 362
12.5 환원과 SAT . . . 363

12.6 3SAT (CIRCUITSAT에서 환원) 365
12.7 최대 독립 집합 (3SAT로부터) 368
12.8 일반적인 패턴 369
12.9 클리크와 정점 덮개 (독립 집합으로부터) 371
12.10그래프 채색(3SAT로부터의 환원) 372
12.11해밀턴 사이클 374
정점 덮개로부터 375
3SAT에서 378
변형과 확장 379
12.12부분집합 합 (정점 덮개로부터) 380
환원하는 자는 주의하십시오! 381
12.13기타 유용한 NP-난해 문제 381
12.14올바른 문제 선택하기 385
12.15사소한 실생활 예시 385
12.16얼룩말 너머로 388
다항 공간 389
지수 시간 390
더 높이! 391
연습문제 392

이미지 출처 **406**

판권 정보 **408**

Hinc incipit algorismus. *Haec algorismus ars praesens dicitur in qua talibus indorum fruimur bis quinque figuris 0. 9. 8. 7. 6. 5. 4. 3. 2. 1.*

— 빌라 데이의 알렉산데르 수사(Friar Alexander de Villa Dei), *Carmen de Algorismo* (c. 1220)

예술가가 자기 작업에 의식적으로 임해야 한다고 요구하시는 것은 옳습니다만,
선생께서는 서로 다른 두 가지를 혼동하고 계십니다.
바로 문제를 푸는 것과 질문을 올바르게 제기하는 것입니다.

— 안톤 체호프, A. S. 수보린에게 보낸 편지에서 (1888년 10월 27일)

낮은 일에서 우리 자신을 기계로 만들면 만들수록,
더 높은 일에 쓸 수 있는 힘을 그만큼 더 많이 풀어놓게 될 것입니다.

— Anna C. Brackett, *The Technique of Rest* (1892)

그런데 저는 지금 새벽 2시 30분에 기법에 관한 글을 쓰고 있습니다. 사람이 기법을 이야기하기 시작하는 순간이야말로 아이디어가 완전히 바닥났다는 증거라고 굳게 믿고 있는데도 말입니다.

— 레이먼드 챈들러, 얼 스탠리 가드너에게 보낸 편지 (1939년 5월 5일)

선한 사람에게는 규칙이 필요 없습니다.
제게 왜 그토록 많은 규칙이 있는지 알게 될 날은 오늘이 아닙니다,

— 닥터 [맷 스미스], "A Good Man Goes to War", *닥터 후* (2011)

0

소개

0.1 알고리즘이란 무엇입니까?

알고리즘은 명시적이고 정밀하며 모호하지 않고 기계적으로 실행할 수 있는 기본 명령어의 나열로, 보통 특정한 목적을 달성하기 위해 만들어집니다. 예를 들어, 다음은 99가 임의의 값일 때 그 성가신 노래 "벽 위의 맥주 99병(99 Bottles of Beer on the Wall)"을 부르는 알고리즘입니다.

```
BottlesOfBeer(n):
  For i ← n down to 1
      Sing "i bottles of beer on the wall, i bottles of beer,"
      Sing "Take one down, pass it around, i − 1 bottles of beer on the wall."
  Sing "No bottles of beer on the wall, no bottles of beer,"
  Sing "Go to the store, buy some more, n bottles of beer on the wall."
```

"algorithm(알고리즘)"이라는 단어는 알고리즘을 두려워하는 고전학자들이 짐작할 법한 것과 달리, "수"를 뜻하는 그리스어 어근 *arithmos*(ἀριθμός)와 "고통"을 뜻하는 *알고리즘*(ἄλγος)에서 유래한 것이 *아닙니다*. 오히려 이 단어는 9세기 페르시아 학자 Muḥammad ibn Mūsā al-Khwārizmī의 이름이 와전된 것입니다.[1] Al-Khwārizmī는 아마도 *Al-Kitāb al-mukhtaṣar fīhīsāb al-ğabr wa'l-muqābala*[2]라는 논저의 저자로 가장 잘 알려져 있을 것이며, 현대 단어 *대수학*가 바로 이 논저에서 유래했습니다. 또 다른 논저에서 al-Khwārizmī는 수를 적고 다루는 현대의 십진법을 설명했습니다. 특히 빈 자리를 나타내기 위해 작은 원, 곧 *ṣifr*를 사용하는 방식도 설명했습니다. 이 체계는 그보다 몇 세기 앞서 인도에서 개발된 것이었습니다. 이 논저에서 설명한 방법은 글로 적은 숫자나 셈돌을 사용하는 것으로, 영어에서 *산법(algorism)* 또는 *augrym*이라고 불리게 되었고, 그 숫자들은 영어에서 *암호*라고 불리게 되었습니다.

자릿값 표기법과 알콰리즈미(al-Khwārizmī)의 저작은 이미 일부 유럽 학자들에게 알려져 있었지만, 유럽에 "힌두-아라비아" 숫자 체계를 널리 퍼뜨린 사람은 피보나치라는 이름으로 더 잘 알려진 중세 이탈리아의 수학자이자 상인인 피사의 레오나르도입니다. 그가 1202년에 쓴 책 *산반서(Liber Abaci)*[3]의 영향도 있어, 13세기 유럽에서는 문자로 쓴 숫자가 계산판(당시에는 *주판*라고 불렀습니다)과 손가락 셈[4]을 대신하여 계산[5]에 선호되는 수단으로 자리 잡기 시작했습니다. 그 이유는 문자로 쓴 십진 숫자가 배우거나 쓰기 더 쉬웠기 때문이 *아니라*, 계산 과정을 검증할 수 있는 기록을 남겨 주었기 때문입니다. 서유럽에서 숫자 표기가 흔해진 것은 활판 인쇄술이 등장한 뒤의 일이며, 19세기 초 값싼 종이가 풍부해진 뒤에야 비로소 어디에서나 쓰이게 되었습니다.

결국 *산법(algorism)*이라는 단어는 그리스어 *arithmos*(그리고 어쩌면 앞서 언급한 *알고리즘*)에서 비롯된 민간 어원을 거쳐 오늘날의 *알고리즘*으로 변했습니다.[6] 그리하여 아주 최근까지도 *알고리즘*이라는 단어는 오로지 "아라비아" 숫자를 사용하는 자릿값 산술의 기계적인 기법만을 가리켰습니다. 이러한 절차를 빠르고 확실하게 수행하도록 훈련받은 사람들은 *알고리스트*나 *계산원*, 혹은 더 간단히 *컴퓨터*라고 불

[1] "압둘라의 아버지이자 모세의 아들인 콰리즘 사람 무함마드". Kwārizm은 우즈베키스탄 호레즘주에 있는 고대 도시로, 지금은 히바라고 불립니다.

[2] "복원과 대비에 의한 계산에 관한 요약서"

[3] *산반서(Liber Abaci)*라는 제목을 "주판의 책"으로 옮기고 싶어지지만, 더 정확한 번역은 "계산의 책"입니다. 피보나치 이전에도 이후에도 이탈리아어 *아바코*는 장치, 방법, 학교, 책 등 수치 계산과 관련된 모든 것을 가리키는 말로 쓰였습니다. 이는 오늘날 영어에서 "computer science"라는 말이 쓰이는 방식이나, "operations research"를 뜻하는 중국어 표현을 직역하면 "산가지를 쓰는 학문"이 되는 것과 매우 비슷합니다.

[4] *자릿수*(손가락이자 숫자)로 셈하기!

[5] *계산*(계산하다)라는 단어는 "작은 돌"을 뜻하는 라틴어 *미적분학*에서 유래했으며, 이는 계산판 위에 놓는 돌, 즉 초서가 *셈돌(augrym stones)*라고 부른 돌을 가리킵니다. 기원전 440년에 헤로도토스는 『*역사*』에서 다음과 같이 썼습니다. "그리스인들은 글을 쓰고 계산할 때(λογίζεσθαι ψήφοις, 직역하면 '조약돌로 셈하다') 왼쪽에서 오른쪽으로 진행하지만, 이집트인들은 그 반대로 합니다. 그런데도 이집트인들은 자기들의 글쓰기 방식이 오른쪽을 향하고 그리스인의 방식은 왼쪽을 향한다고 말합니다." (이상하게도 헤로도토스는 이집트인들이 달걀의 어느 쪽 끝부터 먹었는지에 대해서는 아무 말도 하지 않습니다.)

[6] 일부 중세 문헌은 그리스어 접두사 "algo-"가 "기술" 또는 "입문"을 뜻한다고 주장합니다. 또 다른 문헌들은 "Algus"나 "Algor" 또는 "Argus"라는 이름의 그리스 철학자나 인도의 왕, 혹은 스페인의 왕이 알고리즘을 발명했다고 주장합니다. 어쩌면 단테 알리기에리를 포함한 몇몇 사람은 심지어 그 발명자를 신화 속 그리스의 배 만드는 장인이자 아르고호에 이름을 남긴 아르고 원정대원과 같은 인물로 보기까지 했습니다. 이 우스꽝스러운 주장들 가운데 어느 것이든 역사적으로 정확하려는 의도였는지, 아니면 그저 기억을 돕기 위한 것이었는지는 분명하지 않습니다.

렸습니다.

0.2 곱셈

알고리즘이 정식 학문 연구의 주제가 된 지는 불과 수십 년밖에 되지 않았지만, 알고리즘은 문명의 여명기부터 우리와 함께해 왔습니다. 단계별 산술 계산을 서술한 기록은 인류가 남긴 가장 오래된 문자 언어의 사례에 속하며, 피보나치와 al-Khwārizmī같은 이들의 해설보다, 심지어 그들이 널리 퍼뜨린 자릿값 표기법보다도 훨씬 앞섭니다.

격자 곱셈

큰 수를 곱하는 가장 익숙한 방법은, 적어도 미국 학생들에게는, ***격자 알고리즘***입니다. 이 알고리즘은 피보나치가 *산반서(Liber Abaci)*에서 널리 알렸습니다. 피보나치는 이 알고리즘을 알콰리즈미(al-Khwārizmī)를 비롯한 아랍 문헌에서 배웠고, 알콰리즈미는 다시 브라마굽타의 7세기 논저 *Brāhmasphuṭasiddhānta*를 비롯한 인도 문헌에서 배웠으며, 브라마굽타는 중국 문헌에서 배웠을 수도 있습니다. 이 알고리즘에 대해 현존하는 가장 오래된 기술은 3세기에서 5세기 사이에 중국에서 쓰인 *손자산경*과, 기원후 500년경에 쓰인 아스칼론의 에우토키오스의 아르키메데스 *원의 측정* 주석에 나타나지만, 이 알고리즘이 훨씬 이전부터 알려져 있었다는 증거가 있습니다. 에우토키오스는 이 방법이 기원전 300년경에 살았던 페르가의 아폴로니오스가 쓴 *오키토키온*(Ὠκυτόκιον)이라는 제목의 논저에서 나왔다고 밝히는데, 이 논저는 지금 전해지지 않습니다.[7] 수메르인들은 이미 기원전 2600년에 점토판에 곱셈표를 기록했는데, 이는 그들이 격자 알고리즘을 사용했을 수도 있음을 시사합니다.[8]

격자 알고리즘은 입력 수가 명시적인 숫자 문자열로 표현되어 있다고 가정합니다. 여기서는 10진법으로 작업한다고 가정하겠지만, 이 알고리즘은 다른 어떤 진법으로도 곧바로 일반화됩니다. 표기를 단순화하기 위해,[9] 입력은 한 쌍의 배열 $X[0..m-1]$과 $Y[0..n-1]$로 이루어지며, 이들은 다음 수를 나타냅니다.

$$x = \sum_{i=0}^{m-1} X[i] \cdot 10^i \qquad \text{그리고} \qquad y = \sum_{j=0}^{n-1} Y[j] \cdot 10^j,$$

그리고 마찬가지로 출력은 하나의 배열 $Z[0..m+n-1]$로 이루어지며, 이 배열은 다음 곱을 나타냅니다

$$z = x \cdot y = \sum_{k=0}^{m+n-1} Z[k] \cdot 10^k.$$

[7] 말 그대로 "빠르고 순조로운 출산을 돕는 약"이라는 뜻입니다! 알렉산드리아의 파포스는 에우토키오스보다 약 200년 앞서 *오키토키온*의 몇몇 대목을 옮겨 실었지만, 격자 곱셈 알고리즘에 대한 그의 기술(만약 남겼다면) *역시* 전해지지 않습니다.

[8] 고대 수메르인들이 60진법 자릿값 기수법을 사용하여 매우 큰 수를 정확하게 계산했다는 증거는 충분하지만, 그들이 실제로 사용한 방법을 기록한 자료가 남아 있다는 이야기는 들어 본 적이 없습니다. 일반적인 곱셈표와 역수표 외에도 1부터 59까지의 정수의 제곱을 나열한 표가 발견되었으며, 이 때문에 일부 수학사학자들은 바빌로니아인들이 $xy = ((x+y)^2 - x^2 - y^2)/2$와 같은 항등식을 사용하여 곱셈을 했을 것이라고 추측합니다. 그러나 이 요령은 $x+y < 60$일 때만 통합니다. $x \geq 60$일 때 바빌로니아인들이 x^2을 어떻게 계산했을지에 대해서는 역사가 아무것도 말해 주지 않습니다.

[9] 다만 그리스와 이집트, 릴리퍼트와 블레푸스쿠, 맥과 PC, 혹은 0이 자연수라고 생각하는 사람들과 틀린 사람들 사이의 역사적 적대감에 불을 지필 위험을 무릅쓰고

이 알고리즘은 덧셈과 *한 자리* 곱셈을 기본 연산으로 사용합니다. 덧셈은 간단한 for 루프로 수행할 수 있습니다. 실제로 한 자리 곱셈은 조회표를 이용해 수행하는데, 이 조회표는 점토판에 새기거나, 나무나 대나무 조각에 칠하거나, 종이에 적거나, 읽기 전용 메모리에 저장하거나, 계산하는 사람이 암기합니다. 격자 알고리즘 전체는 다음 공식으로 요약할 수 있습니다

$$x \cdot y \;=\; \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \bigl(X[i] \cdot Y[j] \cdot 10^{i+j}\bigr).$$

격자 알고리즘의 여러 변형은 부분곱 $X[i] \cdot Y[j] \cdot 10^{i+j}$를 서로 다른 순서로 계산하며, 그 합을 구하는 데에도 서로 다른 전략을 사용합니다. 예를 들어 피보나치는 *Liber Abaco*에서 mn개의 부분곱을 자릿값이 낮은 것부터 높은 것 순으로 고려하는 변형을 설명하는데, 이를 현대적인 의사코드로 나타내면 아래와 같습니다.

```
FibonacciMultiply(X[0..m − 1], Y[0..n − 1]):
  hold ← 0
  for k ← 0 to n + m − 1
      for all i and j such that i + j = k
          hold ← hold + X[i] · Y[j]
      Z[k] ← hold mod 10
      hold ← ⌊hold/10⌋
  return Z[0..m + n − 1]
```

피보나치의 알고리즘은 흔히 모든 부분곱을 2차원 표(흔히 "타블로(tableau)", "격자(grate)" 또는 "래티스(lattice)"라고 부릅니다)에 저장한 다음, 그림 0.1의 오른쪽에 나타낸 것처럼 적절히 올림을 하면서 대각선을 따라 더하는 방식으로 수행됩니다. 미국 초등학생들은 그림 0.1의 왼쪽에 나타낸 것처럼 한 인수("피승수")에 다른 인수("승수")의 각 자릿수를 곱하되, 피승수와 각 자릿수의 곱을 모두 적은 다음에 이를 더하도록 배웁니다. 이는 에우토키오스가 설명한 방법이기도 했습니다. 다만 그는 그림 0.2에 나타낸 것처럼 걸맞게도 승수의 자릿수를 왼쪽에서 오른쪽으로 다루었습니다. 이 두 변형(및 몇 가지 다른 변형)은 모두 서양 최초의 *인쇄된* 수학책이자 *Treviso Arithmetic*(트레비소 산술서)으로도 알려진 1458년의 작자 미상 교과서 *L'Arte dell'Abbaco*에 나란히 설명되고 그림으로 제시되어 있습니다.

격자 알고리즘의 이 모든 변형은—그리고 손자(Sunzi), al-Khwārizmī, 피보나치, *L'Arte dell'Abbaco*를 비롯한 여러 문헌에서 설명하는 다른 비슷한 변형들도—임의의 m자리 수와 임의의 n자리 수의 곱을 $O(mn)$ *시간*에 계산합니다. 모든 변형의 실행 시간은 한 자리 수 곱셈의 횟수가 좌우합니다.

배가와 반분

격자 알고리즘은 직접적인 기록 증거가 남아 있는 곱셈 알고리즘 중 가장 오래된 것이 아닙니다. 이보다 더 오래되었고 어쩌면 더 단순하다고도 할 수 있는 알고리즘이 있는데, 이 알고리즘은 자릿값 표기법에 의존하지 않으며 때로는 *러시아 농부 곱셈*, *에티오피아 농부 곱셈*, 또는 그냥 ***농부 곱셈***이라고 불립니다.이 알고리즘의 한 변형은 기원전

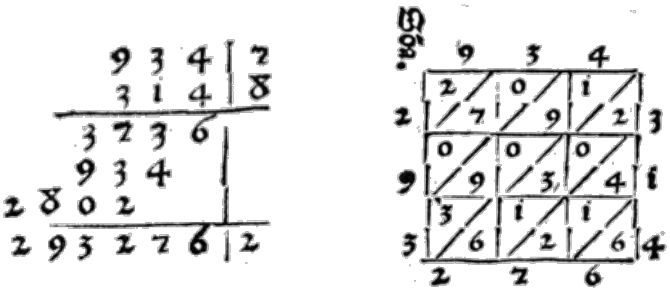


그림 0.1. *L'Arte dell'Abbaco*(1458)에 실린 그림으로, "긴" 곱셈(구거법을 이용한 오류 검산 포함)과 "격자" 곱셈을 사용하여 934 × 314 = 293276을 계산하는 과정을 보여 줍니다. (책 말미의 이미지 출처를 참조하십시오.)

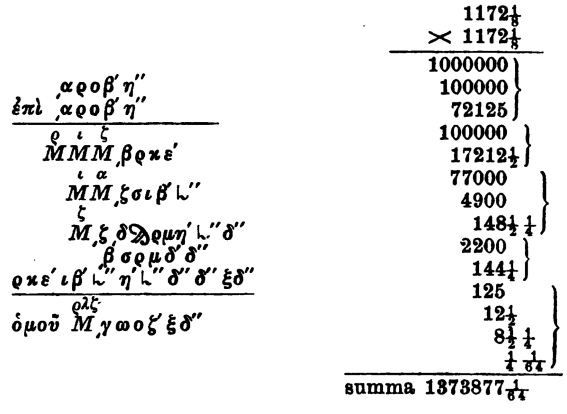


그림 0.2. 아르키메데스의 *원의 측정*에 붙인 주석에서 에우토키오스가 6세기에 수행한 $1172\frac{1}{8} \times 1172\frac{1}{8} = 1373877\frac{1}{64}$ 계산을 요한 하이베르(1891)가 필사하고(왼쪽) 현대 표기법으로 옮긴(오른쪽) 것입니다. (책 끝의 이미지 출처를 참조하십시오.)

1650년경 이집트의 서기관 아메스(Ahmes)가 린드 파피루스(Rhind papyrus)에 옮겨 적은 것으로, 그는 원본 문서가 (당시 기준으로) 약 350년 된 것이라고 주장했습니다.[10] 이 알고리즘은 20세기 후반까지도 동유럽의 초등학교에서 가르쳤으며, 정수 곱셈을 하드웨어에서 직접 구현하지 않았던 초기 디지털 컴퓨터에서도 흔히 사용되었습니다.

농부 곱셈 알고리즘은 임의의 수를 곱하는 어려운 작업을 (1) 홀짝성 판별(짝수인지 홀수인지), (2) 덧셈, (3) ***배가***(수를 두 배로 만들기), (4) ***반분***(수를 절반으로 나누고 내림하기)이라는 네 가지 더 단순한 연산의 나열로 환원합니다.

이 알고리즘의 정확성은 모든 음이 아닌 정수 x와 y에 대해 성립하는 다음의 재귀적

[10] 고대 이집트에서 실제로 사용된 이 알고리즘의 버전은 반으로 나누기나 홀짝 판별을 사용하지 않지만, 비교는 사용합니다. 반으로 나누기를 피하기 위해 이 알고리즘은 두 배 하기를 반복하여 두 개의 표를 미리 계산합니다. 하나는 x를 넘지 않는 2의 거듭제곱을 모두 담고, 다른 하나는 같은 2의 거듭제곱에 y를 곱한 값을 담습니다. 그런 다음 탐욕적 뺄셈으로 합이 x가 되는 2의 거듭제곱들을 찾고, 다른 표에서 이에 대응하는 항목들을 더하여 곱을 구합니다.

```
PeasantMultiply(x, y):
  prod ← 0
  while x > 0
      if x is odd
          prod ← prod + y
      x ← ⌊x/2⌋
      y ← y + y
  return prod
```

x	y		$prod$
			0
123	+456	=	456
61	+912	=	1368
30	~~1824~~		
15	+3648	=	5016
7	+7296	=	12312
3	+14592	=	26904
1	+29184	=	**56088**

그림 0.3. 배가법과 반분법을 이용한 곱셈

항등식으로부터 귀납법에 의해 도출됩니다.

$$x \cdot y = \begin{cases} 0 & \text{만약} x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{만약} x \text{짝수입니다.} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{만약} x \text{홀수입니다} \end{cases}$$

이 점화식이 곧 농부 곱셈 알고리즘이라고 해도 과언이 아닙니다. 반복문으로 작성된 의사코드에 속지 마십시오. 이 알고리즘은 근본적으로 재귀적입니다!

제시된 그대로라면 PeasantMultiply는 $O(\log x)$번의 홀짝 판정, 덧셈, 반분 연산을 수행하지만, $x > y$일 때 두 인자를 서로 바꾸면 이 상한을 $O(\log \min\{x, y\})$로 개선할 수 있습니다. 수를 합리적인 자릿값 표기법(2진법, 10진법, 바빌로니아의 60진법, 이집트의 12진법, 로마 숫자, 중국의 산가지, 주판의 알 위치 등)으로 나타낸다고 가정하면, 각 연산에는 최대 $O(\log(xy)) = O(\log \max\{x, y\})$번의 한 자릿수 연산이 필요합니다. 따라서 알고리즘의 전체 실행 시간은 $O(\log \min\{x, y\} \cdot \log \max\{x, y\}) = O(\log x \cdot \log y)$입니다.

다시 말해, 이 알고리즘은 m자리 수와 n자리 수를 곱하는 데 $O(mn)$ *시간*이 필요합니다. 상수 배를 무시하면 이는 격자 알고리즘과 같은 실행 시간입니다. 이 알고리즘을 손으로 실행하려면 격자 알고리즘보다 (상수 배만큼!) 더 많은 필기 작업이 필요하지만, 필요한 기본 연산은 사람이 수행하기에 더 쉽다고 할 수 있습니다. 사실 수를 이진법으로 나타내면 두 알고리즘은 동등합니다.

컴퍼스와 눈금 없는 자

고대 그리스 기하학자들은 수(더 정확히 말하면 *크기*)를 적절한 길이의 선분과 동일시하였고, 이 선분을 컴퍼스와 눈금 없는 자라는 두 가지 간단한 기계적 도구로 다루었습니다. 이 도구들은 여러 형태로 이미 수 세기 전부터 측량사, 건축가, 그 밖의 장인들이 흔히 사용해 온 것이었습니다. 이 학자들은 *오직* 이 두 도구만을 사용하여, 하나 이상의 주어진 기준점에서 출발해 여러 복잡한 기하학적 작도를 다음과 같은 기본 연산으로 환원하였습니다.

- 식별된 서로 다른 두 점을 지나는 유일한 직선을 그립니다.

- 식별된 한 점을 중심으로 하고 식별된 다른 점을 지나는 유일한 원을 그리십시오.
- 두 직선의 교점(존재하는 경우)을 구하십시오.
- 직선과 원의 교점이 있다면 그 교점을 모두 구하십시오.
- 두 원의 교점을 (존재한다면) 구하십시오.

실제로 그리스의 기하학 학생들은 거의 틀림없이 먼지나 모래로 덮인 탁자인 *아박스* (ἄβαξ) 위에 작도를 그렸을 것입니다.[11] 그보다 수 세기 전에 이집트의 측량사들은 밧줄을 사용해 땅 위에 직선과 원을 정하는 방식으로 같은 작도의 상당수를 수행했습니다.[12] 그러나 유클리드를 비롯한 그리스 기하학자들은 컴퍼스와 눈금 없는 자를 이용한 작도를 정밀한 수학적 *추상*으로 제시했습니다. 즉 점은 *이상적인* 점이고, 직선은 *이상적인* 직선이며, 원은 *이상적인* 원입니다.

그림 0.4는 약 2500년 전 유클리드의 *원론*에 기술된, 두 크기를 곱하거나 나누는 알고리즘을 보여 줍니다. 입력은 서로 다른 네 점 A, B, C, D로 이루어지며, 목표는 $|AZ| = |AC||AD|/|AB|$를 만족하는 점 Z를 작도하는 것입니다. 특히 $|AB|$를 길이의 단위로 정의하면, 이 알고리즘은 $|AC|$와 $|AD|$의 곱을 계산합니다.

유클리드가 먼저 (현대 프로그래머의 표현을 빌리자면) 서브루틴을 작성하여 새로운 기본 연산 RightAngle을 정의한다는 점에 주목하십시오. 정확성은 알고리즘의

```
《P를 지나고 ℓ에 수직인 직선을 작도하십시오.》
RightAngle(ℓ, P):
  Choose a point A ∈ ℓ
  A, B ← Intersect(Circle(P, A), ℓ)
  C, D ← Intersect(Circle(A, B), Circle(B, A))
  return Line(C, D)

《|AZ| = |AC||AD|/|AB|를 만족하는 점 Z를
  작도하십시오.》
MultiplyOrDivide(A, B, C, D):
  α ← RightAngle(Line(A, C), A)
  E ← Intersect(Circle(A, B), α)
  F ← Intersect(Circle(A, D), α)
  β ← RightAngle(Line(E, C), F)
  γ ← RightAngle(β, F)
  return Intersect(γ, Line(A, C))
```

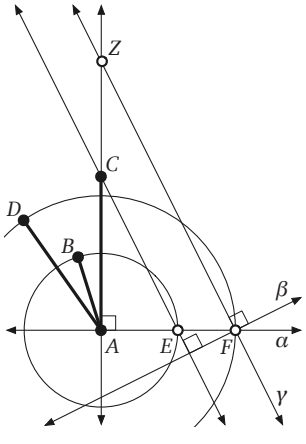


그림 0.4. 컴퍼스와 자를 이용한 곱셈.

정확성은 삼각형 ACE와 AZF가 닮음이라는 관찰에서 따라 나옵니다. 주 알고리즘의 둘째 줄과 셋째 줄은 모호합니다. α는 A를 중심으로 하는 임의의 원과 두 개의 서로

[11] 1부터 9까지의 필기 숫자는 피보나치의 *산반서(Liber Abaci)*보다 적어도 두 세기 앞서 유럽에 "고바르 숫자(gobar numerals)"라는 이름으로 알려져 있었습니다. 이 이름은 먼지를 뜻하는 아랍어 단어 *ghubār*에서 왔으며, 궁극적으로는 모래로 덮인 탁자 위에서 산술 계산을 하던 인도의 관습을 가리킵니다. 그리스어 단어 ἄβαξ는 라틴어 *abacus*의 어원이며, 이 라틴어 단어 역시 원래는 모래 탁자를 가리켰습니다.

[12] "기하학(geometry)"이 무슨 뜻인지 기억하십니까? 훗날 데모크리토스는 이 이집트 측량사들을 다소 경멸적으로 "밧줄을 매는 사람들"이라는 뜻의 *아르페도납타이*(ἀρπεδονάπται)라고 불렀습니다.

다른 점에서 만나기 때문입니다. 하지만 E와 F로 어느 교점을 고르더라도 알고리즘은 실제로 올바르게 동작합니다.

유클리드의 알고리즘은 두 크기(길이)를 곱하는 문제를 일련의 기본적인 컴퍼스와 눈금 없는 자 연산으로 환원합니다. 이러한 연산은 현대의 디지털 컴퓨터에서 정확하게 구현하기 어렵지만, 유클리드의 알고리즘은 디지털 컴퓨터를 위해 *설계된* 것이 아니었습니다. 이 알고리즘은 플라톤적 이상 컴퍼스와 플라톤적 이상 눈금 없는 자를 다루며 *정의상* 각 연산을 상수 시간에 완벽하게 수행할 수 있는 플라톤적 이상 기하학자를 위해 설계되었습니다. 이 계산 모델에서 MultiplyOrDivide는 $O(1)$ 시간에 실행됩니다!

0.3 의회 의석 배분

다음은 정치적으로 매우 중요한 알고리즘의 또 다른 실제 사례입니다. 미국 헌법 제1조 제2절은 다음과 같이 규정합니다.

> 하원의원 수와 직접세는 이 연방에 가입하는 각 주에 그 인구수에 비례하여 배정합니다.... 하원의원 수는 인구 3만 명당 1명을 초과할 수 없으나, 각 주는 적어도 1명의 하원의원을 두어야 합니다....

하원의 의석 수는 한정되어 있으므로 *정확한* 비례 대표를 실현하려면 의원을 공유하거나 분수 단위의 의원을 두어야 하지만, 두 방식 모두 법적으로 허용되지 않습니다. 그 결과 이후 수십 년에 걸쳐 이상적인 분수 해를 공정하게 반올림하기 위한 다양한 의석 배분 알고리즘이 제안되고 사용되었습니다. 오늘날 실제로 사용되는 알고리즘은 ***헌팅턴-힐 방법*** 또는 ***균등 비율에 의한 방법***이라고 불리며, 1911년 인구조사국 통계학자 조지프 힐이 처음 제안했고, 1920년 하버드 수학자 에드워드 헌팅턴이 다듬었으며, 1941년 연방법(2 U.S.C. §2a)에 채택되었고, 1992년 대법원에서 제기된 위헌 소송에서도 살아남았습니다.[13]

헌팅턴-힐 방식은 의석을 한 번에 하나씩 주(state)에 배분합니다. 먼저 전처리 단계에서 각 주에 의석을 하나씩 배분합니다. 그런 다음 메인 루프의 각 반복에서 다음 의석을 *우선순위*가 가장 높은 주에 배정합니다. 각 주의 우선순위는 $P/\sqrt{r(r+1)}$로 정의합니다. 여기서 P는 그 주의 인구이고, r은 그 주에 이미 배분된 의석의 수입니다.

이 알고리즘의 의사코드는 그림 0.5에 나와 있습니다. 입력은 n개 주의 인구를 저장하는 배열 $Pop[1..n]$과 대표자 총수를 나타내는 정수 R로 이루어지며, 알고리즘은 $R \geq n$이라고 가정합니다. (현재 미국에서는 $n = 50$이고 $R = 435$입니다.) 출력 배열 $Rep[1..n]$은 각 주에 배정된 대표자 수를 기록합니다.

이 헌팅턴-힐 구현은 NewPriorityQueue, Insert, ExtractMax 연산을 지원하는 우선순위 큐를 사용합니다. (물론 실제 법률은 우선순위 큐에 대해 아무것도

[13] 대법원은 앞서 연방 지방법원이 내린 판결을 뒤집고, 의회가 선의로 채택한 의석 배분 방법이라면 ***어떤*** 것이든 합헌이라고 만장일치로 판결했습니다(*유나이티드 States Department of Commerce v. Montana*). 현재 의회 의석 배분 알고리즘은 미국 인구조사국 웹사이트 http://www.census.gov/topics/public-sector/congressional-apportionment.html 에 끔찍할 정도로 자세히 설명되어 있습니다. 의석 배분 문제의 역사는 http://www.thirty-thousand.org/pages/Apportionment.htm 에 잘 정리되어 있습니다. 여러 의석 배분 방법을 설명하는 의회조사국(Congressional Research Service)의 보고서는 http://www.fas.org/sgp/crs/misc/R41382.pdf 에서 볼 수 있습니다.

```
ApportionCongress(Pop[1 .. n], R):
  PQ ← NewPriorityQueue
《모든 주에 첫 번째 대표를 배정하십시오》
  for s ← 1 to n
      Rep[s] ← 1
      Insert(PQ, s, Pop[i]/√2)
《남은 n − R명의 대표를 배정합니다》
  for i ← 1 to n − R
      s ← ExtractMax(PQ)
      Rep[s] ← Rep[s] + 1
      priority ← Pop[s] / √(Rep[s](Rep[s] + 1))
      Insert(PQ, s, priority)
  return Rep[1 .. n]
```

그림 0.5. 헌팅턴-힐 의석 배분 알고리즘

말하지 않습니다.) 알고리즘의 출력, 따라서 그 정확성은 이 우선순위 큐를 어떻게 구현하는지에 *전혀* 의존하지 않습니다. 인구조사국은 Excel 스프레드시트의 단일 열에 저장된 정렬된 배열을 사용하며, 이 배열은 매 반복마다 처음부터 다시 계산됩니다. 여러분은 학부 자료구조 수업에서 더 효율적인 구현을 배웠습니다(배웠어야 합니다).

이와 비슷한 의석 배분 알고리즘은 전 세계의 다당제 의회 선거에서도 사용됩니다. 이러한 선거에서는 각 정당에 배분되는 의석 수가 그 정당의 득표수에 비례해야 합니다. 가장 널리 쓰이는 두 방법은 *동트 방식*[14]과 *웹스터–생라그 방식*[15]입니다. 이 두 방식은 헌팅턴-힐 방식의 제곱근 식 대신 각각 $P/(r+1)$과 $P/(2r+1)$을 우선순위로 사용합니다. 헌팅턴-힐 방식은 사실상 미국 하원에서만 사용됩니다. 이는 각 주에 적어도 한 명의 하원의원을 배분해야 한다는 헌법상의 요건 때문이기도 합니다.

0.4 나쁜 예

실제로는 알고리즘이 *아닌* 명령 나열의 전형적인 예로 "마틴의 알고리즘"을 생각해 봅시다:[16]

```
BeAMillionaireAndNeverPayTaxes():
  Get a million dollars.
  If the tax man comes to your door and says, "You have never paid taxes!"
      Say "I forgot."
```

[14] 1792년 토머스 제퍼슨이 고안하였고, 1792년부터 1832년까지 미국 하원 의석 배분에 사용되었으며, 1878년 벨기에의 수학자 빅토르 동트가 재발견하였고, 1888년 스위스의 물리학자 에두아르트 하겐바흐비쇼프가 개량하였습니다.

[15] 1832년 대니얼 웹스터가 고안하였고, 1842년부터 1911년까지 미국 하원 의석 배분에 사용되었으며, 1910년 프랑스의 수학자 앙드레 생라그가 재발견하였고, 1980년 독일의 물리학자 한스 셰퍼스가 다시 한번 재발견하였습니다.

[16] Steve Martin, "You Can Be A Millionaire", *새터데이 나이트 라이브*, 1978년 1월 21일. *코미디는 아름답지 않습니다*, Warner Bros. Records, 1979에도 수록되어 있습니다.

첫 단계만 빼면 꽤 간단합니다. 하지만 그 첫 단계가 정말 만만치 않습니다! 억만장자 CEO나 실리콘밸리 벤처 투자자, 뉴욕시 부동산 투기꾼 같은 사람들은 이것을 알고리즘으로 여길지도 모릅니다. 그들에게는 첫 단계가 명확하면서도[17] 자명하기 때문입니다. 하지만 나머지 불쌍한 우리 서민들에게 마틴의 절차는 실제 알고리즘으로 보기에는 너무 모호합니다. 한편 이것은 ***환원***의 완벽한 예입니다. 이 절차는 백만장자가 되면서 세금을 한 푼도 내지 않는 문제를 백만 달러를 손에 넣는 "더 쉬운" 문제로 *환원*합니다. 이 책에서는 환원을 거듭해서 보게 될 것입니다. 수백 명의 사업가와 정치인이 보여 주었듯이, 더 쉬운 문제를 푸는 방법을 알고 있다면 더 어려운 문제를 푸는 방법은 환원이 알려 줍니다.

마틴의 알고리즘은 앞서 살펴본 몇몇 예시와 마찬가지로 컴퓨터 과학자들이 익숙하게 생각하는 종류의 알고리즘이 아닙니다. 컴퓨터가 수행하기 어려운 연산으로 기술되어 있기 때문입니다. 이 책은 (거의!) 오로지 표준 디지털 컴퓨터에서 무리 없이 구현할 수 있는 알고리즘만을 다룹니다. 이러한 알고리즘의 각 단계는 일반적인 프로그래밍 언어가 직접 지원하는 것(산술 연산, 대입, 반복문, 재귀 등)이거나, 여러분이 이미 하는 법을 배운 것(정렬, 이진 탐색, 트리 순회, "벽에 걸린 맥주 n병" 부르기 등)입니다.

0.5 알고리즘 기술하기

알고리즘을 효과적으로 *설계하고 분석하는* 데 필요한 기술은 알고리즘을 효과적으로 *서술하는* 데 필요한 기술과 서로 얽혀 있습니다. 적어도 제 수업에서는 어떤 알고리즘이든 그에 대한 완전한 서술은 다음 네 가지 요소로 이루어집니다:

- **무엇:** 알고리즘이 해결하는 문제의 정확한 명세입니다.
- **방법:** 알고리즘 자체에 대한 정확한 설명입니다.
- **왜:** 알고리즘이 해결해야 할 문제를 실제로 해결한다는 증명입니다.
- **얼마나 빠른가:** 알고리즘의 실행 시간 분석입니다.

이 네 가지 구성 요소를 반드시 이 순서대로 *개발*해야 하는 것은 아니며, 그렇게 하는 것이 바람직하지도 않습니다. 문제 명세, 알고리즘 설명, 정확성 증명, 시간 분석은 대개 동시에 발전하며, 각 구성 요소를 개발하는 과정이 나머지 구성 요소의 개발에 영향을 줍니다. 예를 들어, 더 빠른 알고리즘을 뒷받침하기 위해 문제 설명을 조금 고쳐야 할 수도 있고, 정확성 증명의 까다로운 경우를 처리하기 위해 알고리즘을 수정해야 할 수도 있습니다. 그럼에도 불구하고 이 구성 요소들을 따로따로 *제시*하는 것이 대개 독자에게 가장 명확합니다.

모든 글쓰기가 그렇듯이, 설명은 알맞은 독자를 겨냥하는 것이 중요합니다. 저는 유능하지만 회의적이며 *여러분만큼 영리하지는 않은* 프로그래머를 독자로 삼아 쓰기를 권합니다. 6개월 전의 자신을 떠올려 보십시오. 새로운 알고리즘을 개발하다 보면 여러분은 자연스럽게 문제에 대해, 그리고 여러분의 알고리즘이 그 문제를 어떻게 푸는지에 대해 많은 직관을 쌓게 되고, 여러분의 비형식적인 추론은 그 직관의 인도를

[17] 어쩌고저쩌고 보안 양자 블록체인 딥러닝 어쩌고.

받게 됩니다. 그러나 나중에 여러분의 알고리즘이나 그로부터 도출한 코드를 *읽는* 사람은 누구도 여러분의 직관이나 경험을 공유하지 못합니다. 여러분의 컴파일러도 마찬가지입니다. 6개월 뒤의 여러분 자신도 마찬가지입니다. 그들이 가진 것은 여러분이 글로 쓴 설명뿐입니다.

여러분의 알고리즘을 다른 누구에게도 설명할 일이 없더라도, 독자를 염두에 두고 알고리즘을 개발하는 것은 여전히 중요합니다. 명확하게 전달하려고 노력하다 보면 더 명확하게 *생각*할 수밖에 없습니다. 특히 여러분의 글을 쓰인 그대로 *정확히* 해석할 *초보자* 독자를 위해 글을 쓰면, 여러분의 상위 수준 아이디어가 지금 당장은 아무리 "자명" 하거나 "직관적" 으로 보이더라도 세부 사항을 하나하나 짚어 가며 해결할 수밖에 없습니다. 마찬가지로 *회의적인* 독자를 위해 글을 쓰면, 여러분의 직관이나 지능을 믿는 대신 정확성과 효율성에 대한 탄탄한 논증을 세울 수밖에 없습니다.[18]

이 점은 아무리 강조해도 지나치지 않습니다: **알고리즘 설계자로서 여러분의 가장 중요한 일은 여러분의 알고리즘이 어떻게, 그리고 왜 작동하는지를 *다른 사람들에게 가르치는 것*입니다.** 여러분의 아이디어를 다른 사람들에게 전달할 수 없다면, 그 아이디어는 존재하지 않는 것이나 마찬가지입니다. 올바르고 효율적인 실행 코드를 만드는 것은 중요하지만 부차적인 목표입니다. 여러분 자신, 교수님, (장래의) 고용주, 동료, 또는 학생들에게 여러분이 똑똑하다는 것을 납득시키는 일은 기껏해야 한참 뒤처진 세 번째 목표에 불과합니다.

문제 명세하기

새로운 알고리즘을 개발하기 시작하기도 전에, 우리는 먼저 그 알고리즘이 어떤 문제를 풀어야 하는지에 대해 합의해야 합니다. 마찬가지로, 알고리즘을 *서술*하기 시작하기도 전에, 우리는 그 알고리즘이 풀어야 할 문제를 먼저 *서술*해야 합니다.

알고리즘 문제는 흔히 현실 세계의 대상을 들어 일상적인 영어로 제시됩니다. 이러한 문제를 수, 배열, 리스트, 그래프, 트리 등 우리가 형식적으로 추론할 수 있는 형식적이고 추상적인 *수학적* 대상으로 다시 서술하는 것은 알고리즘 설계자인 우리의 몫입니다. 또한 문제 서술에 숨겨진 가정이 있는지 판단하고, 그러한 가정을 명시적으로 밝혀야 합니다. (예를 들어, 노래 "벽에 걸린 맥주 n병" 에서 n은 언제나 음이 아닌 정수입니다.[19])

알고리즘을 개발해 나가면서 명세를 다듬어야 할 수도 있습니다. 예를 들어, 우리 알고리즘은 원래의 비형식적인 문제 설명에서 명시되지 않았던 특정한 입력 표현을 요구하거나 특정한 출력 표현을 만들어 낼 수도 있습니다. 또는 우리 알고리즘이 실제로는 원래 풀도록 요청받은 문제보다 *더 일반적인* 문제를 풀 수도 있습니다. (이는 재귀 알고리즘에서 흔히 볼 수 있는 특징입니다.)

명세에는 알고리즘이 실제로 어떻게, 또 왜 동작하는지 모르는 다른 사람도 우리 알고리즘을 블랙박스로 *사용*할 수 있을 만큼의 세부 사항만 담아야 합니다. 특히 각 입력 매개변수의 자료형*과 의미*, 그리고 최종 출력이 입력 매개변수에 정확히 어떻게

[18] 특히 저는 *여러분*이 회의적인 초보자라고 가정합니다!

[19] "벽에 걸린 맥주 $\sqrt{2}$병"을 부르는 사람은 한 번도 들어 본 적이 없습니다. 집합론자들이 "벽에 걸린 맥주 $\aleph_0$병"을 부르는 것은 가끔 *들어 본* 적이 있지만, 어째서인지 그들은 언제나 노래가 끝나기 전에 포기하고 말았습니다.

의존하는지를 기술해야 합니다. 반면에 명세는 알고리즘을 블랙박스로 사용하는 데 필요하지 *않은* 세부 사항을 모두 *의도적으로 숨겨야* 합니다. 중요하지 않은 것은 진정으로 흘려보냅시다.

예를 들어, 격자 알고리즘과 배가-반분 알고리즘은 모두 같은 문제를 풉니다. 즉, 각각 자릿수의 배열로 표현된 두 음이 아닌 정수 x와 y가 주어졌을 때, 곱 $x \cdot y$를 역시 자릿수의 배열로 표현하여 계산하는 문제입니다. 이 알고리즘을 *사용하는* 사람에게는 어떤 알고리즘을 선택하는지가 전혀 중요하지 않습니다. 반면에 그리스의 눈금 없는 자와 컴퍼스 알고리즘은 *다른 문제*를 풉니다. 입력값과 출력값이 자릿수의 배열이 아니라 선분으로 표현되기 때문입니다.

알고리즘 설명하기

컴퓨터 프로그램은 알고리즘을 구체적으로 표현한 것이지만, 알고리즘이 곧 프로그램인 것은 *아닙니다*. 오히려 알고리즘은 바탕이 되는 기본 연산을 지원하는 *어떤* 프로그래밍 언어로도 구현할 수 있는 추상적이고 기계적인 절차입니다. 여러분이 가장 좋아하는 프로그래밍 언어의 독특한 문법적 세부 사항은 전혀 중요하지 않습니다. 그런 것에 집중하면 *실제로* 무슨 일이 일어나고 있는지에서 여러분(과 여러분의 독자)의 주의를 흐트러뜨릴 뿐입니다.[20] 좋은 알고리즘 설명은 코드 자체보다는 실제 프로그램의 *주석*에 써야 할 내용에 더 가깝습니다. 코드는 이야기를 전달하기에 좋지 않은 매체입니다.

반면에, 평이한 영어 산문으로 설명하는 것 역시 대개 좋은 방법이 아닙니다. 알고리즘에는 관용적인 구조가 많습니다. 특히 조건문, 반복문, 함수 호출, 재귀가 그렇습니다. 이러한 구조는 짜임새 없는 산문 속에 너무나 쉽게 묻혀 버립니다. 일상적인 영어에는 모호함과 미묘한 뉘앙스가 가득하지만, 알고리즘은 가능한 한 모호함 없이 기술해야 합니다. 산문은 정밀함을 담기에 부적합한 매체입니다.

제 생각에 알고리즘을 제시하는 가장 명확한 방법은 *의사코드*와 구조화된 영어를 함께 사용하는 것입니다. 의사코드는 형식적인 프로그래밍 언어와 수학의 *구조*를 사용하여 알고리즘을 기본 단계들로 나눕니다. 이 기본 단계 자체는 수학적 표기법이나 순수한 영어, 또는 둘을 적절히 섞어서 쓸 수 있으며, *무엇이든 가장 명확한 방식*을 택하면 됩니다. 잘 작성된 의사코드는 알고리즘의 내부 구조를 드러내면서도 관련 없는 구현 세부 사항은 감추므로, 알고리즘을 더 쉽게 이해하고, 분석하고, 디버깅하고, 구현할 수 있게 해 줍니다.

알고리즘을 설명할 때는 언제나 알고리즘을 완전히 명세하고, 그 정확성을 증명하고, 실행 시간을 분석하는 데 필요한 모든 세부 사항을 설명에 포함해야 합니다.

[20] 물론 이것은 종교적 신념의 문제입니다. 안락의자 언어학자들은 *사피어-워프 가설*을 두고 끊임없이 논쟁합니다. 이 가설은 (대략) 사람들이 자신의 언어가 부과하는 범주 안에서만 생각한다고 주장합니다. 이 원리를 극단적으로 정식화하면, 한 언어의 어떤 개념은 다른 언어 사용자가 도저히 이해할 수 없는데, 이는 단지 기술 발전 때문만이 아니라("jump the shark"나 "Fortnite streamer"를 아람어로 어떻게 번역하겠습니까?) 언어와 문화 사이의 본질적인 구조적 차이 때문이라고 합니다. 보다 회의적인 견해는 스티븐 핑커의 *언어 본능*를 참고하십시오. 이 생각을 서로 다른 프로그래밍 패러다임에 적용하면 어느 정도 설득력이 있다는 점은 인정합니다. (Y 컴비네이터가 뭐였더라? 템플릿은 어떻게 동작하지? Abstract Factory는 또 뭐지?) 다행히도 그러한 차이는 너무 미묘해서 *이* 책의 내용에는 아무런 영향을 미치지 않습니다. 설득력 있는 반례로는 크리스 오카사키의 단행본 *함수형 자료구조*(함수형 자료구조)와 그 이후의 후속 연구들을 참고하십시오.

동시에 알고리즘을 완전히 명세하고, 그 정확성을 증명하고, 실행 시간을 분석하는 데 필요하지 *않은* 세부 사항은 모두 *제외*해야 합니다. (슬라이드.) 좀 더 실용적인 차원에서 말하자면, 우리의 설명은 *이 책을 읽지 않은* 유능하지만 회의적인 프로그래머가 *알고리즘이 왜 작동하는지 이해하지 못하더라도 자신이* 좋아하는 프로그래밍 언어로 그 알고리즘을 빠르고 정확하게 구현할 수 있게 해 주어야 합니다.

의사코드를 작성할 때 제가 따르는 규칙들로 여러분을 지루하게 만들고 싶지는 않지만, 특히 해로운 습관 하나만큼은 반드시 경고해야 하겠습니다. 반복되는 연산을 "먼저 [이것]을 하고, 그다음 [저것]을 하고, ***이런 식으로 계속***합니다."나 "[무언가]가 될 때까지 ***이 과정***을 반복합니다"처럼 비형식적으로 기술하는 일은 ***절대*** 하지 마십시오. "이 수열에서 다음에 올 것은 무엇입니까?" 같은 답답한 시험을 치러 본 사람이라면 누구나 이미 알고 있듯이, 알고리즘의 처음 몇 단계를 기술하는 것은 이후 단계에서 무슨 일이 일어나는지에 대해 거의 또는 전혀 알려 주지 않습니다. 알고리즘에 반복문이 있다면 반복문으로 작성하고, *임의의* 반복에서 무슨 일이 일어나는지를 명시적으로 기술하십시오. 마찬가지로 알고리즘이 재귀적이라면 재귀적으로 작성하고, 각 경우의 경계와 각 경우에 무슨 일이 일어나는지를 명시적으로 기술하십시오.

0.6 알고리즘 분석

알고리즘을 적어 놓고 "보십시오!"라고 말하는 것만으로는 충분하지 않습니다. 우리는 그 알고리즘이 실제로 해야 할 일을 하며 그 일을 효율적으로 한다는 것을 청중에게 (그리고 우리 자신에게도!) 납득시켜야 합니다.

정확성

일부 응용 환경에서는 프로그램이 모든 "합리적인" 입력에 대해 대부분의 경우에 올바르게 동작하는 것으로 충분합니다. 이 책에서는 그렇지 않습니다. 우리는 *가능한 모든* 입력에 대해 *항상* 올바른 알고리즘을 요구합니다. 더욱이 우리는 알고리즘이 올바르다는 것을 *증명*해야 합니다. 직관을 믿거나 몇 가지 테스트 케이스를 시험해 보는 것만으로는 충분하지 않습니다. 때로는 정확성이 정말로 자명한 경우도 있으며, 특히 이전 강의에서 본 알고리즘이라면 더욱 그렇습니다. 반면에 "자명하다"는 말은 너무나 자주 "틀렸다"의 동의어입니다. 이 강의에서 다루는 알고리즘 대부분은 정확성을 증명하는 데 상당한 노력이 필요합니다. 특히 정확성 증명에는 대개 귀납법이 사용됩니다. 우리는 귀납법을 *좋아합니다*. 귀납법은 우리의 *친구*입니다.[21]

물론, 우리 알고리즘이 해야 할 일을 제대로 수행한다는 것을 형식적으로 증명하려면, 먼저 그 알고리즘이 무엇을 해야 하는지부터 형식적으로 기술해야 합니다!

실행 시간

같은 문제를 푸는 여러 알고리즘의 순위를 매길 때 가장 흔히 쓰는 기준은 알고리즘이 얼마나 빨리 실행되는가입니다. 이상적으로는 어떤 문제에 대해서든 가능한 한 가장 빠른 알고리즘을 원합니다. 많은 응용 환경에서는 프로그램이 모든 "합리적인" 입력에

[21] 귀납법이 여러분의 친구가 *아니라면*, 이 책을 공부하기가 힘들 것입니다.

대해 대부분의 경우 효율적으로 실행되기만 해도 괜찮습니다. 하지만 이 책에서는 그렇지 않습니다. 우리는 최악의 경우에도 *항상* 효율적으로 실행되는 알고리즘을 요구합니다.

그렇다면 실행 시간은 어떻게 측정합니까? 구체적인 예로, BottlesOfBeer(n) 노래를 부르는 데에는 시간이 얼마나 걸립니까? 이는 분명히 입력값 n의 함수이지만, 노래를 얼마나 빨리 부를 수 있는지에도 달려 있습니다. 어떤 가수는 한 절을 부르는 데 10초가 걸릴 수 있고, 다른 가수는 20초가 걸릴 수도 있습니다. 기술은 그 가능성의 폭을 더욱 넓힙니다. 모스 부호를 사용하여 전신으로 노래를 불러 주면 한 절에 꼬박 1분이 걸릴 수도 있습니다. 웹에서 mp3를 내려받으면 한 절에 10분의 1초가 걸릴 수도 있습니다. 컴퓨터의 주기억장치에서 mp3를 복제하면 한 절에 불과 몇 마이크로초밖에 걸리지 않을 수도 있습니다.

여기서 중요한 것은 n이 커짐에 따라 노래하는 시간이 어떻게 변하는지입니다. 어떤 기술을 사용하든, BottlesOfBeer($2n$)을 부르는 데는 BottlesOfBeer(n)을 부르는 데 걸리는 시간의 약 두 배가 필요합니다. 이 사실은 점근적 노래 시간 $\Theta(n)$에 반영되어 있습니다.

알고리즘이 특정 명령어를 몇 번 실행하는지, 또는 "코드"의 특정 지점에 몇 번 도달하는지를 세어 시간을 측정할 수 있습니다. 예를 들어 BottlesOfBeer의 모든 절에서 "맥주"라는 단어가 세 번씩 불린다는 점에 주목할 수 있으며, 따라서 "맥주"를 부르는 횟수는 전체 노래 시간을 잘 나타내는 지표가 됩니다. 이 질문에는 정확한 답을 제시할 수 있습니다. BottlesOfBeer(n)은 맥주를 정확히 $3n + 3$번 언급합니다.

여담이지만, 부르는 데 이차 시간이 걸리는 노래는 *아주 많습니다*. 다음 노래는 영어 사용자 대부분에게 익숙할 것입니다:

```
NDaysOfChristmas(gifts[2..n]):
  for i ← 1 to n
      Sing "On the ith day of Christmas, my true love gave to me"
      for j ← i down to 2
          Sing " j gifts[j],"
      if i > 1
          Sing "and"
      Sing "a partridge in a pear tree."
```

NDaysOfChristmas의 입력은 $n - 1$개의 선물로 이루어진 목록이며, 여기서는 배열로 나타냅니다. 노래하는 시간이 $\Theta(n^2)$임은 꽤 쉽게 보일 수 있습니다. 구체적으로, 노래하는 사람은 선물 이름을 $\sum_{i=1}^{n} i = n(n+1)/2$번 부릅니다(배나무 위의 자고새까지 셉니다). 또한 크리스마스의 처음 n일 동안 내 사랑이 나에게 준 선물이 정확히 $\sum_{i=1}^{n}\sum_{j=1}^{i} j = n(n+1)(n+2)/6 = \Theta(n^3)$개라는 것도 쉽게 알 수 있습니다.

이 밖에도 이차 시간이 걸리는 노래로는 "Old MacDonald Had a Farm", "There Was an Old Lady Who Swallowed a Fly", "Hole in the Bottom of the Sea", "Green Grow the Rushes O", "The Rattlin' Bog", "The Court Of King Caractacus", "The Barley-Mow", "If I Were Not Upon the Stage", "Star Trekkin' ", "Ist das nicht ein Schnitzelbank?"[22], "Il Pulcino Pio", "Minkurinn í hænsnakofanum", "Echad Mi

[22] Ja, das ist Otto von Schnitzelpusskrankengescheitmeyer!

Yodea", 그리고 "Το κοκοράκι" 등이 있습니다. 더 많은 예가 궁금하다면 여러분이 가장 좋아하는 미취학 아동에게 물어보십시오.

```
ALOUETTE(lapart[1..n]):
  Chantez « Alouette, gentille alouette, alouette, je te plumerai. »
  pour tout i de 1 à n
      Chantez « Je te plumerai lapart[i]. Je te plumerai lapart[i]. »
      pour tout j de i à 1                                  《거꾸로》
          Chantez « Et lapart[j]! Et lapart[j]! »
      Chantez « Alouette! Alouette! Aaaaaa... »
      Chantez « ...alouette, gentille allouette, alouette, je te plumerai. »
```

어떤 노래들은 이보다 훨씬 더 기이한 가창 시간을 가집니다. 꽤 현대적인 예로 Guy Steele의 "The TELNET Song"이 있는데, 이 노래는 처음 n개의 절을 부르는 데 실제로 $\Theta(2^n)$ 시간이 걸립니다. Steele은 $n = 4$를 권장했습니다. 마지막으로, *결코* 끝나지 않는 노래들도 있습니다.[23]

"The TELNET Song"을 제외하면 이 노래들은 모두 몇 개의 중첩 루프로 가장 자연스럽게 표현되므로, 이 노래들의 실행 노래 시간은 중첩된 합으로 계산할 수 있습니다. *재귀* 알고리즘의 실행 시간은 점화식으로 표현하는 편이 더 쉽습니다. 예를 들어, 농부 곱셈 알고리즘은 다음과 같이 재귀적으로 표현할 수 있습니다:

$$x \cdot y = \begin{cases} 0 & \text{만약} x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{만약} x \text{짝수입니다.} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{만약} x \text{홀수입니다} \end{cases}$$

$T(x, y)$가 $x \cdot y$를 계산하는 데 필요한 홀짝 판별, 덧셈, 반분 연산의 횟수를 나타낸다고 합시다. 이 함수는 기저 사례 $T(0, y) = 0$과 함께 재귀 부등식 $T(x, y) \leq T(\lfloor x/2 \rfloor, 2y) + 2$를 만족합니다. 다음 장에서 설명하는 기법들을 적용하면 상한 $T(x, y) = O(\log x)$를 얻을 수 있습니다.

알고리즘의 실행 시간은 때때로 그 기반이 되는 자료구조나 서브루틴을 구체적으로 어떻게 구현하느냐에 따라 달라집니다. 예를 들어, 헌팅턴-힐 의석 배분 알고리즘 APPORTIONCONGRESS는 $O(N + RI + (R - n)E)$ 시간에 실행됩니다. 여기서 N은 NEWPRIORITYQUEUE의 실행 시간을, I는 INSERT의 실행 시간을, E는 EXTRACTMAX의 실행 시간을 나타냅니다. $R \geq 2n$이라는 합리적인 가정(평균적으로 각 주가 적어도 두 명의 하원의원을 배정받는다는 가정) 아래에서는 이 상한을 $O(N + R(I + E))$로 단순화할 수 있습니다. 정확한 실행 시간은 기반이 되는 우선순위 큐의 구현에 따라 달라집니다. 인구조사국은 우선순위 큐를 정렬되지 않은 배열로 구현합니다. 이 경우 $N = I = \Theta(1)$이고 $E = \Theta(n)$이므로, 인구조사국의 APPORTIONCONGRESS 구현은 $O(Rn)$ *시간*에 실행됩니다. 그러나 우선순위 큐를 이진 힙이나 힙 순서 배열로 구현하면 $N = \Theta(1)$이고 $I = E = O(\log n)$이므로, 전체 알고리즘은 $O(R \log n)$ 시간에 실행됩니다.

마지막으로, 때로는 시간 이외의 계산 자원에 관심을 가질 때도 있습니다. 공간, 동전 던지기 횟수, 캐시 부재나 페이지 부재 횟수, 프로세스 간 메시지 수, 또는 내

[23] 그 노래들은 그저 계속되고 또 계속됩니다, 친구여.

진정한 사랑이 내게 준 선물의 개수 같은 것들입니다. 이러한 자원도 실행 시간을 분석할 때 사용하는 것과 같은 기법으로 분석할 수 있습니다. 예를 들어, 두 n자리 수의 격자 곱셈은 부분곱을 더하기 전에 모두 적어 두면 $O(n^2)$ 공간이 필요하지만, 부분곱을 구하는 즉시 더해 나가면 $O(n)$ 공간만 있으면 됩니다.

연습문제

0. 표준 체스판 위에 표준 기물들이 규칙상 가능한 배치로 놓여 있을 때, 두 플레이어가 모두 완벽하게 둔다면 주어진 시작 위치에서 어느 플레이어가 체스에서 이기는지 판정하는 효율적인 알고리즘을 서술하고 분석하십시오. *[힌트: 자명한 한 줄짜리 풀이가 있습니다!]*

♥ 1.

   (a) 처음 n절을 부르는 데 $\Theta(n^3)$ 시간이 걸리는 노래를 찾으십시오(또는 직접 지으십시오).

   (b) 처음 n개 절을 부르는 데 $\Theta(n \log n)$ 시간이 걸리는 노래를 찾으십시오(또는 직접 지으십시오).

   (c) 처음 n절을 부르는 데 또 다른 기묘한 양의 시간이 걸리는 노래를 찾아보십시오(또는 직접 지어 보십시오).

2. 주의 깊은 독자라면 "벽 위의 맥주 n병"이나 "크리스마스의 n일" 같은 노래에 대한 우리의 분석이 지나치게 단순하다고 지적할 수도 있습니다. 큰 수는 작은 수보다 부르는 데 더 오래 걸리기 때문입니다. 더 일반적으로 말하면, 주어진 길이의 단어는 그 수가 한정되어 있으므로 단어 집합이 클수록 필연적으로 더 긴 단어를 포함하게 됩니다.[24] 부른 *단어*의 수 대신 *음절*의 수를 세면 노래 부르는 시간을 더 정확하게 추정할 수 있습니다.

   (a) 정수 n을 노래로 부르는 데는 시간이 얼마나 걸립니까?

   (b) "벽 위의 맥주 n병"을 부르는 데는 시간이 얼마나 걸립니까?

   (c) "크리스마스의 n일"을 부르는 데 시간이 얼마나 걸립니까?

   평소와 같이, 어떤 함수 f에 대해 $O(f(n))$ 형태로 답을 표현하십시오.

3. 누적식 권주가 "The Barley Mow"는 수백 년 동안 영국 제도 전역에서 불려 왔습니다. 이 노래에는 여러 변형이 있습니다. 그림 0.6에는 데번과 콘월에서 전통적으로 불려 온 한 버전의 의사 가사가 실려 있으며, 여기서 *vessel*$[i]$는 맥주 2^i온스를 담는 용기의 이름입니다.[25]

[24] Ja, das ist das Subatomarteilchenbeschleunigungsnaturmäßigkeitsuntersuchungsmaschine!

[25] 실제로 이 노래에는 다음 용기 중 일부가 등장합니다. nipperkin, quarter-gill, half-a-gill, gill, quarter-pint, half-a-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel/kilderkin, barrel, hogshead, pipe/butt, tun, well, river, ocean입니다. (특히 끝부분의) 몇 가지 예외를 제외하면, 이 목록의 모든 용기는 바로 앞 용기의 두 배 용량입니다. 아일랜드와 스코틀랜드 버전은 가사가 약간 다르며, 대개

```
BarleyMow(n):
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  "We'll drink it out of the jolly brown bowl,"
  "Here's a health to the barley-mow!"
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  for i ← 1 to n
      "We'll drink it out of the vessel[i], boys,"
      "Here's a health to the barley-mow!"
      for j ← i downto 1
          "The vessel[j],"
      "And the jolly brown bowl!"
      "Here's a health to the barley-mow!"
      "Here's a health to the barley-mow, my brave boys,"
      "Here's a health to the barley-mow!"
```

그림 0.6. "보리 더미(The Barley Mow)".

(a) 각 이름 $vessel[i]$가 한 단어이고, 여러분이 1초에 네 단어를 노래할 수 있다고 가정합시다. BarleyMow(n)을 노래하는 데 시간이 얼마나 걸리겠습니까? (점근적으로 정확한 한계를 제시하십시오.)

(b) 이 노래를 임의로 큰 n 값에 대해 부르고 싶다면 그릇 이름을 직접 지어내야 합니다. 반복을 피하려면 n이 커질수록 이 이름들도 점점 길어져야 합니다. $vessel[n]$이 $\Theta(\log n)$개의 음절로 이루어져 있고, 1초에 여섯 음절을 부를 수 있다고 가정합시다. 이제 BarleyMow(n)을 부르는 데 시간이 얼마나 걸리겠습니까? (점근적으로 정확한 한계를 제시하십시오.)

(c) 그릇의 이름을 언급할 때마다 그에 해당하는 양의 맥주를 실제로 마신다고 가정합시다. 유쾌한 갈색 사발(jolly brown bowl)에는 1온스를, 각 $vessel[i]$에는 2^i온스를 마십니다. 이 문제에서는 여러분이 21세 이상이라고 가정할 때, BarleyMow(n)을 부른다면 맥주를 *정확히* 몇 온스 마시게 됩니까? (점근적 한계만 제시하지 말고 *정확한* 답을 제시하십시오.)

4. 헌팅턴-힐 알고리즘 ApportionCongress의 입력은 배열 $Pop[1..n]$과 정수 R이었음을 떠올려 봅시다. 여기서 $Pop[i]$는 i번째 주의 인구이고, R은 배정할 하원의원의 총수입니다. 출력은 배열 $Rep[1..n]$이며, 여기서 $Rep[i]$는 알고리즘이 i번째 주에 배정한 하원의원 수입니다.

"gallon" 다음부터는 사람(여급, 술집 주인, 짐마차꾼 등)으로 바뀝니다.

"Give us once a drink"라는 제목의 이 노래의 초기 버전은 1600년경 존 마스턴(John Marston)이 쓴 희곡 *Jack Drum's Entertainment (or the Comedie of Pasquill and Katherine)*에 등장합니다. ("Giue vs once a drinke for and the black bole. Sing gentle Butler *bally moy*!") 마스턴이 이 "high Dutch Song"을 이 희곡을 위해 따로 썼는지, "bally moy"가 "barley mow"를 잘못 알아들은 말(몬더그린)인지 아니면 그 반대인지, 혹은 애초에 같은 노래이기는 한지에 대해서는 의견이 엇갈립니다. 이런 논의는 맥주 n병을 앞에 두고 나누는 것이 가장 좋습니다.

헌팅턴-힐 알고리즘은 우선순위 큐를 전혀 사용하지 않는 방식으로 설명되기도 합니다. 최상위 알고리즘은 *제수*라고 부르는 양의 실수 D를 "추측"한 다음, 아래의 서브루틴을 실행하여 의석 배분을 계산합니다. 변수 q는 주어진 제수 D에 대해 한 주에 배정되는 대표자의 이상적인 *할당량*이며, 실제로 배정되는 대표자 수는 항상 $\lceil q \rceil$ 또는 $\lfloor q \rfloor$ 중 하나입니다.

```
HHGuess(Pop[1 .. n], R, D):
  reps ← 0
  for i ← 1 to n
      q ← Pop[i]/D
      if q · q < ⌈q⌉ · ⌊q⌋
          Rep[i] ← ⌊q⌋
      else
          Rep[i] ← ⌈q⌉
      reps ← reps + Rep[i]
  return reps
```

최종 반환값 *reps*에는 세 가지 가능성이 있습니다. $reps < R$이면 의원을 충분히 배정하지 못한 것이며, 이는 (적어도 직관적으로는) 제수 D가 너무 작았음을 의미합니다. $reps > R$이면 의원을 너무 많이 배정한 것이며, 이는 (적어도 직관적으로는) 제수 D가 너무 컸음을 의미합니다. 마지막으로 $reps = R$이면 배열 $Rep[1 .. n]$을 최종 의석 배분으로 반환할 수 있습니다. 실제로는 *표준 제수* $D = P/R$에 가까운 소수의 정수 제수로 HHGuess를 호출하여 ($reps = R$인) 올바른 의석 배분을 계산할 수 있습니다.

다음 문제들에서 $P = \sum_{i=1}^{n} Pop[i]$가 n개 주 전체의 총인구를 나타낸다고 하고, $n \le R \le P$라고 가정합니다.

(a) 표준 제수 $D = P/R$로 HHGuess를 호출해도 올바른 의석 배분이 *반드시* 얻어지는 것은 아님을 보이십시오.

(b) HHGuess가 서로 다른 두 제수 D와 D'에 대해 같은 *reps* 값을 반환한다면, 두 제수 모두에 대해 같은 할당 $Rep[1 .. n]$도 계산한다는 것을 증명하십시오.

(c) HHGuess가 올바른 값 R을 반환하면, 앞서 다룬 알고리즘 Apportion-Congress와 동일한 배분 $Rep[1 .. n]$을 계산함을 증명하십시오.

(d) "올바른" 제수 D가 반드시 존재하지는 않음을 증명하십시오! 즉, 모든 실수 $D > 0$에 대하여 HHGuess가 배정하는 대표자 수가 R과 같지 않도록 하는 입력 $Pop[1 .. n]$과 R(단, $n \le R \le P$)을 제시하십시오. *[힌트: HHGuess의 넷째 줄에서 <를 ≤로 바꾸면 어떻게 됩니까?]*

대군을 다스리는 것도 소수의 병사를 다스리는 것과 원리가 같습니다. 그저 그 수를 나누는 문제일 뿐입니다.

— 손자, *손자병법* (c. 400CE), Lionel Giles 영역 (1910)

우리의 삶은 사소한 일들로 허비되고 있습니다.… 간소화하십시오, 간소화하십시오.

— 헨리 데이비드 소로, *월든* (1854)

자, 붐(Voom)이 무엇인지는 묻지 마십시오. 저도 영영 모를 것입니다.
하지만, 세상에! 이것만은 말씀드리겠습니다. 눈 하나는 정말로 싹 치워 줍니다!

— Dr. Seuss [Theodor Seuss Geisel], *모자 쓴 고양이가 돌아왔다* (1958)

어려운 일부터 먼저 하십시오. 쉬운 일은 저절로 해결될 것입니다.

— 데일 카네기의 말로 전해짐

1

재귀

1.1 환원

*환원*은 알고리즘 설계에서 가장 흔히 쓰이는 기법입니다. 어떤 문제 X를 다른 문제 Y로 환원한다는 것은 Y에 대한 알고리즘을 블랙박스나 서브루틴으로 사용하는 X에 대한 알고리즘을 작성한다는 뜻입니다. 결정적으로, 그렇게 얻은 X에 대한 알고리즘의 정확성은 Y에 대한 알고리즘이 *어떻게* 동작하는지에 어떤 식으로든 의존해서는 안 됩니다. 우리가 가정할 수 있는 것은 블랙박스가 Y를 올바르게 푼다는 사실뿐입니다. 블랙박스의 내부 동작은 그야말로 *우리가 알 바가 아니며*, 다른 누군가의 문제입니다. 블랙박스가 순전히 마법으로 동작한다고 말 그대로 생각하는 편이 가장 좋을 때가 많습니다.

예를 들어 앞 장에서 설명한 농부 곱셈 알고리즘은 임의의 두 양의 정수를 곱하는 문제를 더 단순한 세 가지 문제인 덧셈, 반분(절반으로 나누기), 홀짝 판별로 환원합니다. 이 알고리즘은 이 세 가지 연산을 지원하는 추상적인 "양의 정수" 자료형에 의존하지만, 곱셈 알고리즘의 정확성은 구체적인 데이터 표현(탤리 마크, 점토 토큰, 바빌로니아 60진법, 키푸, 산가지, 로마 숫자, 손가락 위치, 오그림 돌, 고바르 숫자, 2진법, 음의 2진법, 그레이 코드, 균형 3진법, 황금비 진법, 4허수 진법, . . .)에도, 그 연산들의 구체적인 구현에도 의존하지 않습니다. 물론 곱셈 알고리즘의 *실행 시간*은 덧셈, 반분, 홀짝 판별 연산의 *실행 시간*에 의존하지만, 이는 *정확성*과는 별개의 문제입니다. 무엇보다 중요한 점은 더 효율적인 수 표현으로 바꾸기만 해도(예를 들어 탤리 마크에서 자릿값 표기법으로) 더 효율적인 곱셈 알고리즘을 만들 수 있다는 것입니다.

마찬가지로, 헌팅턴-힐 알고리즘은 의회 의석 배분 문제를 INSERT와 EXTRACTMAX 연산을 지원하는 우선순위 큐를 유지하는 문제로 환원합니다. 추상 자료형 "우선순위 큐"는 블랙박스이며, 의석 배분 알고리즘의 정확성은 특정 우선순위 큐 자료구조에 의존하지 않습니다. 물론 의석 배분 알고리즘의 *실행 시간*은 INSERT와 EXTRACTMAX 알고리즘의 *실행 시간*에 좌우되지만, 이는 알고리즘의 *정확성*과는 별개의 문제입니다. 환원의 아름다움은 새로운 우선순위 큐 자료구조로 교체하기만 하면 더 효율적인 의석 배분 알고리즘을 만들 수 있다는 데 있습니다. 게다가 그 자료구조를 설계하는 사람은 그것이 의회 의석 배분에 쓰일 것이라는 사실을 알 필요도, 신경 쓸 필요도 없습니다.

알고리즘을 설계할 때 우리는 사용하는 기본 구성 요소가 정확히 어떻게 구현되어 있는지, 또는 우리의 알고리즘이 훨씬 더 큰 문제를 해결하기 위한 구성 요소로 어떻게 사용될지 알지 못할 수도 있습니다. 이러한 무지는 많은 초보자에게 불편하게 느껴지지만, 피할 수 없는 동시에 매우 유용합니다. 구성 요소가 어떻게 작동하는지 정확히 알고 있을 때조차도, 모르는 척하는 것이 *매우* 도움이 되는 경우가 많습니다.

1.2 단순화하고 위임하기

*재귀*는 특히 강력한 형태의 환원으로, 대략 다음과 같이 설명할 수 있습니다:

- 주어진 문제의 인스턴스를 직접 풀 수 있다면 직접 푸십시오.
- 그렇지 않다면, 그 문제를 하나 이상의 ***같은 문제의 더 간단한 인스턴스***로 환원하십시오.

자기 참조가 헷갈린다면, 다른 종류의 환원에서 가정하듯이 더 간단한 문제들은 다른 누군가가 풀어 줄 것이라고 상상하면 도움이 될 수 있습니다. 저는 그 누군가를 ***재귀 요정***이라고 부르기를 좋아합니다. 여러분이 할 *유일한* 일은 원래 문제를 *단순화*하거나, 단순화가 불필요하거나 불가능할 때 그 문제를 직접 푸는 것입니다. 더 간단한 부분 문제는 모두 재귀 요정이 여러분 대신 풀어 줍니다. 그 방법은 여러분이 알 바가 아니니 *참견하지 마십시오*.[1] 수학에 조예가 깊은 독자라면 재귀 요정을 더 공식적인 이름인 ***귀납법 가정***으로 알아볼 것입니다.

어떤 재귀적 방법이든 올바르게 동작하려면 반드시 만족해야 하는 가벼운 기술적 조건이 하나 있습니다. 점점 더 단순한 인스턴스로 향하는 환원이 무한히 이어져서는 안 됩니다. 재귀적 환원은 결국 다른 방법으로 풀 수 있는 기초적인 ***기저 사례***에 도달해야 합니다. 그렇지 않으면 재귀 알고리즘은 영원히 반복됩니다. 이 조건을 만족시키는 가장 흔한 방법은 같은 문제의 ***더 작은*** 인스턴스 하나 이상으로 환원하는 것입니다. 예를 들어 원래 입력이 글럽 n개를 가진 스크리블이라면, 각 재귀 호출의 입력은 글럽이 n개보다 엄격히 적은 스크리블이어야 합니다. 물론 스크리블에 글럽이 하나도 없다면 이는 불가능합니다—글럽이 음수 개일 수는 없습니다. 그건 말도 안

[1] 제가 학부생이었을 때에는 재귀를 재귀 요정 대신 "엘프"의 공으로 돌렸습니다. 이는 늙은 구두장이가 일을 다 끝내지 못한 채 잠자리에 들었는데, 깨어나 보니 엘프들("Wichtelmänner")이 밤사이에 모든 일을 끝내 놓았더라는 그림 형제의 이야기에서 따온 것입니다. 저보다 엔테오젠 경험이 더 풍부한 사람이라면 이 Rekursionswichtelmänner가 테렌스 매케나가 말한 "스스로 변신하는 기계 엘프"임을 알아볼지도 모릅니다.

되는 일입니다!—따라서 그런 경우에는 다른 방법을 사용하여 스크리블을 그린들블로 프해야 합니다.

우리는 이 패턴의 한 가지 사례를 농부 곱셈 알고리즘에서 이미 보았습니다. 이 알고리즘은 다음의 재귀적 항등식에 직접 기반합니다.

$$x \cdot y = \begin{cases} 0 & \text{만약} x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{만약} x \text{짝수입니다} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{만약} x \text{홀수입니다} \end{cases}$$

같은 점화식을 알고리즘으로 나타내면 다음과 같습니다.

```
PeasantMultiply(x, y):
  if x = 0
      return 0
  else
      x' ← ⌊x/2⌋
      y' ← y + y
      prod ← PeasantMultiply(x', y')    《재귀하십시오!》
      if x is odd
          prod ← prod + y
      return prod
```

게으른 이집트 서기관은 x'과 y'을 계산하고, *x'과 y'을 곱해 달라고 더 하급 서기관에게 부탁한* 다음, 필요하면 하급 서기관의 답에 y를 더하는 방식으로 이 알고리즘을 실행할 수 있습니다. $x' < x$이므로 하급 서기관의 문제는 더 단순하며, 양의 정수를 계속 감소시키면 결국 0에 도달합니다. 하급 서기관이 실제로 $x' \cdot y'$을 어떻게 계산하는지는 상급 서기관이 알 바가 아닙니다(그리고 여러분이 알 바도 아닙니다).

1.3 하노이의 탑

하노이의 탑 퍼즐은 프랑스의 교사이자 오락 수학자인 에두아르 뤼카(Édouard Lucas)가 1883년에[2] "N. Claus (de Siam)"("Lucas d'Amiens"의 애너그램)이라는 필명으로 처음 발표했습니다. 그것도 실제로 손에 잡히는 퍼즐로 말입니다! 이듬해 앙리 드 파르빌(Henri de Parville)은 다음과 같은 놀라운 이야기와 함께 이 퍼즐을 소개했습니다:[3]

> 베나레스의 대사원[4] …세계의 중심을 표시하는 둥근 지붕 아래에는 놋쇠 판이 하나 놓여 있고, 그 판에는 높이가 각각 1큐빗이고 굵기가 벌의 몸통만 한 다이아몬드

[2] 뤼카는 훗날 자신이 1876년에 이 퍼즐을 발명했다고 주장했습니다.

[3] 이 영어 번역은 W. W. 라우스 볼(W. W. Rouse Ball)의 1892년 저서 *Mathematical Recreations and Essays*에서 가져온 것입니다.

[4] "베나레스의 대사원"은 거의 틀림없이 인도 우타르프라데시주 바라나시에 있는 카시 비슈바나트 사원이며, 이곳은 가상의 인물 N. Claus가 살았다고 하는 베트남 하노이(Hà Nội)에서 서북서쪽으로 약 2400km 떨어져 있습니다. 공교롭게도 프랑스군은 뤼카가 이 퍼즐을 발표한 바로 그해인 1883년에 하노이를 침공하였고, 이는 결국 하노이가 프랑스령 인도차이나의 수도가 되는 결과로 이어졌습니다.

> 바늘 세 개가 꽂혀 있습니다. 천지 창조 때 신은 이 바늘 중 하나에 순금 원반 예순네 개를 끼워 두었는데, 가장 큰 원반이 놋쇠 판 위에 놓여 있고 나머지 원반은 위로 갈수록 점점 작아져 맨 위의 원반에 이릅니다. 이것이 브라마의 탑입니다. 사제들은 밤낮으로 쉬지 않고 브라마의 고정불변한 법칙에 따라 원반을 한 다이아몬드 바늘에서 다른 바늘로 옮기는데, 이 법칙에 따르면 당번 사제는 한 번에 원반을 두 개 이상 옮겨서는 안 되며, 원반을 바늘에 놓을 때는 그 아래에 더 작은 원반이 없도록 놓아야 합니다. 이렇게 하여 예순네 개의 원반이 천지 창조 때 신이 놓아 둔 바늘에서 다른 바늘 중 하나로 모두 옮겨지면, 탑도 사원도 브라만도 모두 무너져 먼지가 되고, 천둥소리와 함께 세계는 사라질 것입니다.

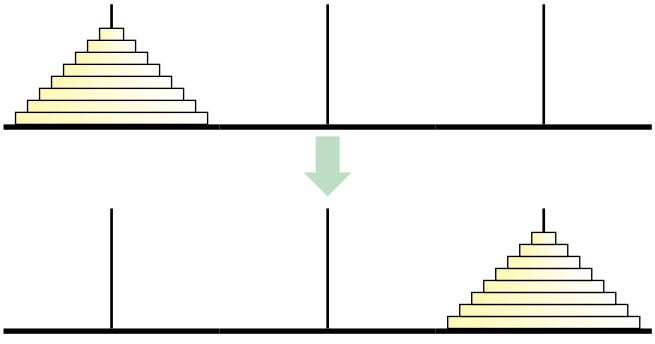


그림 1.1. (원판 8개짜리) 하노이의 탑 퍼즐

물론 훌륭한 컴퓨터 과학자라면 이 이야기를 읽자마자 고정된 상수 64를 변수 n으로 바꾸고 싶어질 것입니다. 또한 실제 퍼즐은 대부분 다이아몬드와 금이 아니라 나무로 만들어지므로, 원반을 놓을 수 있는 세 위치를 "바늘" 대신 "기둥"이라고 부르겠습니다. 남는 세 번째 기둥을 이따금 임시 자리로 사용하되 원반을 자신보다 작은 원반 위에는 한 번도 올려놓지 않으면서, n개의 원반으로 이루어진 탑을 한 기둥에서 다른 기둥으로 어떻게 옮길 수 있겠습니까?

N. Claus (de Siam)가 자신의 퍼즐에 동봉한 소책자에서 지적했듯이, 이 퍼즐을 푸는 비결은 재귀적으로 생각하는 것입니다. 퍼즐 전체를 한꺼번에 풀려고 하는 대신, 가장 큰 원반 하나를 옮기는 일에만 집중해 봅시다. 처음에는 다른 원반들이 모두 가로막고 있으므로 이 원반을 옮길 수 없습니다. 따라서 먼저 그보다 작은 원반 $n-1$개를 여분의 기둥으로 옮겨야 합니다. 그렇게 하고 나면 가장 큰 원반을 목적지로 곧바로 옮길 수 있습니다. 마지막으로 퍼즐을 마무리하려면 작은 원반 $n-1$개를 여분의 기둥에서 목적지로 옮겨야 합니다.

이제 우리가 알아내야 할 것은 어떻게—

안 됩니다!! 멈추십시오!!

이것으로 끝입니다! 다 되었습니다! 우리는 n개 원반 하노이의 탑 문제를 $(n-1)$개 원반 하노이의 탑 문제 두 개로 성공적으로 환원했으며, 이 두 문제는 기쁜 마음으로 재귀 요정에게—루카스의 비유를 더 이어 가자면 사원의 젊은 수도승들에게—넘기면

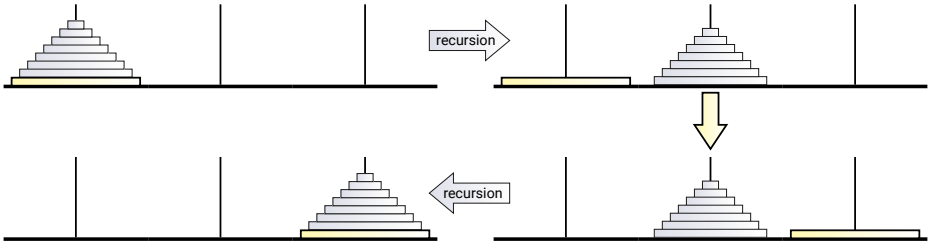


그림 1.2. 하노이의 탑 알고리즘입니다. 맨 아래 원판을 제외한 나머지는 모두 무시하십시오.

됩니다. *우리의* 일은 끝났습니다. 젊은 수도승들을 믿지 못했다면 애초에 그들을 고용하지 않았을 것입니다. 그들이 방해받지 않고 자기 일을 하도록 맡겨 둡시다.

우리의 환원은 미묘하지만 매우 중요한 가정 하나를 전제로 합니다. 바로 *가장 큰 원반이 존재한다*는 가정입니다. 우리의 재귀 알고리즘은 원반의 개수가 양수이기만 하면 언제나 동작하지만, $n = 0$일 때는 동작하지 않습니다. 이 경우는 다른 방법으로 처리해야 합니다. 다행히 베나레스의 수도승들은 훌륭한 불교도답게, 아무것도 하지 않음으로써 원반 0개를 한 기둥에서 다른 기둥으로 순식간에 옮기는 데 아주 능숙합니다.

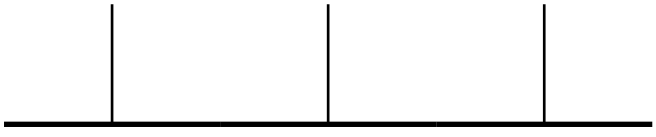


그림 1.3. 하노이의 탑 알고리즘의 공허한 기저 사례입니다. 숟가락은 없습니다.

그 모든 작은 원판들이 어떻게 움직이는지, 더 일반적으로는 재귀를 펼쳤을 때 무슨 일이 일어나는지 생각해 보고 싶은 유혹이 들 수도 있습니다. 하지만 정말로, 그렇게 하지 마십시오. 대부분의 재귀 알고리즘에서 재귀를 펼쳐 보는 것은 필요하지도 않고 도움이 되지도 않습니다. 우리의 ***유일한*** 임무는 주어진 문제 인스턴스를 하나 이상의 더 간단한 인스턴스로 환원하거나, 그러한 환원이 불가능하다면 문제를 직접 푸는 것입니다. 우리의 재귀적 하노이의 탑 알고리즘은 $n = 0$일 때 자명하게 올바릅니다. 임의의 $n \geq 1$에 대해 재귀 요정이 위쪽 $n-1$개의 원판을 올바르게 옮겨 주므로(더 형식적으로 말하면, 귀납 가정에 의해 우리의 재귀 알고리즘이 위쪽 $n-1$개의 원판을 올바르게 옮기므로) 우리의 알고리즘은 올바릅니다.

재귀적 하노이 알고리즘을 의사코드로 나타내면 그림 1.4와 같습니다. 이 알고리즘은 세 번째 임시 기둥(*tmp*)을 임시 보관 장소로 사용하여 n개의 원판 더미를 출발 기둥(*src*)에서 도착 기둥(*dst*)으로 옮깁니다. $n = 0$일 때 이 알고리즘이 올바르게도 아무 일도 하지 않는다는 점에 주목하십시오.

n개의 원반을 옮기는 데 필요한 이동 횟수, 즉 우리 알고리즘의 실행 시간을 $T(n)$이라고 합시다. 아무것도 하지 않는 기저 사례에 따라 $T(0) = 0$이고, 더 일반적인 재귀 알고리즘에 따라 모든 $n \geq 1$에 대해 $T(n) = 2T(n-1)+1$입니다. $T(n)$의 처음

```
HANOI(n, src, dst, tmp):
  if n > 0
      HANOI(n − 1, src, tmp, dst)     《재귀하십시오!》
      move disk n from src to dst
      HANOI(n − 1, tmp, dst, src)     《재귀하십시오!》
```

그림 1.4. 하노이의 탑을 푸는 재귀 알고리즘

몇 개 값을 적어 보면 $T(n) = 2^n - 1$이라고 쉽게 추측할 수 있으며, 간단한 귀납법 증명으로 이 추측이 옳다는 것을 보일 수 있습니다. 특히 64개의 원반으로 이루어진 탑을 옮기려면 $2^{64} - 1 = 18{,}446{,}744{,}073{,}709{,}551{,}615$번의 개별 이동이 필요합니다. 따라서 1초에 한 번씩 옮기는 놀라운 속도로 작업하더라도, 베나레스의 수도승들은 약 5,850억 년("plus de *cinq milliards de siècles*") 동안 일해야 하며, 그제야 탑과 사원과 브라만들이 모두 무너져 먼지가 되고 천둥소리와 함께 세상이 사라질 것입니다.

1.4 병합 정렬

병합 정렬은 범용 프로그램 내장식 컴퓨터를 위해 설계된 가장 초기의 알고리즘 중 하나입니다. 이 알고리즘은 1945년 존 폰 노이만이 개발했습니다. 1947년에는 폰 노이만이 허먼 골드스타인과 함께 발표한 논문에서 EDVAC을 위한 최초의 비수치 프로그램 중 하나로 자세히 기술되었습니다.[5]

1. 입력 배열을 크기가 대략 같은 두 부분 배열로 나눕니다.
2. 각 부분 배열을 재귀적으로 병합 정렬합니다.
3. 새로 정렬된 부분 배열들을 하나의 정렬된 배열로 병합합니다.

입력:	S O R T I N G E X A M P L
분할:	S O R T I N | G E X A M P L
왼쪽 재귀 호출:	I N O R S T | G E X A M P L
올바르게 재귀하십시오:	I N O R S T | A E G L M P X
병합:	A E G I L M N O P R S T X

그림 1.5. 병합 정렬 예시입니다.

첫 번째 단계는 완전히 자명합니다. 배열의 크기를 2로 나누기만 하면 됩니다. 두 번째 단계는 재귀 요정에게 맡길 수 있습니다. 실제 작업은 모두 마지막 병합 단계에서

[5]사실 골드스타인과 폰 노이만이 기술한 것은 오늘날 흔히 상향식 병합 정렬이라고 부르는 비재귀 변형입니다. 당시에는 대규모 데이터 집합을 특수 목적 기계로 정렬했습니다. 이 기계는 거의 모두 IBM이 만든 것이었으며, 이진 기수 정렬의 여러 변형을 사용해 천공 카드를 다루었습니다. 폰 노이만은 EDVAC이 "사람의 개입이나 추가 장비 없이도" IBM의 전용 정렬기보다 더 빠르게 정렬할 수 있으므로 EDVAC은 "범용" 기계이며 특수 목적 정렬 기계는 더 이상 필요하지 않다고 주장했습니다(그리고 이 주장은 성공을 거두었습니다!).

이루어집니다. 알고리즘의 전체 설명은 그림 1.6에 나와 있습니다. 재귀 구조를 분명하게 드러내기 위해 병합 단계를 독립된 서브루틴으로 분리했습니다. 병합 알고리즘도 재귀적입니다. 출력 배열의 첫 번째 원소를 찾은 다음, 입력 배열의 나머지 부분을 재귀적으로 병합합니다.

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])
  《재귀하십시오!》
      MERGESORT(A[m+1..n])
  《재귀하십시오!》
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1; j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i]; i ← i+1
      else if i > m
          B[k] ← A[j]; j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i]; i ← i+1
      else
          B[k] ← A[j]; j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```

그림 1.6. 병합 정렬

정확성

이 알고리즘이 올바름을 증명하기 위해 우리의 오랜 친구인 귀납법을 두 번 적용합니다. 먼저 MERGE 서브루틴에 적용하고, 그다음 최상위 MERGESORT 알고리즘, 즉 병합 정렬 알고리즘에 적용합니다.

보조정리 1.1. *입력에서 부분 배열 $A[1..m]$과 $A[m+1..n]$이 정렬되어 있다고 가정하면, MERGE는 두 부분 배열을 올바르게 병합합니다.*

증명: $A[1..n]$을 임의의 배열이라 하고, m을 부분 배열 $A[1..m]$과 $A[m+1..n]$이 정렬되어 있도록 하는 임의의 정수라고 합시다. 0부터 n까지의 모든 k에 대해, 주 루프의 마지막 $n-k-1$번의 반복이 $A[i..m]$과 $A[j..n]$을 $B[k..n]$으로 올바르게 병합함을 증명합니다. 증명은 병합해야 할 남은 원소의 개수인 $n-k+1$에 대한 귀납법으로 진행합니다.

$k > n$이면, 알고리즘은 아무것도 하지 않음으로써 두 개의 빈 부분 배열을 올바르게 병합합니다. (이것이 귀납적 증명의 기저 사례입니다.) 그렇지 않으면, 주 반복문의 k번째 반복에 대해 고려해야 할 경우가 네 가지 있습니다.

- $j > n$이면 부분 배열 $A[j..n]$이 비어 있으므로 $\min(A[i..m] \cup A[j..n]) = A[i]$입니다.
- $i > m$이면 부분 배열 $A[i..m]$이 비어 있으므로 $\min(A[i..m] \cup A[j..n]) = A[j]$입니다.
- 그렇지 않고 $A[i] < A[j]$이면, $\min(A[i..m] \cup A[j..n]) = A[i]$입니다.
- 그렇지 않으면 $A[i] \geq A[j]$일 수밖에 없으며, $\min(A[i..m] \cup A[j..n]) = A[j]$입니다.

네 경우 모두 $B[k]$에는 $A[i..m] \cup A[j..n]$의 가장 작은 원소가 올바르게 할당됩니다. $B[k] \leftarrow A[i]$로 할당하는 두 경우에는 재귀 요정이 올바르게 병합해 줍니다—죄송합니다, 제 말은 귀납법 가정에 따라 주 반복문의 마지막 $n-k$번의 반복이 $A[i+1..m]$과 $A[j..n]$을 $B[k+1..n]$으로 올바르게 병합한다는 뜻입니다. 마찬가지로 나머지 두 경우에도 재귀 요정이 나머지 부분 배열들을 올바르게 병합합니다. □

정리 1.2. *병합 정렬 알고리즘 MergeSort는 모든 입력 배열 $A[1..n]$을 올바르게 정렬합니다.*

증명: n에 대한 귀납법으로 정리를 증명합니다. $n \le 1$이면 알고리즘은 올바르게 아무것도 하지 않습니다. 그렇지 않으면 재귀 요정이 올바르게 정렬합니다—죄송합니다, 다시 말하자면 귀납법 가정에 의해 우리 알고리즘은 더 작은 두 부분 배열 $A[1..m]$과 $A[m+1..n]$을 올바르게 정렬하며, 그 후 두 부분 배열은 (보조정리 1.1에 의해) 하나의 정렬된 배열로 올바르게 Merge됩니다. □

분석

병합 정렬 알고리즘인 MergeSort는 재귀적이므로, 그 실행 시간은 자연스럽게 점화식으로 표현됩니다. Merge는 반복마다 상수 시간의 작업을 수행하는 단순한 for 루프이므로 분명히 $O(n)$ 시간이 걸립니다. 따라서 MergeSort에 대한 다음 점화식을 곧바로 얻습니다.

$$T(n) = T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + O(n).$$

대부분의 분할 정복 점화식에서와 마찬가지로, (이 장의 뒷부분에서 설명할 *정의역 변환*이라는 기법을 사용하면) 바닥 함수와 천장 함수를 안전하게 제거할 수 있으며, 그러면 더 간단한 점화식 $T(n) = 2T(n/2) + O(n)$을 얻습니다. 재귀 트리 방법(역시 이 장의 뒷부분에서 설명합니다)의 "모든 레벨이 같은" 경우로부터 닫힌 형태의 해 $\boldsymbol{T(n) = O(n \log n)}$이 곧바로 도출됩니다. 재귀 트리에 (아직) 익숙하지 않더라도 해 $T(n) = O(n \log n)$은 귀납법으로 검증할 수 있습니다.

1.5 퀵정렬

퀵정렬은 또 다른 재귀 정렬 알고리즘으로, 토니 호어(Tony Hoare)가 1959년에 발견하여 1961년에 처음 발표했습니다. 이 알고리즘에서 어려운 작업은 재귀 *이전에* 배열을 더 작은 부분 배열로 나누는 일이며, 그 덕분에 정렬된 부분 배열을 병합하는 일은 자명해집니다.

1. 배열에서 *피벗* 원소를 하나 선택하십시오.
2. 배열을 피벗보다 작은 원소들, 피벗 원소 자체, 피벗보다 큰 원소들을 각각 담는 세 개의 부분 배열로 분할하십시오.
3. 첫 번째와 마지막 부분 배열을 재귀적으로 퀵정렬합니다.

입력:	S O R T I N G E X A M P L
피벗을 선택하십시오:	S O R T I N G E X A M [P] L
분할:	A G O E I N L M [P] T X S R
왼쪽 재귀 호출:	A E G I L M N O [P] T X S R
올바르게 재귀하십시오:	A E G I L M N O [P] R S T X

그림 1.7. 퀵정렬 예시입니다.

더 자세한 의사코드는 그림 1.8에 나와 있습니다. PARTITION 서브루틴에서 입력 매개변수 p는 정렬되지 않은 배열에서 피벗 원소가 있는 인덱스이며, 이 서브루틴은 배열을 분할한 뒤 피벗 원소의 새 인덱스를 반환합니다. 효율적인 분할 알고리즘은 여러 가지가 있으며, 여기서 소개하는 알고리즘은 니코 로무토(Nico Lomuto)가 고안한 것으로 알려져 있습니다.[6] 변수 ℓ은 배열에서 피벗 원소보다 작은(ℓess) 항목의 개수를 셉니다.

```
QUICKSORT(A[1..n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← PARTITION(A, p)
      QUICKSORT(A[1..r − 1])
  《재귀하십시오!》
      QUICKSORT(A[r + 1..n])
  《재귀하십시오!》
```

```
PARTITION(A[1..n], p):
  swap A[p] ↔ A[n]
  ℓ ← 0        《#항목 < 피벗》
  for i ← 1 to n − 1
      if A[i] < A[n]
          ℓ ← ℓ + 1
          swap A[ℓ] ↔ A[i]
  swap A[n] ↔ A[ℓ + 1]
  return ℓ + 1
```

그림 1.8. 퀵정렬

정확성

병합 정렬과 마찬가지로, 퀵정렬 QUICKSORT가 올바름을 증명하려면 두 개의 별도 귀납법 증명이 필요합니다. 하나는 PARTITION이 배열을 올바르게 분할한다는 것을 증명하는 것이고, 다른 하나는 PARTITION이 올바르다고 *가정할 때* QUICKSORT가 올바르게 정렬한다는 것을 증명하는 것입니다. PARTITION이 올바름을 증명하려면 다음 루프 불변식을 증명해야 합니다. 메인 루프의 각 반복이 끝날 때, 부분 배열 $A[1..\ell]$의 모든 원소는 $A[n]$보다 작고(ℓess), 부분 배열 $A[\ell+1..i]$의 어떤 원소도 $A[n]$보다 작지 않습니다. 나머지 간단하지만 지루한 세부 사항은 독자의 연습 문제로 남겨 두겠습니다.

[6] 호어(Hoare)는 이보다 복잡한 "양방향" 분할 알고리즘을 제안했으며, 이 알고리즘은 실용적인 면에서 로무토의 알고리즘보다 나은 점이 몇 가지 있습니다. 반면 호어의 분할 알고리즘은 하나 차이 오류(off-by-one error)가 모여들어 무덤을 이루는 곳 중 하나이기도 합니다.

분석

퀵정렬의 분석 역시 병합 정렬의 분석과 비슷합니다. Partition은 반복마다 상수 시간의 작업만 수행하는 단순한 for 루프이므로 분명히 $O(n)$ 시간에 실행됩니다. QuickSort의 경우, 선택한 피벗 원소의 *순위*인 r에 따라 달라지는 점화식을 얻습니다:

$$T(n) = T(r-1) + T(n-r) + O(n)$$

어떻게든 항상 마법처럼 배열 A의 *중앙값* 원소를 피벗으로 고를 수 있다면 $r = \lceil n/2 \rceil$이 되고, 두 부분 문제의 크기는 최대한 서로 비슷해지며, 점화식은 다음과 같이 됩니다

$$T(n) \;=\; T\big(\lceil n/2 \rceil - 1\big) + T\big(\lfloor n/2 \rfloor\big) + O(n) \;\le\; 2T(n/2) + O(n),$$

그러면 재귀 트리 방법을 쓰든, 그보다 더 간단한 "아, 맞다, 그 점화식은 병합 정렬에서 이미 풀었지" 방법을 쓰든 $T(n) = O(n \log n)$을 얻습니다.

사실 이 장의 뒷부분에서 살펴보겠지만, 정렬되지 않은 배열에서 중앙값 원소를 선형 시간에 찾는 것은 실제로 *가능합니다*. 그러나 그 알고리즘은 상당히 복잡하고, $O(\cdot)$ 표기법에 숨겨진 상수가 매우 커서 이를 이용한 정렬 알고리즘은 실용적이지 않습니다. 실제로 대부분의 프로그래머는 배열의 첫 번째 원소나 마지막 원소를 고르는 것처럼 간단한 방법으로 만족합니다. 이 경우 r은 1과 n 사이의 어떤 값이든 가질 수 있으므로, 다음을 얻습니다

$$T(n) = \max_{1 \le r \le n} \big(T(r-1) + T(n-r) + O(n)\big).$$

최악의 경우 두 부분 문제는 완전히 불균형해지며($r = 1$ 또는 $r = n$), 점화식은 $T(n) \le T(n-1) + O(n)$이 됩니다. 이 점화식의 해는 $\boldsymbol{T(n) = O(n^2)}$입니다.

흔히 쓰이는 또 다른 휴리스틱으로 "세 값의 중앙값(median of three)"이 있습니다. 세 원소를 고르고(보통 배열의 처음, 중간, 끝에서 고릅니다) 그 세 원소의 중앙값을 피벗으로 삼는 방법입니다. 이 휴리스틱은 원소 하나만 고르는 것보다 실제로 다소 더 효율적이며, 특히 배열이 이미 (거의) 정렬되어 있을 때 그렇습니다. 그러나 최악의 경우에는 여전히 $r = 2$ 또는 $r = n - 1$이 될 수 있습니다. 세 값의 중앙값 휴리스틱을 사용하면 점화식은 $T(n) \le T(1) + T(n-2) + O(n)$이 되며, 그 해는 여전히 $T(n) = O(n^2)$입니다.

직관적으로 보면 피벗 원소는 "대개" 배열의 중간 어딘가, 예를 들어 순위가 $n/10$과 $9n/10$ 사이인 곳에 위치해야 합니다. 이 관찰은 "평균 경우" 실행 시간이 $O(n \log n)$이어야 함을 시사합니다. 이러한 직관을 형식화할 수는 있지만, 가장 흔한 형식화는 입력 배열의 모든 순열이 동일한 확률로 나타난다는 전혀 비현실적인 가정을 합니다. 실제 세계의 데이터는 무작위일 수도 있지만, 우리가 미리 예측할 수 있는 방식으로 무작위인 것은 아니며, *분명히* 균등하지도 않습니다![7]

때때로 사람들은 어떤 이유에서인지 "최선의 경우" 실행 시간을 고려하기도 합니다. 우리는 그렇게 하지 않겠습니다.

[7] 반면에 피벗 인덱스 p를 균등하게 무작위로 선택하면, QuickSort는 가능한 모든 입력 배열에 대해 높은 *확률로* $O(n \log n)$ 시간에 실행됩니다. 핵심적인 차이는 무작위성을 제어하는 주체가 우리의 알고리즘이지, 우리의 코드를 읽은 뒤에 입력 데이터를 건네주는 전지전능한 악의적 적대자가 아니라는 점입니다. 안타깝게도 무작위 퀵정렬의 분석은 이 책의 범위를 벗어나지만, 관련 강의 노트는 http://algorithms.wtf/에서 찾을 수 있습니다.

1.6 패턴

병합 정렬과 퀵정렬은 모두 ***분할 정복***이라는 일반적인 세 단계 패턴을 따릅니다:

1. 주어진 문제 인스턴스를 *정확히* 같은 문제의 *서로 독립적이고 더 작은* 인스턴스 여러 개로 **분할**합니다.
2. 더 작은 인스턴스 각각을 재귀 요정에게 **위임**합니다.
3. 더 작은 인스턴스들의 해를 **결합**하여 주어진 인스턴스의 최종 해를 구합니다.

어떤 인스턴스의 크기가 특정 상수 임계값보다 작아지면, 재귀를 중단하고 무차별 대입으로 문제를 상수 시간에 직접 해결합니다.

분할 정복 알고리즘이 올바르다는 것을 증명하려면 거의 항상 귀납법이 필요합니다. 실행 시간을 분석하려면 점화식을 세우고 풀어야 하는데, 이 점화식은 보통 (안타깝게도 항상 그렇지는 않지만!) 재귀 트리를 사용하여 풀 수 있습니다.

1.7 재귀 트리

그렇다면 제가 계속 이야기하는 이 "재귀 트리"란 무엇일까요? 재귀 트리는 분할 정복 점화식을 풀기 위한 단순하고 범용적이며 그림을 활용하는 도구입니다. 재귀 트리는 각 재귀적 부분 문제마다 노드가 하나씩 있는 루트 트리입니다. 각 노드의 *값*은 대응하는 부분 문제에서 재귀 호출을 *제외하고* 소요된 시간입니다. 따라서 알고리즘의 전체 실행 시간은 트리에 있는 모든 노드의 값을 합한 것입니다.

이 아이디어를 더 구체적으로 설명하기 위해, 재귀가 아닌 작업에 $O(f(n))$ 시간을 쓴 다음 각각 크기가 n/c인 문제에 대해 r번의 재귀 호출을 수행하는 분할 정복 알고리즘을 생각해 봅시다. 상수 배수를 무시하면(상수 배수는 $O()$ 표기법 안에 숨길 수 있습니다), 이 알고리즘의 실행 시간은 다음 점화식으로 결정됩니다

$$T(n) = r\,T(n/c) + f(n).$$

$T(n)$에 대한 재귀 트리의 루트는 값 $f(n)$과 r개의 자식을 가지며, 각 자식은 (재귀적으로 정의된) $T(n/c)$에 대한 재귀 트리의 루트입니다. 이와 동등하게, 재귀 트리는 깊이 d에 있는 각 노드가 값 $f(n/c^d)$를 담고 있는 완전 r진 트리입니다. (편의상 n이 c의 정수 거듭제곱이어서 n/c^d가 항상 정수라고 가정해도 좋지만, 사실 이는 중요하지 않습니다.)

실제로는 그림 1.9에서처럼 트리의 처음 두세 단계를 직접 그려 보기를 권합니다.

재귀 트리의 리프는 점화식의 기저 사례에 해당합니다. 우리가 구하려는 것은 점근적 한계뿐이므로 정확한 기저 사례는 실제로 중요하지 않습니다. 즉, n_0이 임의의 양의 상수일 때 모든 $n \le n_0$에 대해 $T(n) = 1$이라고 가정해도 무방합니다. 특히 n_0의 값은 분석에 가장 편리한 것으로 얼마든지 고를 수 있습니다. 이 예제에서는 $n_0 = 1$로 정하겠습니다.

이제 $T(n)$은 재귀 트리에 있는 모든 값의 합이며, 이 합은 트리를 레벨별로 살펴봄으로써 계산할 수 있습니다. 각 정수 i에 대해 트리의 i번째 레벨에는 정확히 r^i개의 노드가 있으며, 각 노드의 값은 $f(n/c^i)$입니다. 따라서 다음과 같습니다.

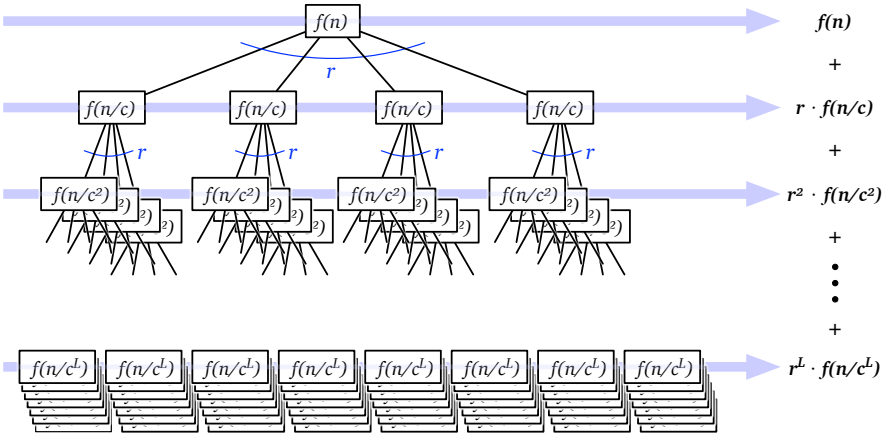


그림 1.9. 점화식 $T(n) = r\,T(n/c) + f(n)$에 대한 재귀 트리

$$T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i) \qquad (\Sigma)$$

여기서 L은 트리의 깊이입니다. $n/c^L = n_0 = 1$이므로, 기저 사례 $n_0 = 1$로부터 곧바로 $\boldsymbol{L = \log_c n}$을 얻습니다. 이로부터 재귀 트리의 잎 개수는 정확히 $r^L = r^{\log_c n} = n^{\log_c r}$입니다. 그러므로 $f(1) = O(1)$이므로 레벨별 합(Σ)의 마지막 항은 $n^{\log_c r} \cdot f(1) = O(n^{\log_c r})$입니다.

단계별 급수(Σ)를 특히 쉽게 계산할 수 있는 흔한 경우로 다음 세 가지가 있습니다.

- **감소하는 경우:** 급수가 *지수적으로 감소하면*, 즉 모든 항이 이전 항보다 상수 배만큼 작으면 $T(n) = O(f(n))$입니다. 이 경우 합은 재귀 트리의 루트에 있는 값이 지배합니다.
- **동일한 경우:** 급수의 모든 항이 같으면 곧바로 $T(n) = O(f(n) \cdot L) = O(f(n)\log n)$을 얻습니다. (상수 c는 $O()$ 표기법에 흡수되어 사라집니다.)
- **증가하는 경우:** 급수가 *지수적으로 증가하면*, 즉 모든 항이 바로 앞 항보다 일정한 상수 배만큼 크면, $T(n) = O(n^{\log_c r})$입니다. 이 경우 합은 재귀 트리의 잎 개수가 지배합니다.

첫 번째와 세 번째 경우에는 등비급수에서 가장 큰 항만 중요하며, 나머지 항은 모두 $O(\cdot)$ 표기법에 흡수됩니다. 감소하는 경우에는 L을 계산할 필요조차 없습니다. 재귀 트리가 무한하더라도 이 점근적 상한은 여전히 성립하기 때문입니다!

기초적인 예로, (단순화한) 병합 정렬 점화식 $T(n) = 2T(n/2) + O(n)$에 대한 재귀 트리의 처음 몇 단계를 그려 보면 모든 단계가 같다는 것을 알 수 있으며, 이로부터 곧바로 $T(n) = O(n \log n)$임이 도출됩니다.

재귀 트리 기법은 재귀적 부분 문제의 크기가 서로 다른 알고리즘에도 사용할 수 있습니다. 예를 들어, 피벗이 *항상* 정렬된 배열의 가운데 3분의 1 구간에 놓이도록

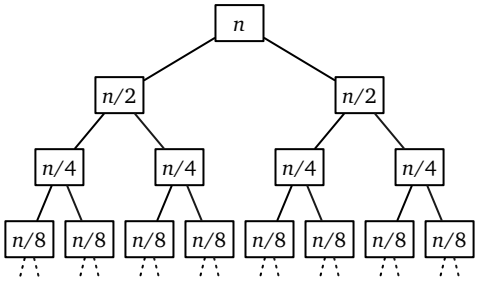


그림 1.10. 병합 정렬의 재귀 트리

퀵정렬을 어떻게든 구현할 수 있다면, 최악의 경우 실행 시간은 다음 점화식을 만족합니다.

$$T(n) \leq T(n/3) + T(2n/3) + O(n).$$

이 점화식은 무서워 보일 수도 있지만, 실제로는 꽤 다루기 쉽습니다. 그 결과로 얻는 재귀 트리의 몇 단계를 그려 보면, 어느 단계에서든 값의 합이 *많아야* n이고(더 깊은 단계에서는 일부 노드가 빠져 있을 수 있습니다) 트리 전체의 깊이가 $\log_{3/2} n = O(\log n)$이라는 사실을 금방 알 수 있습니다. 이로부터 곧바로 $T(n) = O(n \log n)$이 성립합니다. (게다가 재귀 트리에서 *꽉 찬* 단계의 수는 $\log_3 n = \Omega(\log n)$이므로, 이 보수적인 분석은 기껏해야 상수 배만큼만 개선될 수 있으며, 이는 우리의 목적상 전혀 개선되지 않는다는 뜻입니다.) 재귀 트리가 균형 잡혀 있지 않다는 사실은 전혀 중요하지 않습니다.

더 극단적인 예로, 퀵정렬의 최악의 경우 점화식 $T(n) = T(n-1) + T(1) + O(n)$은 완전히 불균형한 재귀 트리를 만들며, 이 트리에서는 각 내부 노드의 자식 중 하나가 리프입니다. 레벨별 합은 우리가 다룬 세 가지 기본 범주 중 어디에도 속하지 않지만, 모든 레벨 값이 최대 n이고 레벨이 최대 n개라는 점을 관찰하면 여전히 해 $T(n) = O(n^2)$를 유도할 수 있습니다. (이번에도 이 보수적인 분석은 정확한 상한을 줍니다. $n/2$개의 레벨이 각각 최소 $n/2$의 값을 가지기 때문입니다.)

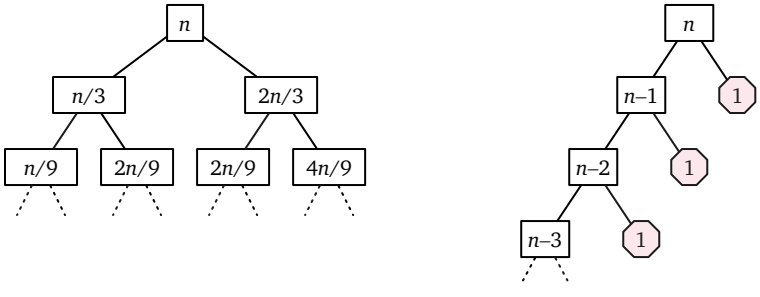


그림 1.11. 좋은 피벗을 사용한 퀵정렬(왼쪽)과 최악의 피벗을 사용한 퀵정렬(오른쪽)의 재귀 트리

♥바닥 함수와 천장 함수를 무시해도 괜찮습니다, 정말입니다

꼼꼼한 독자라면 우리의 분석이 중요한 세부 사항을 슬쩍 덮어 두었다고 반론할지도 모릅니다. 병합 정렬의 실행 시간은 *실제로는* 점화식 $T(n) = 2T(n/2) + O(n)$을 따르지 않습니다. 애초에 입력 크기 n은 홀수일 수도 있는데, 크기가 $42\frac{1}{2}$이나 $17\frac{7}{8}$인 배열을 정렬한다는 것이 대체 무슨 의미이겠습니까? 실제 병합 정렬의 점화식은 이보다 다소 더 지저분합니다:

$$T(n) = T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + O(n).$$

물론 귀납법을 사용하여 $T(n) = O(n \log n)$임을 확인할 수도 있지만, 그에 필요한 계산은 끔찍할 것입니다. 다행히 점화식에서 바닥 함수와 천장 함수를 제거하는 간단한 기법이 있는데, 이를 *정의역 변환*이라고 부릅니다.

- 먼저, 우리는 상한을 유도하고 있으므로 $T(n)$을 안전하게 과대추정할 수 있습니다. 한 번은 두 부분 문제의 크기가 같다고 간주하여 과대추정하고, 또 한 번은 천장 함수를 없애기 위해 과대추정합니다:[8]

$$T(n) \;\le\; 2T\big(\lceil n/2 \rceil\big) + n \;\le\; 2T(n/2 + 1) + n.$$

- 둘째로, 새로운 함수 $S(n) = T(n + \alpha)$를 정의하되, $S(n)$이 더 간단한 점화식 $S(n) \le 2S(n/2) + O(n)$을 만족하도록 상수 α를 선택합니다. 올바른 상수 α를 찾기 위해, 주어진 T에 대한 점화식으로부터 S에 대한 점화식을 유도합니다:

$$\begin{aligned} S(n) &= T(n + \alpha) && [S\text{의 정의}] \\ &\le 2T(n/2 + \alpha/2 + 1) + n + \alpha && [T\text{에 대한 점화식}] \\ &= 2S(n/2 - \alpha/2 + 1) + n + \alpha && [S\text{의 정의}] \end{aligned}$$

  $\alpha = 2$로 두면 이 점화식은 $S(n) \le 2S(n/2) + n + 2$로 단순화되며, 이는 정확히 우리가 원하던 결과입니다.

- 마지막으로, 재귀 트리 방법에 따르면 $S(n) = O(n \log n)$이며, 따라서

$$T(n) \;=\; S(n - 2) \;=\; O\big((n - 2)\log(n - 2)\big) \;=\; O(n \log n),$$

  약속한 그대로입니다.

비슷한 정의역 변환을 사용하면 *모든* 분할 정복 점화식에서 바닥 함수, 천장 함수, 심지어 저차항까지 제거할 수 있습니다. 하지만 이제 이 사실을 깨달았으니, 다시는 세부 사항을 힘들게 일일이 따져 볼 필요가 없습니다! 지금부터 저는 어떤 분할 정복 점화식을 마주하든 바닥 함수와 천장 함수와 저차항을 조용히 양탄자 밑으로 쓸어 넣을 것이며, 여러분도 그렇게 하시기를 권합니다.

[8] 형식적으로 말하면, 우리는 T를 정수뿐만 아니라 *실수* 위에서 정의된 함수로 다루고 있으며, 이 함수는 값이 중요하지 않은 어떤 실수 $C \ge 0$과 $n_0 > 0$에 대해 모든 $n \le n_0$에서 $T(n) = C$라는 기저 사례와 함께 주어진 점화식을 만족합니다. n이 마침 정수라면 $T(n)$은 크기가 n인 입력에 대한 알고리즘의 실행 시간과 일치하지만, 그 사실 역시 중요하지 않습니다.

1.8 ♥선형 시간 선택

퀵정렬을 논의하면서, 저는 정렬되지 않은 배열의 중앙값을 선형 시간에 찾을 수 있다고 지나가듯 언급했습니다. 이러한 알고리즘은 1970년대 초에 Manuel Blum, 밥 Floyd, Vaughan Pratt, Ron Rivest, 밥 Tarjan이 처음 발견했습니다. 사실 이들의 알고리즘은 배열과 정수 k를 입력으로 받아 n개 원소로 이루어진 배열에서 k번째로 작은 원소를 선택하는 더 일반적인 문제를 풉니다. 이를 위해 *퀵셀렉트* 또는 *외팔 퀵정렬*이라고 불리는 알고리즘의 변형을 사용합니다. 퀵셀렉트는 1961년에 Tony Hoare가 처음 기술했는데, 그가 퀵정렬을 처음 발표한 바로 그 페이지에 말 그대로 함께 실렸습니다.

퀵셀렉트

일반적인 퀵셀렉트 알고리즘은 피벗 원소를 하나 선택하고, 퀵정렬 알고리즘 QuickSort와 동일한 Partition 서브루틴을 사용하여 배열을 분할한 다음, 두 부분 배열 중 *하나만*, 구체적으로는 원래 입력 배열의 k번째로 작은 원소를 포함하는 부분 배열만 재귀적으로 탐색합니다. 퀵셀렉트의 의사코드는 그림 1.12에 나와 있습니다.

```
QuickSelect(A[1..n], k):
  if n = 1
      return A[1]
  else
      Choose a pivot element A[p]
      r ← Partition(A[1..n], p)
      if k < r
          return QuickSelect(A[1..r−1], k)
      else if k > r
          return QuickSelect(A[r+1..n], k−r)
      else
          return A[r]
```

그림 1.12. 퀵셀렉트, 또는 외팔 퀵정렬

이 알고리즘에는 두 가지 중요한 특징이 있습니다. 첫째, 퀵정렬과 마찬가지로 퀵셀렉트의 정확성은 피벗을 어떻게 선택하는지에 좌우되지 않습니다. 둘째, 실제로 관심 있는 것이 *중앙값*을 선택하는 경우(특수한 경우인 $k = n/2$)뿐이더라도, Hoare의 재귀 전략을 쓰려면 더 일반적인 선택 문제를 고려해야 합니다. 입력 배열 $A[1..n]$의 중앙값이 더 작은 두 부분 배열 $A[1..r-1]$과 $A[r+1..n]$ 중 어느 한쪽의 중앙값이 되는 경우는 거의 없기 때문입니다.

퀵셀렉트 알고리즘 QuickSelect의 최악의 경우 실행 시간은 퀵정렬 알고리즘 QuickSort와 비슷한 점화식을 따릅니다. 우리는 r의 값도, 두 부분 배열 중 어느 쪽을 재귀적으로 탐색하게 될지도 알지 못하므로 최악의 경우를 가정해야 합니다.

$$T(n) \leq \max_{1 \leq r \leq n} \max\{T(r-1), T(n-r)\} + O(n)$$

ℓ이 재귀적 부분 문제의 길이를 나타내도록 하면 점화식을 약간 단순화할 수 있습니다.

$$T(n) \leq \max_{0 \leq \ell \leq n-1} T(\ell) + O(n)$$

선택한 피벗 원소가 항상 배열에서 가장 작은 원소이거나 가장 큰 원소라면, 점화식은 $T(n) = T(n-1) + O(n)$으로 단순화되며, 이로부터 $T(n) = O(n^2)$임을 알 수 있습니다. (이 점화식의 재귀 트리는 단순한 경로에 불과합니다.)

좋은 피벗

어떤 상수 $\alpha < 1$에 대해 $\ell \le \alpha n$을 만족하는 ***좋은*** 피벗을 *어떻게든* 마법처럼 고를 수 있다면, 이러한 이차 최악의 경우 동작을 피할 수 있습니다. 이 경우 점화식은 다음과 같이 단순해집니다.

$$T(n) \le T(\alpha n) + O(n).$$

이 점화식을 전개하면 감소하는 등비급수가 되며, 이 급수는 가장 큰 항이 지배하므로 $T(n) = O(n)$입니다. (이번에도 재귀 트리는 단순한 경로일 뿐입니다. $O(n)$ 실행 시간의 상수는 상수 α에 따라 달라집니다.)

다시 말해, 중앙값에 *가깝기만* 한 원소라도 선형 시간에 어떻게든 빠르게 찾을 수 있다면, *정확한* 중앙값도 선형 시간에 찾을 수 있습니다. 그러므로 이제 우리에게 필요한 것은 근사 중앙값 요정뿐입니다. Blum-Floyd-Pratt-Rivest-Tarjan 알고리즘은 입력 배열에서 신중하게 고른 부분집합의 중앙값을 ***재귀적으로*** 계산하여 좋은 퀵셀렉트 피벗을 선택합니다. 근사 중앙값 요정은 사실 변장한 재귀 요정일 뿐입니다!

구체적으로 말하면, 입력 배열을 $\lceil n/5 \rceil$개의 블록으로 나눕니다. 이때 마지막 블록을 제외한 각 블록에는 정확히 5개의 원소가 들어갑니다. (마지막 블록이 가득 차지 않으면 ∞를 몇 개 채워 넣으면 됩니다.) 각 블록의 중앙값을 무차별 대입으로 계산하고, 그 중앙값들을 새 배열 $M[1..\lceil n/5 \rceil]$에 모은 다음, 이 새 배열의 중앙값을 *재귀적으로* 계산합니다. 마지막으로, 블록 중앙값들의 중앙값(아래 의사코드에서는 "*mom*"이라고 부릅니다)을 퀵셀렉트의 피벗으로 사용합니다.

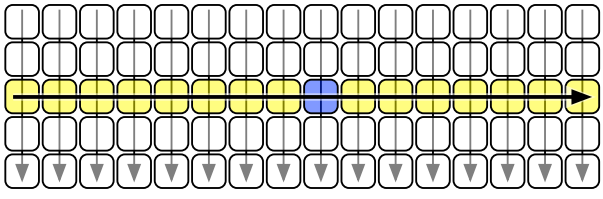
```
MomSelect(A[1..n], k):
  if n ≤ 25                                          《또는 그 밖의 무엇이든》
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          M[i] ← MedianOfFive(A[5i − 4..5i])          《무차별 대입!》
      mom ← MomSelect(M[1..m], ⌊m/2⌋)                 《재귀!》
      r ← Partition(A[1..n], mom)
      if k < r
          return MomSelect(A[1..r − 1], k)            《재귀!》
      else if k > r
          return MomSelect(A[r + 1..n], k − r)        《재귀!》
      else
          return mom
```

MomSelect는 재귀를 서로 다른 두 가지 목적으로 사용합니다. 첫 번째는 피벗 원소(*mom*)를 고르기 위해서이고, 두 번째는 그 피벗의 한쪽에 있는 원소들을 탐색하기 위해서입니다.

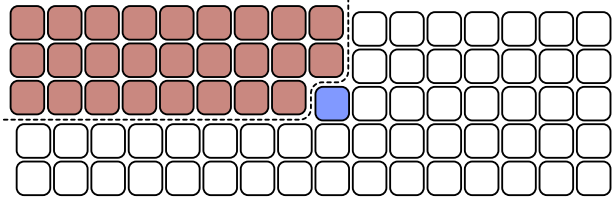
분석

그렇다면 이 알고리즘은 왜 빠릅니까? 첫 번째 핵심 통찰은 ***중앙값들의 중앙값이 좋은 피벗이라는 것***입니다. mom은 $\lfloor \lceil n/5 \rceil / 2 \rfloor - 1 \approx n/10$개의 블록 중앙값보다 크고, 각 블록 중앙값은 자기 블록의 다른 두 원소보다 큽니다. 따라서 mom은 입력 배열에서 적어도 $3n/10$개의 원소보다 크며, 대칭적으로 mom은 적어도 $3n/10$개의 원소보다 작습니다. 따라서 최악의 경우에도 두 번째 재귀 호출은 크기가 최대 $7n/10$인 배열을 탐색합니다.

입력 배열을 $5 \times \lceil n/5 \rceil$ 격자로 그리면 알고리즘의 동작을 시각화할 수 있습니다. 이 격자에서 각 열은 연속된 다섯 개의 원소를 나타냅니다. 설명을 위해, 모든 열을 위에서 아래로 정렬한 다음 열들을 가운데 원소를 기준으로 정렬한다고 상상해 봅시다. (*알고리즘이 실제로 이렇게 하지는 않습니다!* 이 점을 강조해 둡니다.) 이렇게 배치하면 중앙값들의 중앙값은 격자의 중심에 가장 가까운 원소입니다.



격자의 처음 세 행의 왼쪽 절반에는 $3n/10$개의 원소가 있으며, 이들은 각각 mom보다 작습니다. 우리가 찾는 원소가 mom보다 크다면, 알고리즘은 재귀하기 전에 이 $3n/10$개의 원소를 포함하여 mom보다 작은 *모든 것*을 버립니다. 따라서 재귀적 부분 문제의 입력에는 최대 $7n/10$개의 원소가 들어 있습니다. 대칭적인 논증에 따르면, 대상 원소가 mom보다 작은 경우에는 mom보다 큰 원소를 적어도 $3n/10$개 버리므로, 재귀적 부분 문제의 입력에는 최대 $7n/10$개의 원소가 남습니다.



좋습니다. mom이 좋은 피벗이라는 것은 확인했지만, 우리 알고리즘은 여전히 재귀 호출을 한 번이 아니라 *두 번* 수행합니다. 그렇다면 선형 시간을 어떻게 증명할 수 있겠습니까? 두 번째 핵심 통찰은 두 재귀적 부분 문제의 *전체* 크기가 원래 입력 배열의 크기보다 상수 배만큼 작다는 것입니다. 이 알고리즘의 최악의 경우 실행 시간은 다음 점화식을 따릅니다.

$$T(n) \leq T(n/5) + T(7n/10) + O(n).$$

이 점화식의 재귀 트리를 그려 보면, 재귀 트리의 각 레벨에서 수행하는 총 작업량이 이전 레벨 총 작업량의 $9/10$ 이하임을 알 수 있습니다. 따라서 레벨별 합은 지수적으

로 감소하며, 이로부터 해 $T(n) = O(n)$을 얻습니다. (다시 말하지만, 재귀 트리가 불균형하다는 사실은 전혀 중요하지 않습니다.) 만세! 엄마, 고맙습니다!

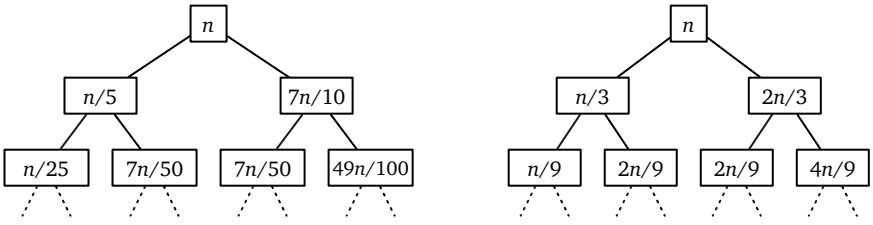


그림 1.13. MomSelect와 블록 크기가 3인 유사한 선택 알고리즘의 재귀 트리

타당성 검사

이쯤에서 많은 학생이 그 마법의 상수 5에 대해 묻습니다. 왜 하필 그 블록 크기를 선택했을까요? 그 답은 5가 재귀 트리 분석에서 지수적 감소를 얻을 수 있는 가장 작은 홀수 블록 크기라는 것입니다! (짝수 블록 크기는 추가적인 복잡한 문제를 일으킵니다.) 만약 크기가 3인 블록을 대신 사용했다면, 실행 시간 점화식은 다음과 같았을 것입니다.

$$T(n) \le T(n/3) + T(2n/3) + O(n).$$

우리는 이 점화식을 전에 본 적이 있습니다! 재귀 트리의 모든 레벨은 총합이 *최대* n이고, 재귀 트리의 깊이는 $\log_{3/2} n = O(\log n)$이므로, 이 점화식의 해는 $T(n) \le O(n \log n)$입니다. (게다가 재귀 트리에는 완전한 레벨이 $\log_3 n$개 있으므로 이 분석은 빈틈이 없습니다.) 3개 원소짜리 블록을 사용하는 중앙값들의 중앙값 선택은 정렬보다 빠르지 않습니다.

더 정밀하게 분석해 보면, 비교만 세더라도 $O()$ 표기법에 숨겨진 상수가 상당히 크다는 것을 알 수 있습니다. 5개 원소의 중앙값을 선택하는 데는 최대 6번의 비교가 필요하므로, 재귀적 부분 문제를 구성하는 데 최대 $6n/5$번의 비교가 필요합니다. 재귀 호출 후에 배열을 단순하게 분할하면 $n - 1$번의 비교가 필요하지만, 우리는 이미 피벗보다 큰 원소 $3n/10$개와 피벗보다 작은 원소 $3n/10$개를 알고 있으므로 실제로 분할에는 $2n/5$번의 추가 비교만 필요합니다. 따라서 최악의 경우 비교 횟수에 대한 더 정확한 점화식은 다음과 같습니다.

$$T(n) \le T(n/5) + T(7n/10) + 8n/5.$$

재귀 트리 방법을 사용하면 다음과 같은 상한을 얻습니다.

$$T(n) \le \frac{8n}{5} \sum_{i \ge 0} \left(\frac{9}{10}\right)^i = \frac{8n}{5} \cdot 10 = 16n.$$

실제로는 중앙값들의 중앙값 선택이 이 최악의 경우 분석이 예측하는 것만큼 느리지 않습니다. 재귀의 모든 단계에서 최악의 피벗이 선택될 가능성은 극히 낮기 때문입니다. 그러나 적당히 큰 배열에서도 여전히 정렬보다 느립니다.[9]

[9] 사실 실제로 피벗 원소를 고르는 올바른 방법은 *균등하게 무작위로* 고르는 것입니다. 그러면 중앙값을

1.9 빠른 곱셈

이전 장에서 우리는 두 n자리 수를 $O(n^2)$ 시간에 곱하는 두 가지 고대 알고리즘인 초등학교 격자 알고리즘과 이집트 농부 알고리즘을 살펴보았습니다.

자릿수 배열을 반으로 나누고 다음 항등식을 활용하면 더 효율적인 알고리즘을 얻을 수 있을지도 모릅니다:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

이 점화식을 보면 두 n자리 수 x와 y를 곱하는 다음 분할 정복 알고리즘이 곧바로 떠오릅니다. 네 개의 부분 곱 ac, bc, ad, bd는 각각 재귀적으로 계산하지만, 마지막 줄의 곱셈은 재귀적이지 *않습니다*. 10의 거듭제곱을 곱하는 일은 자릿수를 왼쪽으로 옮기고 알맞은 개수의 0을 채워 넣는 것으로 처리할 수 있으며, 이 모든 작업이 $O(n)$ 시간에 끝나기 때문입니다.

```
SplitMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋; b ← x mod 10^m      ⟪x = 10^m a + b⟫
      c ← ⌊y/10^m⌋; d ← y mod 10^m      ⟪y = 10^m c + d⟫
      e ← SplitMultiply(a, c, m)
      f ← SplitMultiply(b, d, m)
      g ← SplitMultiply(b, c, m)
      h ← SplitMultiply(a, d, m)
      return 10^(2m) e + 10^m (g + h) + f
```

이 알고리즘의 정확성은 귀납법으로 쉽게 증명됩니다. 이 알고리즘의 실행 시간은 다음 점화식을 따릅니다.

$$T(n) = 4T(\lceil n/2 \rceil) + O(n).$$

재귀 트리 방법을 쓰면 이 점화식은 *증가하는* 등비급수가 되며, 따라서 $T(n) = O(n^{\log_2 4}) = O(n^2)$입니다. 사실 이 알고리즘은 격자 알고리즘과 마찬가지로 x의 각 자릿수를 y의 각 자릿수와 하나하나 곱합니다. 그러니 이 방법은 통하지 않은 것 같습니다. 아쉽습니다. 좋은 아이디어였는데 말입니다.

1950년대 중반, 20세기 수학의 거장 중 한 사람인 안드레이 콜모고로프(Andrei Kolmogorov)는 두 n자리 수를 이차 미만 시간에 곱하는 알고리즘은 *존재하지 않는다*고 공개적으로 추측했습니다. 콜모고로프는 1960년 모스크바 대학교에서 세미나를 열었습니다. 그 자리에서 그는 자신의 "n^2 추측"을 다시 내세우며 이후 모임에서 다룰 계획이었던 몇 가지 관련 문제를 제시했습니다. 거의 정확히 일주일 뒤, 아나톨리 카라추바(Anatoliĭ Karatsuba)라는 23세 학생이 콜모고로프에게 놀라운 반례를 내놓았습니다. 카라추바 본인의 말에 따르면 다음과 같습니다.

찾는 데 필요한 비교 횟수의 기댓값은 최대 $4n$입니다. 더 자세한 내용은 http://algorithms.wtf 에 있는 제 무작위 알고리즘 강의 노트를 참고하십시오.

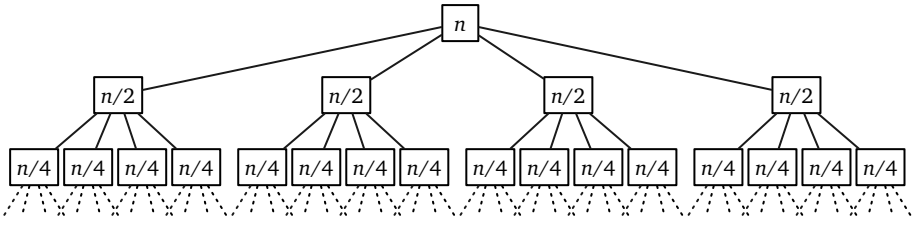


그림 1.14. 단순한 분할 정복 곱셈의 재귀 트리

> 세미나가 끝난 뒤 저는 콜모고로프에게 새 알고리즘과 n^2 추측의 반증에 대해 이야기했습니다. 이것이 자신의 매우 그럴듯한 추측과 모순되었기 때문에 콜모고로프는 몹시 동요했습니다. 다음 세미나 모임에서 콜모고로프는 직접 참석자들에게 제 방법을 소개했고, 그 시점에서 세미나는 종료되었습니다.

카라추바는 다음 대수적 항등식을 이용하면 가운데 계수 $bc + ad$를 나머지 두 계수 ac와 bd로부터 재귀 곱셈을 *단 한 번*만 더 수행하여 계산할 수 있음을 관찰했습니다.

$$ac + bd - (a - b)(c - d) = bc + ad$$

이 요령을 사용하면 아래와 같이 이전 알고리즘의 재귀 호출 네 번을 단 세 번의 재귀 호출로 대체할 수 있습니다.

```
FastMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋; b ← x mod 10^m      ⟪x = 10^m a + b⟫
      c ← ⌊y/10^m⌋; d ← y mod 10^m      ⟪y = 10^m c + d⟫
      e ← FastMultiply(a, c, m)
      f ← FastMultiply(b, d, m)
      g ← FastMultiply(a − b, c − d, m)
      return 10^(2m) e + 10^m (e + f − g) + f
```

카라추바의 FastMultiply 알고리즘의 실행 시간은 다음 점화식을 따릅니다.

$$T(n) \le 3T(\lceil n/2 \rceil) + O(n)$$

이번에도 재귀 트리 방법을 쓰면 이 점화식은 증가하는 등비급수가 되지만, 새로운 해는 $T(n) = O(n^{\log_2 3}) = \boldsymbol{O(n^{1.58496})}$에 불과합니다. 이는 앞서 구한 이차 시간 한계보다 크게 개선된 결과입니다.[10] 카라추바의 알고리즘은 알고리즘의 설계와 분석을 정식 학문 분야로 출범시켰다고 볼 수 있습니다.

[10] 이 설명은 실제 역사를 약간 단순화한 것입니다. 사실 카라추바는 $(a+b)(c+d) - ac - bd = bc + ad$라는 공식에 기반한 알고리즘을 제안했습니다. 이 알고리즘 역시 $O(n^{\lg 3})$ 시간에 실행되지만, 실제 점화식은 조금 더 복잡합니다. $a - b$와 $c - d$는 여전히 m자리 수이지만, $a + b$와 $c + d$는 각각 $m + 1$자리가 될 수도 있기 때문입니다. 여기에 제시한 단순화는 도널드 커누스가 고안한 것입니다.

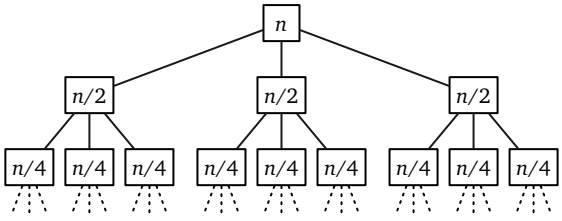


그림 1.15. 카라추바의 분할 정복 곱셈 알고리즘의 재귀 트리

카라추바의 아이디어를 더 발전시켜 수를 더 많은 조각으로 나누고 더 복잡한 방식으로 결합하면 훨씬 더 빠른 곱셈 알고리즘을 얻을 수 있습니다. 안드레이 툼(Andrei Toom)은 임의의 정수를 각각 n/k 자리인 k개의 부분으로 나눈 뒤, 단 $2k-1$번의 재귀 곱셈만으로 곱을 계산하는 무한한 알고리즘족을 발견했습니다. 이후 스티븐 쿡(Stephen Cook)은 박사 학위 논문에서 툼의 알고리즘을 더욱 단순화했습니다. 임의의 고정된 k에 대해 툼-쿡 알고리즘은 $O(n^{1+1/(\lg k)})$ 시간에 실행되며, 여기서 $O(\cdot)$ 표기법에 숨겨진 상수는 k에 따라 달라집니다.

결국 이 분할 정복 전략 덕분에 가우스는(정말입니다) ***고속 푸리에 변환***을 발견했습니다.[11] 기본 FFT 알고리즘 자체는 $O(n \log n)$ 시간에 실행됩니다. 그러나 정수 곱셈에 FFT를 사용하면 약간의 추가 오버헤드가 생깁니다. 1971년 아르놀트 쇤하게(Arnold Schönhage)와 폴커 슈트라센(Volker Strassen)이 발표한 최초의 FFT 기반 정수 곱셈 알고리즘은 $O(n \log n \log\log n)$ 시간에 실행됩니다. 쇤하게-슈트라센 알고리즘은 수십 년 동안 이론적으로 가장 빠른 정수 곱셈 알고리즘의 자리를 지켰습니다. 그 뒤 마르틴 퓌러(Martin Fürer)가 오랫동안 이어진 일련의 기술적 개선 가운데 첫 번째 개선을 발견했습니다. 마침내 2019년에 데이비드 하비(David Harvey)와 요리스 판 데르 회번(Joris van der Hoeven)이 $O(n \log n)$ 시간에 실행되는 알고리즘을 발표했습니다.[12]

1.10 거듭제곱

수 a와 양의 정수 n이 주어졌을 때 a^n을 계산하려 한다고 합시다. 표준적인 단순한 방법은 a를 곱하는 곱셈을 $n-1$번 수행하는 간단한 for 루프입니다.

```
SlowPower(a, n):
  x ← a
  for i ← 2 to n
      x ← x · a
  return x
```

이 반복 알고리즘에는 n번의 곱셈이 필요합니다.

[11] 고속 푸리에 변환에 관한 강의 노트는 http://algorithms.wtf 를 참고하십시오.

[12] 사실 약 75000자리가 넘는 정수를 곱할 때 *실제*로 가장 빠른 알고리즘은 쇤하게-슈트라센 알고리즘입니다. 퓌러, 하비, 판 데르 회번 등이 내놓은 더 최근의 알고리즘들은 자릿수가 우주에 있는 입자 수보다 많은 정수에서만 "실제로" 더 빠릅니다.

입력 매개변수 a는 정수일 수도 있고, 유리수일 수도 있으며, 부동소수점 수일 수도 있습니다. 사실 곱하는 방법만 알고 있다면 a는 수일 필요조차 없습니다. 예를 들어 같은 알고리즘으로 어떤 유한한 수를 법으로 하는 거듭제곱(암호 알고리즘에서 흔히 쓰이는 연산)을 계산할 수도 있고, 행렬의 거듭제곱(점화식의 값을 계산하거나 그래프에서 최단 경로를 계산하는 데 쓰이는 연산)을 계산할 수도 있습니다. 우리가 어떤 종류의 대상을 곱하고 있는지 모르기 때문에 곱셈 한 번에 시간이 얼마나 걸리는지 알 *수 없습니다*. 따라서 실행 시간을 곱셈 횟수로 분석할 수밖에 없습니다.

훨씬 더 빠른 분할 정복 방법도 있습니다. 이 방법은 원래 인도의 운율학자 핑갈라(Piṅgala)가 기원전 2세기에 제안한 것으로, 다음과 같은 간단한 재귀 공식을 사용합니다:

$$a^n = \begin{cases} 1 & \text{만약} n = 0 \\ (a^{n/2})^2 & \text{만약} n > 0 \text{그리고} n \text{짝수입니다} \\ (a^{\lfloor n/2 \rfloor})^2 \cdot a & \text{그렇지 않으면} \end{cases}$$

```
PiṅgalaPower(a, n):
  if n = 1
      return a
  else
      x ← PiṅgalaPower(a, ⌊n/2⌋)
      if n is even
          return x · x
      else
          return x · x · a
```

이 알고리즘이 수행하는 곱셈의 총 횟수는 점화식 $T(n) \le T(n/2) + 2$를 만족합니다. 재귀 트리 방법을 사용하면 곧바로 $T(n) = O(\log n)$이라는 해를 얻을 수 있습니다.

이전 장의 이집트 농부 곱셈 알고리즘에서 덧셈을 곱셈으로 바꾸면(특히 배가(duplation)를 제곱으로 바꾸면) 거의 동일한 거듭제곱 알고리즘을 직접 유도할 수도 있습니다.

$$a^n = \begin{cases} 1 & \text{만약} n = 0 \\ (a^2)^{n/2} & \text{만약} n > 0 \text{그리고} n \text{짝수입니다} \\ (a^2)^{\lfloor n/2 \rfloor} \cdot a & \text{그렇지 않으면} \end{cases}$$

```
PeasantPower(a, n):
  if n = 1
      return a
  else if n is even
      return PeasantPower(a², n/2)
  else
      return PeasantPower(a², ⌊n/2⌋) · a
```

"제곱과 반분(squaring and mediation)"이라고 불러도 무리가 없을 이 알고리즘 역시 곱셈을 $O(\log n)$번만 수행합니다.

이 두 알고리즘은 모두 점근적으로 최적입니다. a^n을 계산하는 알고리즘은 어떤 것이든 *반드시* 최소 $\Omega(\log n)$번의 곱셈을 수행해야 합니다. 곱셈을 한 번 할 때마다 지금까지 계산한 가장 큰 거듭제곱의 지수가 많아야 두 배로 늘어나기 때문입니다. 실제로 n이 2의 거듭제곱이면 두 알고리즘 모두 정확히 $\log_2 n$번의 곱셈이 필요하며, 이는 *정확히* 최적입니다. 그러나 n이 다른 값일 때는 조금 더 빠른 방법이 있습니다. 예를 들어 Pi˙ngalaPower와 PeasantPower는 각각 곱셈 여섯 번으로 a^{15}를 계산하지만, 실제로는 곱셈 다섯 번이면 충분합니다:

- 핑갈라: $a \to a^2 \to a^3 \to a^6 \to a^7 \to a^{14} \to a^{15}$
- 농부 방식: $a \to a^2 \to a^4 \to a^8 \to a^{12} \to a^{14} \to a^{15}$
- 최적: $a \to a^2 \to a^3 \to a^5 \to a^{10} \to a^{15}$

주어진 지수 n에 대해 필요한 곱셈 횟수의 절대적 최솟값을 효율적으로 계산할 수 있는지는 오랫동안 풀리지 않은 문제입니다.

연습문제

하노이의 탑

1. 원래의 재귀적 하노이의 탑 알고리즘이 다음의 각 비재귀 알고리즘과 *정확히* 같은 이동 순서를 수행함을 증명하십시오. 즉, 같은 원판을 같은 기둥에서 같은 기둥으로 같은 순서대로 옮김을 증명하십시오. 기둥에는 0, 1, 2라는 이름이 붙어 있으며, 우리의 문제는 (24쪽에 나와 있는 것처럼) n개의 원판 더미를 기둥 0에서 기둥 2로 옮기는 것입니다.

   (a) n이 짝수이면 기둥 1과 2를 서로 바꾸십시오. i번째 단계에서는 기둥 $i \bmod 3$을 피하는 유일한 합법적인 이동을 하십시오. 합법적인 이동이 없다면 모든 원판이 기둥 $i \bmod 3$에 있는 것이므로 퍼즐이 풀린 것입니다.

   (b) 첫 번째 이동에서는 n이 짝수이면 원판 1을 기둥 1로, n이 홀수이면 기둥 2로 옮기십시오. 그다음에는 직전 이동과 다른 원판을 움직이는 유일한 합법적 이동을 반복해서 수행하십시오. 그러한 이동이 더 이상 존재하지 않으면 퍼즐이 풀린 것입니다.

   (c) 원판 $n+1$, $n+2$, $n+3$이 각각 기둥 0, 1, 2의 맨 아래에 놓여 있다고 가정하십시오. 다음 조건을 만족하는 합법적인 이동은 하나뿐이며, 그러한 이동이 더 이상 불가능할 때까지 이 이동을 반복하십시오.

   - 홀수 원판을 다른 홀수 원판 바로 위에 놓지 마십시오.
   - 짝수 원판을 다른 짝수 원판 바로 위에 놓지 마십시오.
   - 직전의 수를 되돌리지 마십시오.

   (d) $n/2^k$가 정수가 아니게 되는 가장 작은 정수 k를 $\rho(n)$으로 나타냅시다. 예를 들어 $42/2^1$은 정수이지만 $42/2^2$는 정수가 아니므로 $\rho(42) = 2$입니다. (같은

말로, $\rho(n)$은 n의 이진 표현에서 최하위 1의 위치에 1을 더한 값입니다.) 그 동작이 자의 눈금과 닮았기 때문에 $\rho(n)$을 *자 함수*라고 부르기도 합니다.

```
RulerHanoi(n):
  i ← 1
  while ρ(i) ≤ n
      if n − i is even
          move disk ρ(i) forward      ⟪0 → 1 → 2 → 0⟫
      else
          move disk ρ(i) backward     ⟪0 → 2 → 1 → 0⟫
  i ← i + 1
```

2. 하노이의 탑은 훨씬 더 오래된 기계식 퍼즐에서 비교적 최근에 파생된 퍼즐입니다. 이 오래된 퍼즐은 중국 연결 고리(Chinese linked rings), 바그노디에(Baguenaudier), 카르다노의 고리(Cardan's Rings), 멜레다(Meleda), 페이션스(Patience), 타이어링 아이언(Tiring Irons), 죄수의 자물쇠(Prisoner's Lock), 스핀아웃(Spin-Out) 등 수많은 이름으로 알려져 있습니다. 이 퍼즐은 16세기에 이미 중국과 유럽 양쪽에서 널리 알려져 있었습니다. 이탈리아의 수학자 루카 파치올리(Luca Pacioli)는 1498년에서 1506년 사이에 집필한 미출간 논고 *De Viribus Quantitatis*에서 7고리 퍼즐과 그 해법을 서술하였습니다.[13] 그로부터 불과 몇 년 뒤, 명나라의 시인 양신(Yang Shen)은 9고리 퍼즐을 "여인과 아이들을 위한 장난감"이라고 묘사하였습니다. 근거는 없지만 이 퍼즐은 2세기 중국의 한 장군이 만들었다고 전해집니다. 이 장군은 전쟁에 나가 있는 동안 아내가 시간을 보낼 수 있도록 이 퍼즐을 주었다고 합니다.

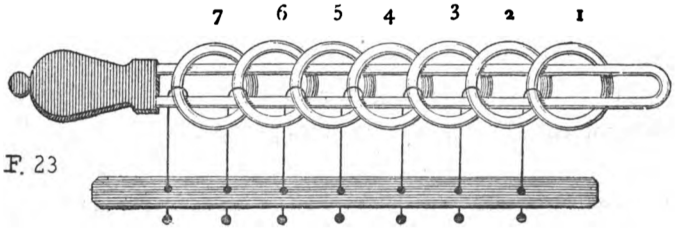


그림 1.16. 에두아르 뤼카의 *Récréations Mathématiques*(1891)에 실린 7고리 바게노디에 (책 끝의 이미지 출처를 참조하십시오.)

바그노디에(Baguenaudier) 퍼즐은 물리적 형태가 다양하지만, 가장 흔한 형태 중 하나는 긴 금속 고리 하나와 여러 개의 링으로 이루어져 있으며, 이 링들은 움직일 수 있는 막대로 단단한 받침대에 연결되어 있습니다. 처음에는 그림 1.16에 나타낸 것처럼 고리가 링들에 꿰어져 있으며, 퍼즐의 목표는 이 고리를 빼내는 것입니다.

[13] *De Viribus Quantitatis* [*수의 힘에 관하여*]는 오락 수학에 관한 중요한 초기 저작이며, 아마도 현존하는 가장 오래된 마술 논고일 것입니다. 파치올리는 15세기 후반 수학을 거의 빠짐없이 담은 백과사전인 *Summa de Aritmetica*로 더 잘 알려져 있으며, 이 책에는 복식부기에 관한 최초의 서술이 담겨 있습니다.

좀 더 추상적으로 말하면, 이 퍼즐은 고리마다 비트 하나씩을 대응시킨 비트 수열로 모델링할 수 있습니다. 이때 루프가 i번째 고리를 통과하면 i번째 비트는 1이고, 그렇지 않으면 0입니다. (여기서는 그림 1.16에 나타낸 것처럼 고리에 오른쪽부터 왼쪽으로 번호를 매깁니다.) 이 퍼즐에서 허용되는 합법적인 이동은 다음 두 가지입니다:

- 언제나 1번째(= 가장 오른쪽) 비트는 뒤집을 수 있습니다.
- 비트 문자열이 정확히 z개의 0으로 끝나면 $(z+2)$번째 비트를 뒤집을 수 있습니다.

이 퍼즐의 목표는 n개의 1로 이루어진 문자열을 n개의 0으로 이루어진 문자열로 바꾸는 것입니다. 예를 들어, 다음과 같이 21번 이동하면 고리 5개짜리 퍼즐을 풀 수 있습니다:

$$\begin{aligned}
&\texttt{11111} \xrightarrow{1} \texttt{11110} \xrightarrow{3} \texttt{11010} \xrightarrow{1} \texttt{11011} \xrightarrow{2} \texttt{11001} \xrightarrow{1} \texttt{11000} \\
&\xrightarrow{5} \texttt{01000} \xrightarrow{1} \texttt{01001} \xrightarrow{2} \texttt{01011} \xrightarrow{1} \texttt{01010} \xrightarrow{3} \texttt{01110} \\
&\xrightarrow{1} \texttt{01111} \xrightarrow{2} \texttt{01101} \xrightarrow{1} \texttt{01100} \xrightarrow{4} \texttt{00100} \xrightarrow{1} \texttt{00101} \\
&\xrightarrow{2} \texttt{00111} \xrightarrow{1} \texttt{00110} \xrightarrow{3} \texttt{00010} \xrightarrow{1} \texttt{00011} \xrightarrow{2} \texttt{00001} \xrightarrow{1} \texttt{00000}
\end{aligned}$$

♦(a) 어떤 이동도 바로 직전 이동의 역이 아닌 이동 순서열을 *축약된* 순서열이라고 부릅시다. 임의의 음이 아닌 정수 n에 대해, n-고리 바게노디에 퍼즐을 푸는 축약된 이동 순서열이 *정확히 하나* 존재함을 증명하십시오. *[힌트: 그래프에 이미 익숙하다면 이 문제는 훨씬 쉬워집니다.]*

(b) Baguenaudier 퍼즐을 푸는 알고리즘을 기술하십시오. 입력은 고리의 개수 n이며, 알고리즘은 퍼즐을 푸는 축약된 이동 수열을 출력해야 합니다. 예를 들어 정수 5가 입력으로 주어지면 알고리즘은 수열 1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1을 출력해야 합니다.

(c) 여러분의 알고리즘은 n에 대한 함수로 나타낼 때 *정확히* 몇 번 이동합니까? 여러분의 답이 옳음을 증명하십시오.

3. 하노이의 탑 역사에서 덜 알려진 이야기로, 13세기 초에 사원이 베나레스에서 피사로 잠시 옮겨진 일이 있습니다.[14] 이 이전은 부유한 상인이자 수학자였던 레오나르도 피보나치가 주도했습니다. 그는 신성 로마 제국의 황제 프리드리히 2세의 요청을 받았는데, 황제는 십자군 원정에서 돌아온 병사들에게서 이 사원에 관한 이야기를 전해 들은 바 있었습니다. 피사의 탑과 그곳을 지키는 수도승들은 유명해졌고, 이는 피사가 이탈리아반도에서 손꼽히는 교역 중심지로 자리 잡는 데 이바지했습니다.

   안타깝게도 사원을 옮기자마자 다이아몬드 바늘 하나가 한쪽으로 기울기 시작했습니다. 너무 많이 사용되어 기울어진 탑이 쓰러질 수 있다는 우려 때문에, 피보나치는 사제들을 설득하여 더 완화된 규칙을 채택하게 했습니다: ***기울어진 바늘에 있는 원반은 개수에 상관없이 한 번의 이동으로 다른 바늘에 함께 옮길 수 있습니다.*** 더

[14] 이 이야기의 일부는 실제로 사실입니다.

큰 원반을 더 작은 원반 위에 올려놓는 것은 여전히 금지되었으며, 원반을 기울어진 바늘 *위로* 옮기거나 두 수직 바늘 사이에서 옮길 때는 한 번에 하나씩 옮겨야 했습니다.

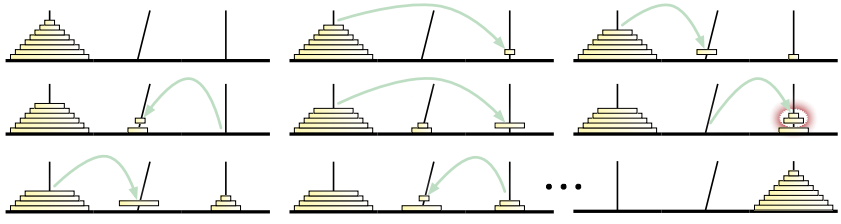


그림 1.17. 피사의 탑. 다섯 번째 이동에서는 기울어진 기둥에서 원반 두 개를 빼냅니다.

피보나치의 새 규칙 덕분에 사제들은 베나레스에서보다 피사에서 우주의 종말을 조금 더 빨리 앞당길 수 있었습니다. 다행히도 새로 즉위한 교황 그레고리오 9세가 프리드리히 2세를 파문한 뒤로 피사의 사제들은 기이한 수학적 습관을 지닌 외국 이단자들을 받아들이는 데 전보다 덜 호의적이 되었고, 그 결과 신전은 피사에서 다시 베나레스로 옮겨졌습니다. 그 후 얼마 지나지 않아 신전이 있던 자리에 종탑이 세워졌는데, 이 종탑 역시 거의 세워지자마자 기울기 시작했습니다.

가능한 한 가장 적은 이동 횟수로 n개의 원판 더미를 한 *수직* 바늘에서 다른 *수직* 바늘로 옮기는 알고리즘을 설명하십시오. 여러분의 알고리즘은 *정확히* 몇 번 이동합니까?

4. 다음과 같이 제한된 하노이의 탑 퍼즐 변형들을 생각해 봅시다. 각 문제에서 기둥에는 0, 1, 2의 번호가 붙어 있으며, 여러분의 과제는 문제 1과 똑같이 n개의 원판으로 이루어진 더미를 기둥 0에서 기둥 2로 옮기는 것입니다.

   (a) 기둥 1과 기둥 2 사이에서 원판을 직접 옮기는 것이 금지되어 있다고 가정합시다. 즉, 모든 이동에는 기둥 0이 관여해야 합니다. 이 버전의 퍼즐을 가능한 한 적은 이동 횟수로 푸는 알고리즘을 설명하십시오. 여러분의 알고리즘은 *정확히* 몇 번 이동합니까?

   ♣♥(b) 원반을 기둥 0에서 기둥 2로, 기둥 2에서 기둥 1로, 또는 기둥 1에서 기둥 0으로만 옮길 수 있다고 가정합시다. 이는 기둥들이 원형으로 배치되어 시계 방향으로 번호가 매겨져 있고, 원반을 반시계 방향으로만 옮길 수 있다고 가정하는 것과 같습니다. 이 버전의 퍼즐을 가능한 한 적은 횟수로 옮겨서 푸는 알고리즘을 설명하십시오. 여러분의 알고리즘은 원반을 몇 번 옮깁니까?

   ♣♥(c) 마지막으로, 유일한 제약이 원반을 기둥 0에서 기둥 2로 직접 옮길 수 없다는 것뿐이라고 가정합시다. 이 버전의 퍼즐을 가능한 한 적은 횟수의 이동으로 푸는 알고리즘을 설명하십시오. 여러분의 알고리즘은 원반을 몇 번 옮깁니까? *[힌트: 행렬! 이 변형은 다른 두 변형보다 분석하기가 훨씬 더 어렵습니다.]*

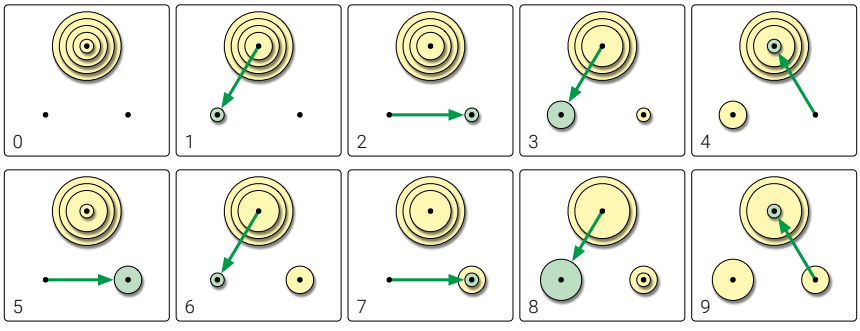


그림 1.18. 반시계 방향 하노이의 탑 풀이에서 처음 몇 차례의 이동입니다.

5. 하노이의 탑 퍼즐을 더 복잡하게 바꾼 다음 변형을 생각해 봅시다. 이 퍼즐에는 1부터 k까지 번호가 매겨진 k개의 기둥이 한 줄로 놓여 있습니다. 한 번의 차례에 임의의 인덱스 i에 대해 기둥 i의 가장 작은 원판을 기둥 $i-1$이나 기둥 $i+1$로 옮길 수 있습니다. 평소와 같이 작은 원판 위에 큰 원판을 놓을 수는 없습니다. 여러분의 임무는 n개의 원판 더미를 기둥 1에서 기둥 k로 옮기는 것입니다.

   (a) $k = 3$인 경우에 대한 재귀 알고리즘을 서술하십시오. 여러분의 알고리즘은 *정확히* 몇 번 이동합니까? (이 문제는 문제 4(a)와 정확히 같습니다.)

   (b) $k = n + 1$인 경우에 대해 최대 $O(n^3)$번의 이동을 필요로 하는 재귀 알고리즘을 설명하십시오. *[힌트: (a)를 이용하십시오.]*

   ♥(c) $k = n + 1$인 경우에 최대 $O(n^2)$번 이동하는 재귀 알고리즘을 설명하십시오. *[힌트: (a)를 사용하지 마십시오.]*

   ♥(d) $k = \sqrt{n}$인 경우에 최대 다항식 횟수의 이동만 필요로 하는 재귀 알고리즘을 설명하십시오. (어떤 다항식입니까??)

   ♥(e) 임의의 n과 k에 대한 재귀 알고리즘을 기술하고 분석하십시오. 이동 횟수가 n에 대한 다항식으로 제한되려면 k는 (n의 함수로서) 얼마나 작아야 합니까?

재귀 트리

6. 재귀 트리를 사용하여 다음 점화식을 각각 푸십시오.

$$A(n) = 2A(n/4) + \sqrt{n} \quad B(n) = 2B(n/4) + n \quad C(n) = 2C(n/4) + n^2$$
$$D(n) = 3D(n/3) + \sqrt{n} \quad E(n) = 3E(n/3) + n \quad F(n) = 3F(n/3) + n^2$$
$$G(n) = 4G(n/2) + \sqrt{n} \quad H(n) = 4H(n/2) + n \quad I(n) = 4I(n/2) + n^2$$

7. 재귀 트리를 사용하여 다음 각 점화식을 푸십시오.

   (j) $J(n) = J(n/2) + J(n/3) + J(n/6) + n$

   (k) $K(n) = K(n/2) + 2K(n/3) + 3K(n/4) + n^2$

   (l) $L(n) = L(n/15) + L(n/10) + 2L(n/6) + \sqrt{n}$

♥8. 재귀 트리를 사용하여 다음 점화식을 각각 푸십시오.

(m) $M(n) = 2M(n/2) + O(n \log n)$

(n) $N(n) = 2N(n/2) + O(n/\log n)$

(p) $P(n) = \sqrt{n}\,P(\sqrt{n}) + n$

(q) $Q(n) = \sqrt{2n}\,Q(\sqrt{2n}) + \sqrt{n}$

정렬

9. 크기가 모두 다른 팬케이크 n개가 쌓여 있다고 가정합시다. 작은 팬케이크가 큰 팬케이크 위에 오도록 팬케이크를 정렬하려고 합니다. 수행할 수 있는 연산은 *뒤집기* 하나뿐입니다. 즉, 1과 n 사이의 어떤 정수 k에 대해 맨 위의 팬케이크 k개 아래에 뒤집개를 넣고 그 팬케이크들을 한꺼번에 뒤집는 것입니다.

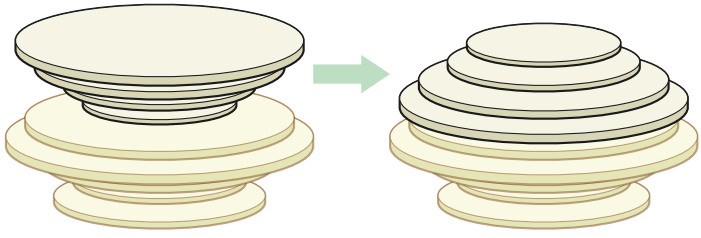


그림 1.19. 맨 위의 팬케이크 네 장을 뒤집습니다.

(a) $O(n)$번의 뒤집기로 임의의 n개짜리 팬케이크 더미를 정렬하는 알고리즘을 설명하십시오. 여러분의 알고리즘은 최악의 경우 *정확히* 몇 번의 뒤집기를 수행합니까?[15] *[힌트: 이 문제는 하노이의 탑과 아무 관련이 없습니다.]*

(b) 모든 양의 정수 n에 대해, 정렬하는 데 $\Omega(n)$번의 뒤집기가 필요한 팬케이크 n개짜리 더미를 설명하십시오.

(c) 이제 각 팬케이크의 한쪽 면이 탔다고 가정합시다. 모든 팬케이크의 탄 면이 아래를 향하도록, 임의의 n개 팬케이크 더미를 $O(n)$번의 뒤집기로 정렬하는 알고리즘을 설명하십시오. 여러분의 알고리즘은 최악의 경우 *정확히* 몇 번의 뒤집기를 수행합니까?

10. *세 값의 중앙값(median-of-three)* 휴리스틱은 배열의 첫 번째, 마지막, 가운데 원소를 살펴보고 이 세 원소의 중앙값을 퀵정렬의 피벗으로 사용한다는 점을 떠올려 보십시오. 세 값의 중앙값 휴리스틱을 사용하는 퀵정렬은 최악의 경우 크기 n인 배열을 정렬하는 데 $\Omega(n^2)$ 시간이 걸린다는 것을 증명하십시오. 구체적으로, 임의의 정수 n에 대해, 세 값의 중앙값 퀵정렬의 모든 재귀 호출에서 피벗이 항상 배열의 두 번째로 작은 원소가 되도록 하는 1부터 n까지의 정수의 순열을 기술하십시오. 이 순열을 설계하려면 PARTITION 서브루틴을 깊이 이해하고 있어야 합니다.

[15] n개의 팬케이크(탄 것이든 타지 않은 것이든)를 정렬하는 데 최악의 경우 필요한 최적 뒤집기 횟수의 *정확한* 값은 오랫동안 풀리지 않은 미해결 문제입니다. 그저 할 수 있는 만큼 최선을 다하십시오.

(a) 준비 운동 삼아 PARTITION 서브루틴이 *안정적*이라고 가정하십시오. 즉, 이 서브루틴은 피벗보다 작은 모든 원소의 기존 순서를 보존하고, 피벗보다 작은 모든 원소의 기존 순서를 보존합니다.

♥(b) PARTITION 서브루틴이 29쪽에 제시된 특정 알고리즘을 사용한다고 가정하십시오. 이 알고리즘은 안정적이지 *않습니다*.

11. (a) 이봐요, 모! 이봐요, 래리! 다음 알고리즘이 실제로 입력을 정렬한다는 것을 증명하십시오!

```
STOOGESORT(A[0..n−1]):
  if n = 2 and A[0] > A[1]
      swap A[0] ↔ A[1]
  else if n > 2
      m = ⌈2n/3⌉
      STOOGESORT(A[0..m−1])
      STOOGESORT(A[n−m..n−1])
      STOOGESORT(A[0..m−1])
```

(b) $m = \lceil 2n/3 \rceil$을 $m = \lfloor 2n/3 \rfloor$으로 바꾸어도 STOOGESORT가 여전히 올바르게 정렬하겠습니까? 답의 근거를 제시하십시오.

(c) STOOGESORT가 수행하는 비교 횟수에 대한 점화식을 (기저 사례를 포함하여) 제시하십시오.

(d) 점화식을 풀고, 여러분의 해가 옳음을 증명하십시오. *[힌트: 천장 함수는 무시하십시오.]*

(e) STOOGESORT가 실행하는 *교환*의 횟수가 최대 $\binom{n}{2}$임을 증명하십시오.

12. 다음의 잔혹하고 기이한 정렬 알고리즘은 게리 밀러(Gary Miller)가 제안했습니다:

```
CRUEL(A[1..n]):
  if n > 1
      CRUEL(A[1..n/2])
      CRUEL(A[n/2+1..n])
      UNUSUAL(A[1..n])
```

```
UNUSUAL(A[1..n]):
  if n = 2
      if A[1] > A[2]                          《유일한 비교입니다!》
          swap A[1] ↔ A[2]
  else
      for i ← 1 to n/4          《두 번째와 세 번째 4분의 1 구간을 맞바꿉니다.》
          swap A[i + n/4] ↔ A[i + n/2]
      UNUSUAL(A[1..n/2])              《왼쪽 절반에 대해 재귀 호출합니다》
      UNUSUAL(A[n/2+1..n])            《오른쪽 절반에 대해 재귀 호출합니다》
      UNUSUAL(A[n/4+1..3n/4])         《가운데 절반에 대해 재귀 호출합니다》
```

이 알고리즘이 수행하는 비교는 입력 배열의 값에 전혀 의존하지 않습니다. 이러한 정렬 알고리즘을 ***망각형(oblivious)***이라고 합니다. 이 문제에서는 입력 크기 n이 항상 2의 거듭제곱이라고 가정하십시오.

(a) CRUEL이 임의의 입력 배열을 올바르게 정렬함을 귀납법으로 증명하십시오. *[힌트: 1이 $n/4$개, 2가 $n/4$개, 3이 $n/4$개, 4가 $n/4$개 들어 있는 배열을 생각해 보십시오. 이 특수한 경우만으로 충분한 이유는 무엇입니까?]*

(b) UNUSUAL에서 for 루프를 제거하면 CRUEL이 올바르게 정렬하지 *못한다*는 것을 증명하십시오.

(c) UNUSUAL의 마지막 두 줄을 서로 바꾸면 CRUEL이 올바르게 정렬하지 *못한다*는 것을 증명하십시오.

(d) UNUSUAL의 실행 시간은 얼마입니까? 답의 근거를 제시하십시오.

(e) CRUEL의 실행 시간은 얼마입니까? 답의 근거를 제시하십시오.

13. 배열 $A[1..n]$에서 ***역전***이란 $i < j$이고 $A[i] > A[j]$를 만족하는 인덱스 쌍 (i, j)입니다. 원소가 n개인 배열의 역전 개수는 0(배열이 정렬되어 있는 경우)과 $\binom{n}{2}$(배열이 역순으로 정렬되어 있는 경우) 사이입니다. 원소가 n개인 배열의 역전 개수를 $O(n \log n)$ 시간에 세는 알고리즘을 설명하고 분석하십시오. *[힌트: 병합 정렬을 수정하십시오.]*

14. (a) n개의 점으로 이루어진 두 집합이 주어졌다고 가정합시다. 한 집합 $\{p_1, p_2, \ldots, p_n\}$은 직선 $y = 0$ 위에 있고, 다른 집합 $\{q_1, q_2, \ldots, q_n\}$은 직선 $y = 1$ 위에 있습니다. 각 점 p_i를 대응하는 점 q_i와 연결하여 n개의 선분으로 이루어진 집합을 만듭니다. 이 선분들 가운데 서로 교차하는 쌍이 몇 개인지를 $O(n \log n)$ 시간에 알아내는 분할 정복 알고리즘을 설명하고 분석하십시오. *[힌트: 앞 문제를 참고하십시오.]*

(b) 이제 *단위원 위에 있는* n개의 점으로 이루어진 두 집합 $\{p_1, p_2, \ldots, p_n\}$과 $\{q_1, q_2, \ldots, q_n\}$이 주어졌다고 가정합시다. 각 점 p_i를 대응하는 점 q_i와 연결합니다. 이 선분들 가운데 서로 교차하는 쌍이 몇 개인지 $O(n \log^2 n)$ 시간에 구하는 분할 정복 알고리즘을 설명하고 분석하십시오. *[힌트: (a)번의 풀이를 활용하십시오.]*

♥(c) (b)를 $O(n \log n)$ 시간에 해결하는 알고리즘을 설명하십시오. *[힌트: (b)에서 구한 풀이를 활용하십시오!]*

15. (a) 입력으로 0과 $n - \sqrt{n}$ 사이의 임의의 정수 k가 주어지면 부분 배열 $A[k + 1..k + \sqrt{n}]$을 제자리에서 정렬하는 서브루틴 SQRTSORT(k)가 있습니다. 이 서브루틴을 호출하여 입력 배열 $A[1..n]$을 정렬하는 알고리즘을 설명하십시오. (문제를 단순화하기 위해 $\sqrt{n}$은 정수라고 가정합니다.) 여러분의 알고리즘은 ***오직*** SQRTSORT를 호출하는 방법으로만 입력 배열을 조사하거나 수정할 수 있습니다. 특히 여러분의 알고리즘은 배열 원소를 직접 비교하거나, 이동하거나, 복사해서는 안 됩니다. 최악의 경우 여러분의 알고리즘은 SQRTSORT를 몇 번 호출합니까?

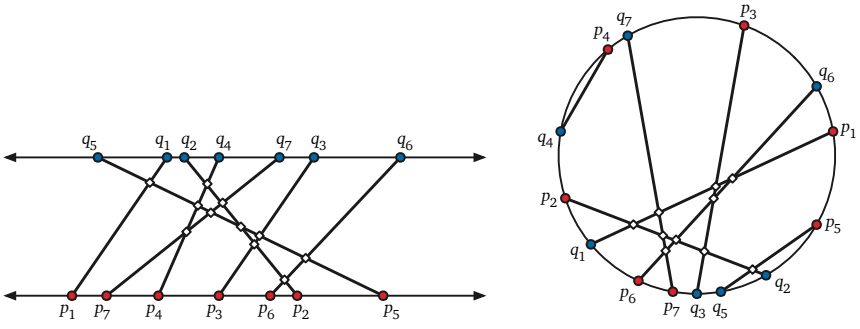


그림 1.20. 끝점이 평행선 위에 있는 선분들 중 서로 교차하는 쌍 열한 개와, 끝점이 원 위에 있는 선분들 중 서로 교차하는 쌍 열 개입니다.

♣(b) (a)에서 만든 알고리즘이 상수 배의 차이를 제외하면 최적임을 증명하십시오. 다시 말해, 여러분의 알고리즘이 SqrtSort를 호출하는 횟수를 $f(n)$이라고 할 때, 어떤 알고리즘도 SqrtSort를 $o(f(n))$번만 호출해서는 정렬할 수 없음을 증명하십시오.

(c) 이제 SqrtSort가 (a)에서 만든 정렬 알고리즘을 호출하는 방식으로 재귀적으로 구현되었다고 가정합시다. 예를 들어 재귀의 두 번째 단계에서 이 알고리즘은 대략 $n^{1/4}$ 크기의 배열들을 정렬합니다. 이렇게 만들어진 정렬 알고리즘의 최악의 경우 실행 시간은 얼마입니까? (분석을 간단히 하기 위해, 배열 크기 n이 2^{2^k} 꼴이어서 제곱근을 반복해서 취해도 항상 정수가 된다고 가정하십시오.)

선택

16. n개의 항목으로 이루어진 집합 S가 주어지고, 각 항목에는 *가치*와 *무게*가 있다고 가정합시다. 임의의 원소 $x \in S$에 대해 다음 두 부분집합을 정의합니다.

- $S_{<x}$는 S의 원소 중 값이 x의 값보다 작은 원소들의 집합입니다.
- $S_{>x}$는 S의 원소 중 값이 x의 값보다 큰 원소들의 집합입니다.

임의의 부분집합 $R \subseteq S$에 대해, $w(R)$이 R에 속한 원소들의 가중치 합을 나타낸다고 합시다. R의 ***가중 중앙값***은 $w(S_{<x}) \le w(S)/2$이고 $w(S_{>x}) \le w(S)/2$를 만족하는 임의의 원소 x입니다.

주어진 가중치 집합의 가중 중앙값을 $O(n)$ 시간에 계산하는 알고리즘을 설명하고 분석하십시오. 입력은 정렬되지 않은 두 배열 $S[1..n]$과 $W[1..n]$으로 이루어지며, 각 인덱스 i에 대해 i번째 원소의 값은 $S[i]$이고 가중치는 $W[i]$입니다. 모든 값은 서로 다르고 모든 가중치는 양수라고 가정해도 됩니다.

17. (a) 임의의 배열 $A[1..n]$에 어떤 값이 $n/4$개보다 많이 들어 있는지를 $O(n)$ 시간에 판별하는 알고리즘을 설명하십시오.

(b) 임의의 배열 $A[1..n]$과 정수 k가 주어졌을 때, A에 어떤 값이 k개보다 많이 들어 있는지 판별하는 알고리즘을 설명하고 분석하십시오. 알고리즘의 실행 시간을 n과 k 모두에 대한 함수로 나타내십시오. **해싱이나 기수 정렬, 또는 입력값의 순서가 아니라 정확한 값 자체에 의존하는 그 밖의 어떤 방법도 사용하지 마십시오.**

18. 서로 다른 수로 이루어진 배열 $A[1..5]$의 중앙값을 최대 6번의 비교로 계산하는 알고리즘을 설명하십시오. 의사코드를 작성하는 대신 ***결정 트리***를 사용하여 알고리즘을 설명하십시오. 결정 트리란 각 내부 노드에 "$A[i] \gtrless A[j]$?" 형태의 비교가 들어 있고 각 리프에 배열의 인덱스가 들어 있는 이진 트리입니다.

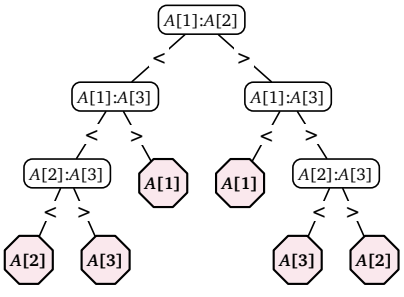


그림 1.21. 최대 3번의 비교로 원소 3개짜리 배열의 중앙값 찾기

19. 그림 1.22에 나타낸 Blum-Floyd-Pratt-Rivest-Tarjan MomSelect 알고리즘의 일반화를 생각해 봅시다. 이 알고리즘은 입력 배열을 크기가 5인 블록 $\lceil n/5 \rceil$개 대신 크기가 b인 블록 $\lceil n/b \rceil$개로 분할한다는 점만 다르고 나머지는 모두 동일합니다.

```
MOM_bSELECT(A[1..n], k):
  if n ≤ b^2
      use brute force
  else
      m ← ⌈n/b⌉
      for i ← 1 to m
          M[i] ← MEDIANOFB(A[b(i − 1) + 1..bi])
      mom_b ← MOM_bSELECT(M[1..m], ⌊m/2⌋)
      r ← PARTITION(A[1..n], mom_b)
      if k < r
          return MOM_bSELECT(A[1..r − 1], k)
      else if k > r
          return MOM_bSELECT(A[r + 1..n], k − r)
      else
          return mom_b
```

그림 1.22. 매개변수화된 선택 알고리즘 계열입니다. 문제 19를 참고하십시오.

(a) b가 상수라고 가정하고(따라서 서브루틴 MEDIANOFB는 $O(1)$ 시간에 실행됩니다), MOM$_b$SELECT의 실행 시간에 대한 점화식을 세우십시오. 특히, 재귀적 부분 문제들의 크기는 상수 b에 어떻게 의존합니까? 짝수 b와 홀수 b의 경우를 따로 고려하십시오.

(b) MOM$_1$SELECT의 최악의 경우 실행 시간은 얼마입니까? *[힌트: 이것은 함정 문제입니다.]*

♣♥(c) MOM$_2$SELECT의 최악의 경우 실행 시간은 얼마입니까? *[힌트: 이것은 불공평한 질문입니다!]*

♥(d) MOM$_3$SELECT의 최악의 경우 실행 시간은 얼마입니까? 실행 시간의 상한을 구하는 것은 간단하지만, 어려운 부분은 이 분석이 실제로 빈틈없다는 것, 즉 상한이 실제로 달성된다는 것을 보이는 것입니다. *[힌트: 문제 10을 참고하십시오.]*

♥(e) MOM$_4$SELECT의 최악의 경우 실행 시간은 얼마입니까? 이번에도 어려운 부분은 이 분석을 더 개선할 수 없음을 보이는 것입니다.[16]

(f) 임의의 상수 $b \geq 5$에 대해 알고리즘 MOM$_b$SELECT는 $O(n)$ 시간에 실행되지만, b 값에 따라 상수 배가 달라집니다. b개의 수의 중앙값을 찾는 데 필요한 최소 비교 횟수를 $M(b)$로 나타냅시다. $M(b)$의 정확한 값은 $b \leq 13$인 경우에만 알려져 있습니다:

b	1	2	3	4	5	6	7	8	9	10	11	12	13
$M(b)$	0	1	3	4	6	8	10	12	14	16	18	20	23

5 이상 13 이하의 각 b에 대해, 적당한 명시적 상수 α_b를 사용한 $T(n) \leq \alpha_b n$ 형태로 MOM$_b$SELECT의 실행 시간 상한을 구하십시오. (예를 들어, 39쪽에서 우리는 $\alpha_5 \leq 16$임을 보였습니다.)

(g) b가 어떤 값일 때 상수 α_b가 가장 작아집니까? *[힌트: 함정 문제입니다!]*

20. 주 피벗을 고르기 위해 작은 중앙값들로 이루어진 층을 하나 더 사용하는, 그림 1.23에 나타낸 Blum-Floyd-Pratt-Rivest-Tarjan SELECT 알고리즘의 변형이 $O(n)$ 시간에 실행됨을 증명하십시오.

21. (a) 두 개의 정렬된 배열 $A[1..n]$과 $B[1..n]$이 주어졌다고 가정합시다. A와 B의 합집합에서 중앙값 원소를 $\Theta(\log n)$ 시간에 찾는 알고리즘을 설명하십시오. 배열에 중복된 원소가 없다고 가정해도 됩니다.

(b) 두 정렬된 배열 $A[1..m]$과 $B[1..n]$, 그리고 정수 k가 주어졌다고 가정합시다. $A \cup B$에서 k번째로 작은 원소를 $\Theta(\log(m+n))$ 시간에 찾는 알고리즘을 설명하십시오. 예를 들어 $k = 1$이면 알고리즘은 $A \cup B$의 가장 작은 원소를 반환해야 합니다.) *[힌트: (a)의 해답을 활용하십시오.]*

[16] 네 원소의 중앙값은 두 번째로 작은 원소이거나 두 번째로 큰 원소입니다. 2014년에 Ke Chen과 Adrian Dumitrescu는 $k < n/2$일 때는 두 번째로 작은 원소를, $k > n/2$일 때는 두 번째로 큰 원소를 찾도록 MOM$_4$SELECT를 수정하면 그 결과로 얻는 알고리즘이 $O(n)$ 시간, 즉 선형 시간에 실행된다는 것을 증명했습니다! 자세한 내용은 그들의 논문 "Select with Groups of 3 or 4 Takes Linear Time" (WADS 2015, arXiv:1409.3600)을 참고하십시오.

```
MomomSelect(A[1..n], k):
  if n ≤ 81
      use brute force
  else
      m ← ⌈n/3⌉
      for i ← 1 to m
          M[i] ← MedianOf3(A[3i − 2..3i])
      mm ← ⌈m/3⌉
      for j ← 1 to mm
          Mom[j] ← MedianOf3(M[3j − 2..3j])
      momom ← MomomSelect(Mom[1..mm], ⌊mm/2⌋)
      r ← Partition(A[1..n], momom)
      if k < r
          return MomomSelect(A[1..r − 1], k)
      else if k > r
          return MomomSelect(A[r + 1..n], k − r)
      else
          return momom
```

그림 1.23. mom들의 중앙값을 이용한 선택; 문제 20을 참조하십시오).

♥(c) 이제 *세 개의* 정렬된 배열 $A[1..n]$, $B[1..n]$, $C[1..n]$과 정수 k가 주어졌다고 가정합시다. $A \cup B \cup C$에서 k번째로 작은 원소를 $O(\log n)$ 시간에 찾는 알고리즘을 설명하십시오.

(d) 마지막으로, 모든 행 $A[i, \cdot]$이 정렬된 2차원 배열 $A[1..m, 1..n]$과 정수 k가 주어졌다고 가정합시다. A에서 k번째로 작은 원소를 가능한 한 빠르게 찾는 알고리즘을 설명하십시오. 여러분의 알고리즘의 실행 시간은 m에 어떻게 의존합니까? *[힌트: 먼저 16번 문제를 푸십시오.]*

산술

22. 1854년에 고고학자들은 기원전 2000년경에 새겨진 수메르 점토판을 발견했습니다. 이 점토판에는 59까지의 정수의 제곱이 나열되어 있었습니다. 이 발견을 근거로 일부 학자들은 고대 수메르인들이 $x \cdot y = (x^2 + y^2 - (x - y)^2)/2$와 같은 항등식을 사용해 곱셈을 제곱으로 환원하여 계산했다고 추측했습니다. 안타깝게도 바로 그 학자들은 수메르인들이 더 큰 수를 어떻게 제곱했다는 것인지에 대해서는 아무 말도 하지 않습니다. 4천 년이 지난 지금, 우리는 마침내 재귀의 힘으로 이 수메르 수학자들을 고된 삶에서 구해 낼 수 있습니다!

(a) 세 개의 $\lceil n/2 \rceil$자리 수를 제곱하는 문제로 환원하여 임의의 n자리 수를 $O(n^{\lg 3})$ 시간에 제곱하는 카라추바 알고리즘의 변형을 설명하십시오. (실제로 카라추바는 1960년에 이 방법을 고안했습니다.)

(b) 여섯 개의 $\lceil n/3 \rceil$자리 수를 제곱하는 문제로 환원하여, 임의의 n자리 수를 $O(n^{\log_3 6})$ 시간에 제곱하는 재귀 알고리즘을 설명하십시오.

♥(c) 임의의 n자리 수를 제곱하는 문제를 *다섯* 개의 $(n/3 + O(1))$자리 수를 제곱하는 문제로만 환원하여, $O(n^{\log_3 5})$ 시간에 제곱하는 재귀 알고리즘을 설명하십시오. *[힌트: $(a+b+c)^2 + (a-b+c)^2$은 무엇입니까?]*

23. (a) $n \geq m$을 만족하는 임의의 n과 m에 대해, 임의의 m자리 수와 임의의 n자리 수를 $O(nm^{\lg 3 - 1})$ 시간에 곱하는 카라추바 알고리즘의 변형을 설명하고 분석하십시오.

(b) 문항 (a)의 알고리즘을 서브루틴으로 사용하여 2^n의 십진 표현을 $O(n^{\lg 3})$ 시간에 계산하는 알고리즘을 설명하십시오. (한 번에 한 자리씩 계산하는 표준 알고리즘은 $\Theta(n^2)$ 시간이 걸립니다.)

(c) 임의의 n비트 이진수의 십진 표현을 $O(n^{\lg 3})$ 시간에 계산하는 분할 정복 알고리즘을 설명하십시오. *[힌트: 실행 시간에 로그 인자가 추가로 붙지 않도록 주의하십시오.]*

♥(d) 두 n자리 수를 $O(M(n))$ 시간에 곱할 수 있다고 가정합시다. 임의의 n비트 이진수의 십진 표현을 $O(M(n) \log n)$ 시간에 계산하는 알고리즘을 설명하십시오. *[힌트: 어려운 부분은 분석입니다. 정의역 변환을 이용하십시오.]*

24. 음이 아닌 정수 n의 팩토리얼 $n!$을 계산하는 다음의 고전적인 재귀 알고리즘을 살펴봅시다:

```
FACTORIAL(n):
  if n = 0
      return 1
  else
      return n · FACTORIAL(n − 1)
```

(a) 이 알고리즘은 곱셈을 몇 번 수행합니까?

(b) $n!$을 이진수로 쓰려면 몇 비트가 필요합니까? 잘 알려진 함수 $f(n)$을 사용하여 답을 $\Theta(f(n))$ 꼴로 표현하십시오. *[힌트: $(n/2)^{n/2} < n! < n^n$.]*

(c) (b)에 대한 답을 통해 곱셈 횟수가 FACTORIAL의 실제 실행 시간을 잘 추정하는 척도가 *아니라는* 것을 확신하게 될 것입니다. 격자 알고리즘이나 배가와 반분(duplation and mediation) 중 어느 방법을 사용하든, 임의의 k자리 수와 임의의 l자리 수를 $O(k \cdot l)$ 시간에 곱할 수 있습니다. 이 곱셈 알고리즘을 서브루틴으로 사용한다면 FACTORIAL의 실행 시간은 얼마입니까?

(d) 다음 재귀 알고리즘도 팩토리얼 함수를 계산하지만, 곱셈을 다른 방식으로 묶어 계산합니다:

```
FALLING(n, m):                                  《n!/(n − m)!을 계산하십시오.》
  if m = 0
      return 1
  else if m = 1
      return n
  else
      return FALLING(n, ⌊m/2⌋) · FALLING(n − ⌊m/2⌋, ⌈m/2⌉)
```

초등학교에서 배우는 곱셈 방법을 사용한다면 FALLING(n, n)의 실행 시간은 얼마입니까? *[힌트: 늘 그렇듯이 바닥 함수와 천장 함수는 무시하십시오.]*

(e) 임의의 $k \geq l$에 대해 임의의 k자리 수와 임의의 l자리 수를 $O(k \cdot l^{\lg 3-1}) = O(k \cdot l^{0.585})$ 시간에 곱하는 카라추바 알고리즘의 변형을 설명하고 분석하십시오.

♥(f) (e)에서 수정한 카라추바 곱셈을 사용하면 FACTORIAL(n)과 FALLING(n, n)의 실행 시간은 각각 얼마입니까?

25. 두 양의 정수 x와 y의 ***최대공약수***는 $\gcd(x, y)$로 표기하며, x/d와 y/d가 모두 정수가 되는 가장 큰 정수 d입니다. 기원전 300년경에 쓰인 유클리드의 *원론*은 $\gcd(x, y)$를 계산하는 다음과 같은 재귀 알고리즘을 설명합니다:[17]

```
EUCLIDGCD(x, y):
  if x = y
      return x
  else if x > y
      return EUCLIDGCD(x − y, y)
  else
      return EUCLIDGCD(x, y − x)
```

(a) EUCLIDGCD가 $\gcd(x, y)$를 올바르게 계산함을 증명하십시오.[18] 구체적으로는 다음과 같습니다:

i. EUCLIDGCD(x, y)가 x와 y를 모두 나눈다는 것을 증명하십시오.

ii. x와 y의 모든 공약수가 EUCLIDGCD(x, y)의 약수임을 증명하십시오.

(b) EUCLIDGCD(x, y)의 최악의 경우 실행 시간을 x와 y에 대한 함수로 나타내면 어떻게 됩니까? ($x - y$를 계산하는 데 $O(\log x + \log y)$ 시간이 걸린다고 가정하십시오.)

[17] 유클리드의 알고리즘은 때때로 가장 오래된 재귀 알고리즘, 심지어 가장 오래된 *자명하지 않은* 알고리즘으로 잘못 소개되기도 합니다. 그러나 자명하지 않으면서 재귀적이기도 한 이집트의 배가·반분 알고리즘이 유클리드보다 적어도 1500년 앞섭니다.

[18] 유클리드는 이를 증명하지 않았습니다. *원론* 제7권의 명제 1은 EUCLIDGCD$(x, y) = 1$이면 x와 y가 서로소(즉, $\gcd(x, y) = 1$)라고 서술하지만, 그 증명은 $x \bmod (y \bmod (x \bmod y)) = 1$인 특수한 경우만 다룹니다. 명제 2는 x와 y가 서로소가 *아니면* EUCLIDGCD$(x, y) = \gcd(x, y)$라고 서술하지만, 그 증명은 $\gcd(x, y) = y$인 경우와 $\gcd(x, y) = y \bmod (x \bmod y)$인 경우라는 특수한 경우만 다룹니다. 끝으로, 이 두 명제를 합쳐도 EUCLIDGCD가 올바르다는 완전한 증명이 되지 않습니다. 유클리드처럼 하지 마십시오.

(c) 다음 알고리즘 또한 $\gcd(x, y)$를 계산한다는 것을 증명하십시오:

```
FastEuclidGCD(x, y):
  if y = 0
      return x
  else if x > y
      return FastEuclidGCD(y, x mod y)
  else
      return FastEuclidGCD(x, y mod x)
```

(d) FastEuclidGCD(x, y)의 최악의 경우 실행 시간을 x와 y의 함수로 나타내면 무엇입니까? (x mod y를 계산하는 데 $O(\log x \cdot \log y)$ 시간이 걸린다고 가정하십시오.)

(e) 다음 알고리즘도 $\gcd(x, y)$를 계산함을 증명하십시오:

```
BinaryGCD(x, y):
  if x = y
      return x
  else if x and y are both even
      return 2 · BinaryGCD(x/2, y/2)
  else if x is even
      return BinaryGCD(x/2, y)
  else if y is even
      return BinaryGCD(x, y/2)
  else if x > y
      return BinaryGCD((x − y)/2, y)
  else
      return BinaryGCD(x, (y − x)/2)
```

(f) x와 y의 함수로 나타낼 때, BinaryGCD(x, y)의 최악의 경우 실행 시간은 얼마입니까? ($x - y$를 계산하는 데 $O(\log x + \log y)$ 시간이 걸리고, $z/2$를 계산하는 데 $O(\log z)$ 시간이 걸린다고 가정하십시오.)

배열

26. 임의로 선택한 칸 하나가 제거된 $2^n \times 2^n$ 체커판이 주어졌다고 합시다. 각각 3개의 칸으로 이루어진 **L**자 모양 타일로 이 판을 빈틈이나 겹침 없이 채우는 타일링을 계산하는 알고리즘을 설명하고 분석하십시오. 입력은 정수 n과, 빠진 칸의 행과 열을 나타내는 두 개의 n비트 정수입니다. 출력은 $(4^n - 1)/3$개 타일의 위치와 방향으로 이루어진 목록입니다. 알고리즘은 $O(4^n)$ 시간에 실행되어야 합니다. *[힌트: 먼저 이러한 타일링이 항상 존재함을 증명하십시오.]*

27. 여러분은 n명의 대의원이 참석한 정당 전당대회(혹은 교수 회의일 수도 있습니다)에 온 방문객이며, 각 대의원은 정확히 하나의 정당에 소속되어 있습니다. 어떤 대의원이 어느 정당에 속하는지는 알아낼 방법이 없습니다. 특히 그것을 물어보면

여러분은 즉시 전당대회에서 쫓겨나게 됩니다. 하지만 두 대의원을 서로 소개해 주면 그 두 사람이 *같은* 정당에 속하는지는 알 수 있습니다. 같은 정당의 당원들은 항상 미소를 지으며 다정하게 악수를 나누고, 다른 정당의 당원들은 항상 성난 눈빛으로 노려보며 서로 모욕을 주고받습니다.[19]

(a) 대의원의 과반수가 같은 정당에 속해 있다고 가정합시다. 이 다수당에 속한 대의원을 모두 찾아내는 효율적인 알고리즘을 설명하십시오.

(b) 이제 정당이 셋 이상이지만 한 정당이 *상대 다수*를 차지한다고 가정합시다. 즉, 그 정당에 속한 사람이 다른 어느 정당에 속한 사람보다도 많습니다. 상대 다수당이 적어도 p명으로 이루어져 있다고 가정하고, 상대 다수당에 속한 사람들을 가능한 한 알뜰하게 정확히 골라내는 실용적인 절차를 제시하십시오. 제발 부탁드립니다.

28. 스멀리언 섬에는 세 종류의 주민이 있습니다. *기사*는 항상 진실을 말하고, *건달*은 항상 거짓말을 하며, *보통 사람*은 진실을 말할 때도 있고 그렇지 않을 때도 있습니다. 섬의 모든 주민은 다른 모든 주민의 이름과 유형(기사, 건달, 보통 사람)을 알고 있습니다. 여러분은 모든 주민의 유형을 알아내고자 합니다.

어떤 주민에게든 다른 주민의 유형을 알려 달라고 요청할 수 있습니다. 구체적으로, "*X* 씨, *Y*의 유형은 무엇입니까?" 라고 물으면 *X*는 다음과 같이 대답합니다.

- *X*가 기사라면, *X*는 *Y*의 올바른 유형을 답할 것입니다.
- *X*가 악당이라면, *X*는 *Y*가 *아닌* 두 유형 중 *어느* 쪽으로든 대답할 수 있습니다.
- *X*가 보통 사람이라면, *X*는 세 가지 유형 중 *어느* 것으로든 대답할 수 있습니다.

주민들은 정확히 이 형식을 따르지 않는 질문은 모두 무시합니다. 특히 주민에게 그 자신의 유형을 물어볼 수는 없습니다. 같은 주민에게 같은 질문을 여러 번 해도 항상 같은 답이 돌아오므로, 어떤 질문이든 두 번 이상 할 필요는 없습니다.

(a) 주민의 엄밀한 과반수가 기사라는 사실을 안다고 가정합시다. 모든 주민의 유형을 판별하는 효율적인 알고리즘을 설명하십시오.

(b) 주민 중 기사가 절반 이하라면 모든 주민의 유형을 판별할 수 없음을 증명하십시오.

29. 대부분의 그래픽 하드웨어는 *blit*, 즉 **bl**ock **t**ransfer라는 저수준 연산을 지원합니다. 이 연산은 픽셀 맵(픽셀 값의 2차원 배열)의 직사각형 조각을 한 위치에서 다른 위치로 빠르게 복사합니다. 이는 표준 C 라이브러리 함수 `memcpy()`의 2차원 버전입니다.

$n \times n$ 픽셀 맵을 시계 방향으로 90° 회전시키려 한다고 가정합시다. 적어도 n이 2의 거듭제곱일 때 쓸 수 있는 한 가지 방법은 픽셀 맵을 $n/2 \times n/2$ 크기의 블록 네 개로 나누고, 다섯 번의 블릿(blit)을 차례로 수행하여 각 블록을 올바른 위치로

[19] 현실의 정치는 이 단순화된 모형보다 훨씬 더 복잡하지만, 이 책은 이론서입니다!

옮긴 다음, 각 블록을 재귀적으로 회전시키는 것입니다. (왜 다섯 번일까요? 하노이의 탑 퍼즐에 세 번째 기둥이 필요한 것과 같은 이유입니다.) 또는 블록들을 *먼저* 재귀적으로 회전시킨 *다음* 블릿으로 제자리에 옮길 수도 있습니다.

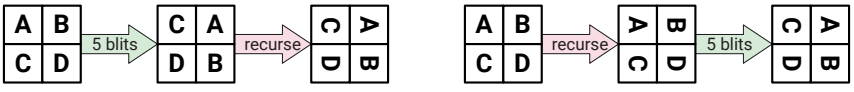


그림 1.24. 픽셀 맵을 회전하는 두 가지 알고리즘.

(a) n이 2의 거듭제곱일 때 알고리즘의 두 버전이 모두 올바르다는 것을 증명하십시오.

(b) n이 2의 거듭제곱일 때 이 알고리즘은 블릿을 *정확히* 몇 번 수행합니까?

(c) 이 알고리즘이 2의 거듭제곱일 때뿐만 아니라 임의의 n에 대해서도 동작하도록 수정하는 방법을 설명하십시오. 수정한 알고리즘은 블릿을 몇 번 수행합니까?

(d) $k \times k$ 블릿(blit)에 $O(k^2)$ 시간이 걸린다면, 여러분 알고리즘의 실행 시간은 얼마입니까?

(e) $k \times k$ 블릿에 $O(k)$ 시간만 걸린다면 어떻게 되겠습니까?

30. n개의 서로 다른 수로 이루어진 배열 $A[0..n-1]$이 ***바이토닉***(bitonic)이라는 것은, 유일한 인덱스 i와 j가 존재하여 $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$이고 $A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$입니다. 다시 말해, 바이토닉

그림 1.25(저자의 초상화)는 생략되었습니다. 원서 61쪽을 참고하십시오.

그림 1.25. 첫 번째 회전 알고리즘(블릿 후 재귀)이 동작하는 모습입니다. (책 끝의 이미지 출처를 참조하십시오.)

수열은 증가하는 수열 뒤에 감소하는 수열이 이어지는 형태이거나, 순환 이동하면 그러한 형태가 되는 수열입니다. 예를 들어,

배열 | 4 | 6 | 9 | 8 | 7 | 5 | 1 | 2 | 3 | 은 바이토닉이지만,

배열 | 3 | 6 | 9 | 8 | 7 | 5 | 1 | 2 | 4 | 은 바이토닉이 *아닙니다*.

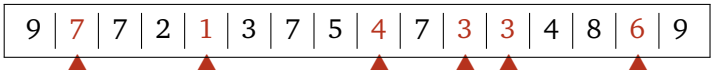
n개의 원소로 이루어진 바이토닉 배열에서 *가장 작은* 원소를 $O(\log n)$ 시간에 찾는 알고리즘을 설명하고 분석하십시오. 입력 배열의 수들은 모두 서로 다르다고 가정해도 됩니다.

31. 서로 다른 정수 n개로 이루어진 배열 $A[1..n]$이 주어졌다고 가정합시다. 각 정수는 양수, 음수 또는 0일 수 있으며, 배열은 $A[1] < A[2] < \cdots < A[n]$이 되도록 오름차순으로 정렬되어 있습니다.

(a) $A[i] = i$를 만족하는 인덱스 i를 계산하거나, 그러한 인덱스가 존재하지 않음을 올바르게 보고하는 빠른 알고리즘을 설명하십시오.

(b) $A[1] > 0$임을 미리 알고 있다고 가정합시다. $A[i] = i$를 만족하는 인덱스 i를 계산하거나, 그러한 인덱스가 존재하지 않음을 올바르게 보고하는 훨씬 더 빠른 알고리즘을 설명하십시오. *[힌트: 이것은* ***정말로*** *쉽습니다.]*

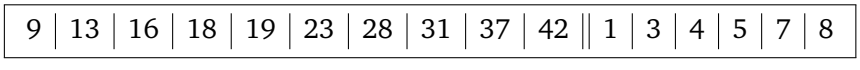
32. $A[1] \geq A[2]$이고 $A[n-1] \leq A[n]$이라는 특별한 성질을 가진 배열 $A[1..n]$이 주어졌다고 가정합시다. 원소 $A[x]$가 양쪽 이웃보다 작거나 같으면, 더 형식적으로 말해 $A[x-1] \geq A[x]$이고 $A[x] \leq A[x+1]$이면, $A[x]$를 *극솟값*이라고 합니다. 예를 들어, 다음 배열에는 극솟값이 여섯 개 있습니다:



배열 전체를 훑으면 $O(n)$ 시간에 국소 최솟값을 찾을 수 있다는 것은 자명합니다. $O(\log n)$ 시간에 국소 최솟값을 찾는 알고리즘을 설명하고 분석하십시오. *[힌트: 주어진 경계 조건에서는 배열에 국소 최솟값이* ***반드시*** *하나 이상 존재합니다. 그 이유는 무엇입니까?]*

33. 1과 $n-1$ 사이의 어떤 ***알려지지 않은*** 정수 k에 대해, 서로 다른 수 n개로 이루어진 정렬된 배열이 k칸만큼 *회전*된 채로 주어졌다고 가정합시다. 즉, 어떤 접두사 $A[1..k]$는 오름차순으로 정렬되어 있고, 그에 대응하는 접미사 $A[k+1..n]$도 오름차순으로 정렬되어 있으며, $A[n] < A[1]$을 만족하는 배열 $A[1..n]$이 주어집니다.

예를 들어, 다음과 같은 16개의 원소로 이루어진 배열이 주어질 수 있습니다(여기서 $k = 10$입니다).



(a) 미지의 정수 k를 계산하는 알고리즘을 설명하고 분석하십시오.

(b) 주어진 배열에 주어진 수 x가 들어 있는지 판별하는 알고리즘을 기술하고 분석하십시오.

34. 액션 블록버스터 *패스트 앤 임파서블 XIII¾: 소모품 정의의 마지막 수호자들 리로디드*의 2막 마지막에서, 악당 메타포 박사는 히어로 리그/포스/스쿼드 전원에게 최면을 걸어 절벽 가장자리를 따라 한 줄로 길게 세운 뒤, 미리 정해 둔 신호가 떨어지면 각 영웅이 자신의 왼쪽과 오른쪽에서 자신보다 키가 큰 가장 가까운 영웅을 쏘도록 지시합니다.

모든 영웅 n명의 키가 왼쪽부터 오른쪽 순서대로 배열 $Ht[1..n]$에 주어진다고 가정합시다. (급여 분쟁을 피하기 위해 제작자들은 어떤 두 영웅도 키가 같지 않아야 한다고 고집했습니다.) 그러면 다음의 무차별 대입 알고리즘을 사용하여 각 영웅의 왼쪽 대상과 오른쪽 대상을 $O(n^2)$ 시간에 계산할 수 있습니다.

```
WhoTargetsWhom(Ht[1..n]):
  for j ← 1 to n
                    《영웅 j의 왼쪽 대상 L[j]를 찾습니다》
      L[j] ← None
      for i ← 1 to j − 1
          if Ht[i] > Ht[j]
              L[j] ← i
          《영웅 j에 대한 오른쪽 대상 R[j]를 찾습니다》
      R[j] ← None
      for k ← n down to j + 1
          if Ht[k] > Ht[j]
              R[j] ← k
  return L[1..n], R[1..n]
```

(a) WhoTargetsWhom의 출력을 $O(n \log n)$ 시간에 계산하는 분할 정복 알고리즘을 설명하십시오.

(b) n명의 영웅 중 적어도 $\lfloor n/2 \rfloor$명이 대상임을 증명하십시오. 즉, 출력 배열 $R[0..n-1]$과 $L[0..n-1]$에 (None을 제외하고) 서로 다른 값이 적어도 $\lfloor n/2 \rfloor$개 들어 있음을 증명하십시오.

(c) 아아, 메타포 박사의 사악한 계획은 성공합니다. 미리 정해 둔 신호가 떨어지자 모든 영웅이 동시에 각자의 대상을 쏘고, 모든 대상은 절벽 아래로 떨어져 죽은 것처럼 보입니다. 메타포는 이 비열한 실험을 몇 번이고 되풀이합니다. 학살이 끝날 때마다 그는 남은 영웅들에게 같은 알고리즘에 따라 새 대상을 고르게 한 뒤, 다음 신호에 맞춰 그 대상을 쏘게 합니다. 결국 영웅 크루/동맹/패거리 가운데 키가 가장 작은 단 한 명만 살아남습니다.[20]

메타포 박사의 치명적인 과정이 마침내 끝나기까지 몇 라운드가 걸리는지 계산하는 알고리즘을 설명하고 분석하십시오. 만점을 받으려면 알고리즘이 $O(n)$ 시간에 실행되어야 합니다.

35. 여러분은 인기 게임 쇼 "이웃을 이겨라!"의 참가자입니다. 여러분 앞에는 상자들이 $m \times n$ 격자 모양으로 놓여 있고, 상자마다 서로 다른 수가 하나씩 들어 있습니다. 상자 하나를 여는 데에는 $100가 듭니다. 여러분의 목표는 격자에서 이웃한 상자들(위, 아래, 왼쪽, 오른쪽)보다 큰 수가 들어 있는 상자를 찾는 것입니다. 다른 어떤 상대보다도 적은 돈을 쓰면, 라스베이거스 2인 일주일 여행권과 Rice-A-Roni™(여러분이 헤어날 수 없을 만큼 중독된 바로 그 식품) 1년 치를 상품으로 받습니다.

(a) $m = 1$이라고 가정합시다. 양쪽 이웃보다 모두 큰 수를 찾는 알고리즘을 설명하십시오. 최악의 경우 여러분의 알고리즘은 상자를 몇 개 엽니까?

♥(b) $m = n$이라고 가정합시다. 이웃한 모든 수보다 큰 수를 하나 찾는 알고리즘을 설명하십시오. 최악의 경우 여러분의 알고리즘은 상자를 몇 개 엽니까?

[20] 손에 땀을 쥐게 하는 마지막 막에서, 영웅 팀/그룹/협회의 마지막 생존자인 다람쥐 레트콘이 시간을 거슬러 올라가 나머지 $n-1$명의 영웅을 실물과 똑같은 풍선 조각상으로 소급하여 바꿔치기함으로써 모두를 구합니다. 그러니까, 네, 사실상 *어벤져스: 엔드게임*입니다.

♣♥(c) (b)에서 제시한 해법이 상수 배 이내에서 최적임을 증명하십시오.

36. (a) 어떤 양의 정수 ℓ에 대해 $n = 2^\ell - 1$이라고 합시다. 누군가가 *서로 다른* ℓ비트 문자열로 이루어진 정렬되지 않은 배열 $A[1..n]$을 가지고 있다고 주장한다고 가정합시다. 따라서 A에 나타나지 *않는* ℓ비트 문자열은 정확히 하나입니다. 나아가 A에 접근하는 ***유일한*** 방법은 함수 FetchBit(i, j)를 호출하는 것뿐이라고 가정합시다. 이 함수는 문자열 $A[i]$의 j번째 비트를 $O(1)$ 시간에 반환합니다. FetchBit을 $O(n)$번만 호출하여 A에서 빠진 문자열을 찾는 알고리즘을 설명하십시오.

♥(b) 이제 어떤 양의 정수 k와 ℓ에 대해 $n = 2^\ell - k$라고 가정하고, 이번에도 *서로 다른* ℓ비트 문자열들로 이루어진 배열 $A[1..n]$이 주어진다고 합시다. FetchBit를 $O(n \log k)$번만 호출하여 A에서 빠진 k개의 문자열을 찾는 알고리즘을 설명하십시오.

트리

37. 이 문제에서 이진 트리의 *부분 트리*란 연결된 임의의 부분 그래프를 의미합니다. 모든 내부 노드가 두 개의 자식을 가지고 모든 리프의 깊이가 정확히 같으면, 그 이진 트리를 *완전*하다고 합니다. 주어진 이진 트리의 *가장 큰 완전 부분 트리*를 계산하는 재귀 알고리즘을 설명하고 분석하십시오. 알고리즘은 이 부분 트리의 루트와 깊이를 모두 반환해야 합니다. 예시는 그림 1.26을 참조하십시오.

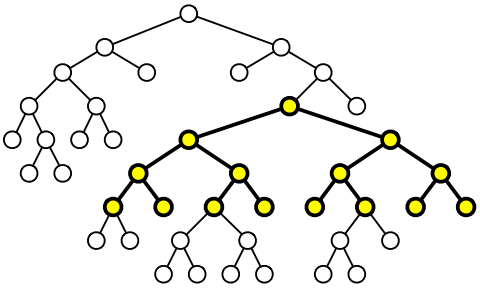


그림 1.26. 이 이진 트리에서 가장 큰 완전 부분 트리의 깊이는 3입니다.

38. T를 정점이 n개인 이진 트리라고 합시다. 임의의 정점 v를 삭제하면 T는 최대 세 개의 부분 트리로 나뉘며, 이 부분 트리들은 각각 v의 왼쪽 자식(있는 경우), v의 오른쪽 자식(있는 경우), v의 부모(있는 경우)를 포함합니다. 이렇게 나뉜 더 작은 트리들이 각각 최대 $n/2$개의 정점을 가지면 v를 ***중심*** 정점이라고 부릅니다. 예시는 그림 1.27을 참조하십시오.

임의로 주어진 이진 트리에서 중심 정점을 찾는 알고리즘을 설명하고 분석하십시오. *[힌트: 먼저 모든 트리에 중심 정점이 존재함을 증명하십시오.]*

39. (a) George O'Jungle 교수는 노드가 27개인 이진 트리를 가지고 있으며, 이 트리의 각 노드에는 로마자 알파벳 문자 또는 문자 **&** 중 하나가 서로 겹치지

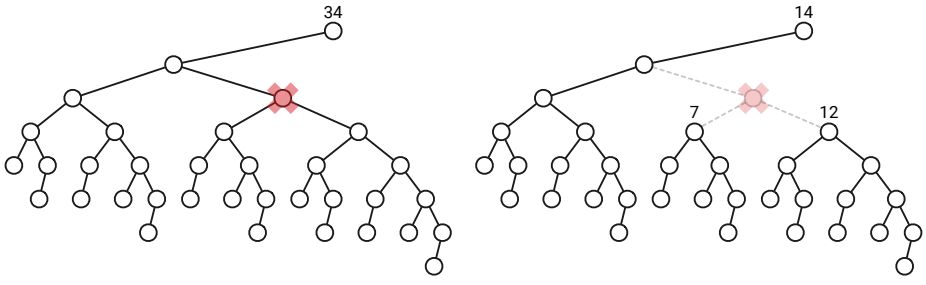


그림 1.27. 노드 34개로 이루어진 이진 트리에서 중심 정점을 삭제하면 노드가 각각 14개, 7개, 12개인 부분 트리가 남습니다.

않게 레이블로 붙어 있습니다. 이 트리를 전위 순회와 후위 순회로 탐색하면 노드를 다음 순서로 방문합니다:

- 전위 순회: `I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X`
- 후위 순회: `H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I`

George의 이진 트리를 그리십시오.

(b) 리프가 아닌 모든 노드의 자식이 정확히 두 개이면 그 이진 트리를 *정 이진 트리*(full binary tree)라고 부른다는 것을 상기하십시오.

i. 임의의 *정*(full) 이진 트리의 전위 순회 노드 순서와 후위 순회 노드 순서가 입력으로 주어졌을 때, 그 트리를 재구성하는 재귀 알고리즘을 설명하고 분석하십시오.

ii. 전위 순회와 후위 순회의 노드 순서만으로 *임의의* 이진 트리를 재구성하는 알고리즘은 존재하지 않음을 증명하십시오.

(c) 전위 순회와 *중위 순회* 노드 순서가 입력으로 주어질 때, *임의의* 이진 트리를 재구성하는 재귀 알고리즘을 설명하고 분석하십시오.

(d) 전위 순회 노드 순서만 주어졌을 때, 임의의 이진 *탐색* 트리를 재구성하는 재귀 알고리즘을 서술하고 분석하십시오.

♥(e) 전위 순회 노드 순서만 주어졌을 때 임의의 이진 *탐색* 트리를 $O(n)$ ***시간 안에*** 재구성하는 재귀 알고리즘을 설명하고 분석하십시오. (b)–(e)에서는 모든 키가 서로 다르며 입력과 일치하는 이진 트리가 적어도 하나 존재한다고 가정하십시오.

40. 2차원 상자 안에 n개의 점이 흩어져 있다고 가정합시다. ***kd-트리***[21]는 다음과 같이 점들을 재귀적으로 분할합니다. 상자 내부에 점이 하나도 없으면 작업을 마칩니다.

[21] "kd-트리"("케이 *디* 트리"로 발음합니다)라는 용어는 원래 "k차원 트리"의 약어였지만, 오늘날에는 이 어원을 따지지 않고 사용합니다. 제정신인 사람이라면 *명백히* 더 나은 d를 두고 문자 k로 차원을 나타내는 일은 *절대* 없을 것이라는 점도 그 이유 중 하나입니다. 어원에 맞추려면 이 문제의 자료구조를 "2d-트리"(또는 "2-d 트리")라고 불러야 하겠지만, 지금은 "2차원 kd-트리"가 표준 명칭입니다. 다음도 참고하십시오:

그렇지 않으면 상자 안(경계 위가 *아닌* 곳)에 있는 중앙값 점을 지나는 *수직*선으로 상자를 더 작은 두 상자로 나누어, 점들을 최대한 고르게 분할합니다. 그런 다음 더 작은 두 상자를 *각각 90도 회전한 뒤에* 각 상자 안의 점들에 대해 kd-트리를 재귀적으로 구축합니다. 따라서 재귀의 각 단계마다 수직 분할과 수평 분할을 번갈아 수행합니다. 최종적으로 남는 빈 상자들을 *셀*이라고 부릅니다.

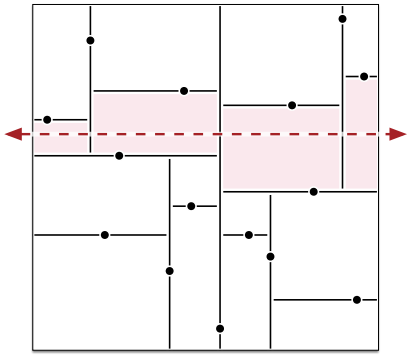


그림 1.28. 15개의 점에 대한 kd-트리입니다. 점선은 음영 처리된 네 개의 셀을 가로지릅니다.

(a) 셀의 개수를 n에 대한 함수로 나타내면 몇 개입니까? 답이 옳음을 증명하십시오.

(b) 최악의 경우, 수평선 하나가 가로지를 수 있는 칸의 개수는 n의 함수로 나타낼 때 *정확히* 몇 개입니까? 답이 옳다는 것을 증명하십시오. 어떤 정수 k에 대해 $n = 2^k - 1$이라고 가정하십시오. *[힌트: $f(16) = 4$를 만족하는 함수 f는 하나만 있는 것이 아닙니다.]*

(c) kd-트리에 저장된 n개의 점이 주어졌다고 가정합시다. (그림의 점선과 같은) 수평선보다 위에 있는 점의 개수를 가능한 한 빠르게 세는 알고리즘을 설명하고 분석하십시오. *[힌트: (b) 부분을 이용하십시오.]*

(d) n개의 점을 담은 kd-트리가 주어질 때, 변이 수평과 수직 방향인 직사각형 R 안에 놓인 점의 개수를 세는 효율적인 알고리즘을 기술하고 분석하십시오. *[힌트: (c) 문항을 이용하십시오.]*

41. CS 225를 새로 수강하는 학생인 밥 라텐버(Bob Ratenbur)는 이진 트리의 전위, 중위, 후위 순회를 수행하는 코드를 작성하려고 합니다. 밥은 순회 알고리즘의 기본 아이디어를 대충은 이해하고 있지만, 실제로 구현하려고 할 때마다 재귀 호출을 자꾸 뒤섞어 버립니다. 마감 5분 전, 밥은 다음과 같은 구조의 코드를 허둥지둥 제출합니다:

B-트리(아마도), 알파 셰이프, 베타 스켈레톤, 엡실론 넷, 포토맥강, 미시시피강, 미시간호, 타호호, 맨해튼섬, 라브레아 타르 피츠, 사하라 사막, 킬리만자로산, 남베트남, 동티모르, 은하수 은하, 타운스빌시, 그리고 자율주행 자동차.

```
PreOrder(v):
  if v = Null
      return
  else
      print label(v)
      ■Order(left(v))
      ■Order(right(v))
```

```
InOrder(v):
  if v = Null
      return
  else
      ■Order(left(v))
      print label(v)
      ■Order(right(v))
```

```
PostOrder(v):
  if v = Null
      return
  else
      ■Order(left(v))
      ■Order(right(v))
      print label(v)
```

이 의사코드의 각 ■는 접두사 Pre, In, Post 중 하나를 가리고 있습니다. 또한 다음 함수 호출은 각각 밥이 제출한 코드에 정확히 한 번씩 나타납니다:

PreOrder(*left*(*v*))	PreOrder(*right*(*v*))
InOrder(*left*(*v*))	InOrder(*right*(*v*))
PostOrder(*left*(*v*))	PostOrder(*right*(*v*))

따라서 밥의 코드로 가능한 경우는 정확히 36가지입니다. 안타깝게도 밥은 실행 파일을 제출한 뒤 실수로 소스 코드를 삭제했기 때문에, 여러분도 밥도 어느 위치에서 어떤 함수가 호출되었는지 알지 못합니다.

이제 어떤 ***알 수 없는*** 이진 트리 T에서 실행된 밥의 순회 알고리즘의 출력이 주어졌다고 가정합시다. 밥의 출력은 친절하게도 세 배열 $Pre[1..n]$, $In[1..n]$, $Post[1..n]$으로 파싱되어 있습니다. 이 순회 수열들과 일치하는 이진 트리 T는 정확히 하나뿐이라고 가정해도 됩니다. 특히, 알 수 없는 트리 T의 정점 레이블은 모두 서로 다르며, T의 모든 내부 노드는 정확히 두 개의 자식을 가집니다.

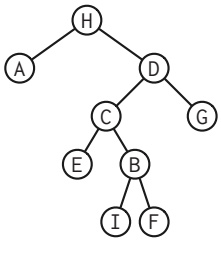
(a) 주어진 순회 순서들로부터 미지의 트리 T를 재구성하는 알고리즘을 설명하십시오.

(b) 주어진 순회 수열로부터 밥의 코드를 재구성하거나, 그 순회 수열과 부합하는 알고리즘 집합이 둘 이상 존재함을 올바르게 보고하는 알고리즘을 설명하십시오.

예를 들어, 다음과 같은 입력이 주어지면

$$Pre[1..n] = [\texttt{H A E C B I F G D}]$$
$$In[1..n] = [\texttt{A H D C E I F B G}]$$
$$Post[1..n] = [\texttt{A E I B F C D G H}]$$

여러분이 작성할 첫 번째 알고리즘은 다음 트리를 반환해야 합니다:



그리고 두 번째 알고리즘은 다음 코드를 복원해야 합니다.

```
PreOrder(v):
  if v = Null
      return
  else
      print label(v)
      PreOrder(left(v))
      PostOrder(right(v))
```

```
InOrder(v):
  if v = Null
      return
  else
      PostOrder(left(v))
      print label(v)
      PreOrder(right(v))
```

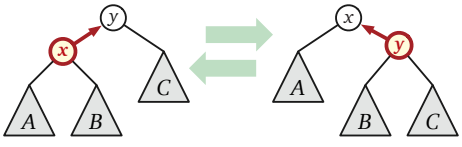
```
PostOrder(v):
  if v = Null
      return
  else
      InOrder(left(v))
      InOrder(right(v))
      print label(v)
```

42. T를 각 노드에 서로 다른 수치 값이 저장된 이진 트리라고 합시다. T가 ***이진 탐색 트리***일 필요충분조건은 (1) T가 비어 있거나, (2) T가 다음의 재귀적 조건을 만족하는 것임을 상기하십시오:

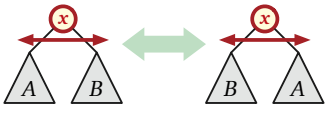
- T의 왼쪽 부분 트리는 이진 탐색 트리입니다.
- 왼쪽 부분 트리의 모든 값은 루트의 값보다 작습니다.
- T의 오른쪽 부분 트리는 이진 탐색 트리입니다.
- 오른쪽 부분 트리의 모든 값은 루트의 값보다 큽니다.

이진 트리에 대한 다음 두 연산을 살펴봅시다:

- 임의의 노드를 위로 ***회전***합니다.[22]



- 임의의 노드의 왼쪽 부분 트리와 오른쪽 부분 트리를 ***교환***합니다.



이 두 연산 모두에서 부분 트리 A, B, C 중 일부나 전부가 비어 있을 수도 있고, 하나도 비어 있지 않을 수도 있습니다.

(a) 노드 값이 서로 다른 *임의의* n-노드 이진 트리를 최대 $O(n^2)$번의 회전과 교환을 사용하여 이진 탐색 트리로 변환하는 알고리즘을 서술하십시오. 그림 1.29는 노드 다섯 개로 이루어진 이진 트리를 여덟 번의 연산으로 이진 탐색 트리로 변환하는 과정을 보여 줍니다.

알고리즘은 부모 포인터나 자식 포인터를 직접 수정하거나, 새 노드를 생성하거나, 기존 노드를 삭제할 수 없습니다. 트리를 수정하는 *유일한* 방법은 회전과 교환입니다.

[22] 회전은 이진 트리에 있는 노드들의 중위 순서를 보존합니다. 이러한 이유도 있어서 회전은 AVL 트리, 레드-블랙 트리, 스플레이 트리, 희생양 트리, 트립을 비롯한 여러 종류의 균형 이진 탐색 트리를 유지하는 데 사용됩니다. 이 자료구조들 대부분에 대한 강의 노트는 http://algorithms.wtf 를 참고하십시오.

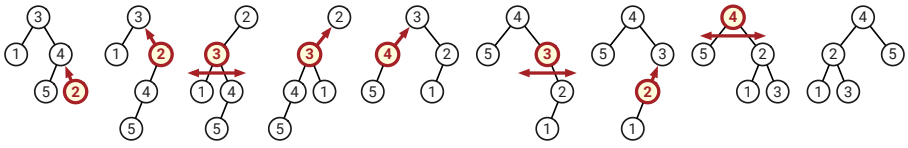


그림 1.29. 이진 트리 "정렬하기": 회전 2, 회전 2, 교환 3, 회전 3, 회전 4, 교환 3, 회전 2, 교환 4.

반면에, 그 계산이 트리를 수정하지 않는 한 원하는 것은 무엇이든 비용 없이 *계산*할 수 있습니다. 알고리즘의 실행 시간은 그 알고리즘이 수행하는 회전과 교환의 횟수로 *정의*됩니다.

♥(b) 임의의 n-노드 이진 트리를 최대 $O(n \log n)$번의 회전과 교환을 사용하여 이진 탐색 트리로 변환하는 알고리즘을 설명하십시오.

(c) n개의 노드로 이루어진 임의의 이진 *탐색* 트리를 같은 노드 값들을 갖는 다른 임의의 이진 *탐색* 트리로 $O(n)$번의 회전만으로(교환은 *전혀* 사용하지 않고) 변환할 수 있음을 증명하십시오.

♥(d) **미해결 문제:** 임의의 n-노드 이진 트리를 $O(n)$번의 회전과 교환만으로 이진 탐색 트리로 변환하는 알고리즘을 설명하거나, 그러한 알고리즘이 존재할 수 없음을 증명하십시오. *[힌트: 저는 불가능하다고 생각합니다.]*

그러나 신앙의 규범으로도 문맥으로도 모호함을 해소할 수 없는 경우에는, 떠오르는 여러 방법 가운데 어느 것을 택하여 문장에 구두점을 찍더라도 우리를 막을 것은 아무것도 없습니다.

— 히포의 아우구스티누스, *De doctrina Christiana* (397년)
마커스 도즈(Marcus Dods) 번역 (1892년)

저는 저녁 식사를 내팽개치고 실험실로 달려 돌아갔습니다. 그곳에서 흥분한 나머지 탁자 위에 있던 모든 비커와 증발 접시의 내용물을 맛보았습니다. 다행히도 그중 어느 것에도 부식성 액체나 독성 액체는 들어 있지 않았습니다.

— 사카린 발견에 관하여, 콘스탄틴 팔베르크,
Scientific American (1886)

어떤 사상가에게든 가장 큰 과제는 해답을 허용하는 방식으로 문제를 진술하는 것입니다.

— 버트런드 러셀의 말로 전해짐

길을 가다 갈림길을 만나면, 그 길로 가십시오.

— 요기 베라 (자기 집으로 가는 길을 알려 주며)

2

백트래킹

이 장에서는 ***백트래킹***이라는 또 다른 중요한 재귀 전략을 설명합니다. 백트래킹 알고리즘은 계산 문제의 해를 한 번에 작은 조각 하나씩 점진적으로 구성하려고 시도합니다. 해의 다음 구성 요소로 여러 대안 중 하나를 골라야 할 때마다, 알고리즘은 **모든** 대안을 재귀적으로 평가한 다음 그중 가장 좋은 것을 선택합니다.

2.1 N 퀸 문제

대표적인 백트래킹 문제는 고전적인 ***n 퀸 문제***입니다. 이 문제는 1848년 독일의 체스 애호가 막스 베젤(Max Bezzel)이 "Schachfreund"라는 필명으로 표준 8×8 체스판에 대해 처음 제안했고, 1869년 프랑수아조제프 외스타슈 리오네(François-Joseph Eustache Lionnet)가 더 일반적인 $n \times n$ 체스판에 대해 제안했습니다. 이 문제는 $n \times n$ 체스판 위에 퀸 n개를 어떤 두 퀸도 서로 공격하지 않도록 배치하는 것입니다. 체스 규칙에 익숙하지 않은 독자를 위해 설명하면, 이는 어떤 두 퀸도 같은 행, 같은 열, 같은 대각선 위에 놓이지 않는다는 뜻입니다.

1850년 저명한 수학자 카를 프리드리히 가우스는 친구 하인리히 슈마허에게 보낸 편지에서, 여덟 퀸 문제의 해가 92개라는 프란츠 나우크의 주장은 시행착오를 거쳐 몇

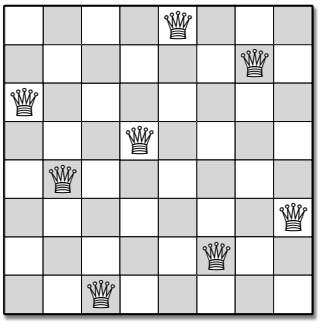


그림 2.1. 배열 [5, 7, 1, 4, 2, 8, 6, 3]으로 표현되는, 8퀸 문제에 대한 가우스의 첫 번째 해

시간이면 쉽게 확인할 수 있다고 썼습니다. ("*Schwer ist es übrigens nicht, durch ein methodisches Tatonniren sich diese Gewissheit zu verschaffen, wenn man 1 oder ein paar Stunden daran wenden will.*") 그가 사용한 *Tatonniren*이라는 표현은 프랑스어 *tâtonner*에서 유래한 말로, 마치 어둠 속에 있는 것처럼 앞을 보지 못한 채 손으로 더듬거나 더듬어 찾거나 헤매는 것을 뜻합니다.

가우스의 편지는 n-퀸 문제를 푸는 다음과 같은 재귀 전략을 설명했습니다. 1882년에는 프랑스의 오락 수학자 에두아르 뤼카(Édouard Lucas)도 같은 전략을 설명했는데, 그는 이 방법을 에마뉘엘 라키에르(Emmanuel Laquière)의 것이라고 밝혔습니다. 맨 위 행부터 시작하여 한 번에 한 행씩 판 위에 퀸을 놓습니다. r번째 퀸을 놓을 때는 간단한 for 반복문으로 행 r의 n개 칸을 왼쪽에서 오른쪽으로 하나씩 차례대로 모두 시도합니다. 어떤 칸이 앞서 놓은 퀸에게 공격받는다면 그 칸은 무시합니다. 그렇지 않다면 그 칸에 퀸을 잠정적으로 놓고, 이후 행들에 퀸을 모순 없이 배치하는 방법을 *재귀적*으로 더듬어 찾습니다.

그림 2.2는 그 결과로 얻은 알고리즘을 보여 줍니다. 이 알고리즘은 주어진 부분 해와 모순되지 않는 완전한 n-퀸 해를 모두 재귀적으로 열거합니다. 가우스를 따라 퀸의 위치는 배열 $Q[1..n]$으로 나타내며, 여기서 $Q[i]$는 i행의 어느 칸에 퀸이 놓여 있는지를 나타냅니다. PlaceQueens가 호출될 때 입력 매개변수 r은 첫 번째 빈 행의 인덱스이고, 접두사 $Q[1..r-1]$에는 처음 $r-1$개 퀸의 위치가 들어 있습니다. 특히 아무 제약 없이 모든 n-퀸 해를 계산하려면 PlaceQueens$(Q[1..n], 1)$을 호출하면 됩니다. 바깥쪽 for 루프는 r행에 퀸을 놓을 수 있는 모든 위치를 고려하고, 안쪽 루프는 r행의 후보 위치가 처음 $r-1$개 행에 이미 놓인 퀸들과 모순되지 않는지 검사합니다.

PlaceQueens의 실행 과정은 ***재귀 트리***를 사용하여 나타낼 수 있습니다. 이 트리의 각 노드는 재귀적 부분 문제에 대응하며, 따라서 하나의 유효한 부분 해에 대응합니다. 특히 루트는 빈 판($r = 0$인 경우)에 대응합니다. 재귀 트리의 간선은 재귀 호출에 대응합니다. 리프는 더 이상 확장할 수 없는 부분 해에 대응하는데, 이는 이미 모든 행에 퀸이 놓여 있거나, 다음 빈 행의 모든 위치가 기존 퀸에게 공격받고 있기 때문입니다. 완전한 해를 찾는 백트래킹 탐색은 이 트리에 대한 깊이 우선 탐색과 동등합니다.

```
PlaceQueens(Q[1..n], r):
  if r = n + 1
      print Q[1..n]
  else
      for j ← 1 to n
          legal ← True
          for i ← 1 to r − 1
              if (Q[i] = j) or (Q[i] = j + r − i) or (Q[i] = j − r + i)
                  legal ← False
          if legal
              Q[r] ← j
              PlaceQueens(Q[1..n], r + 1)                    《재귀!》
```

그림 2.2. n 퀸 문제에 대한 가우스와 라키에르의 백트래킹 알고리즘입니다.

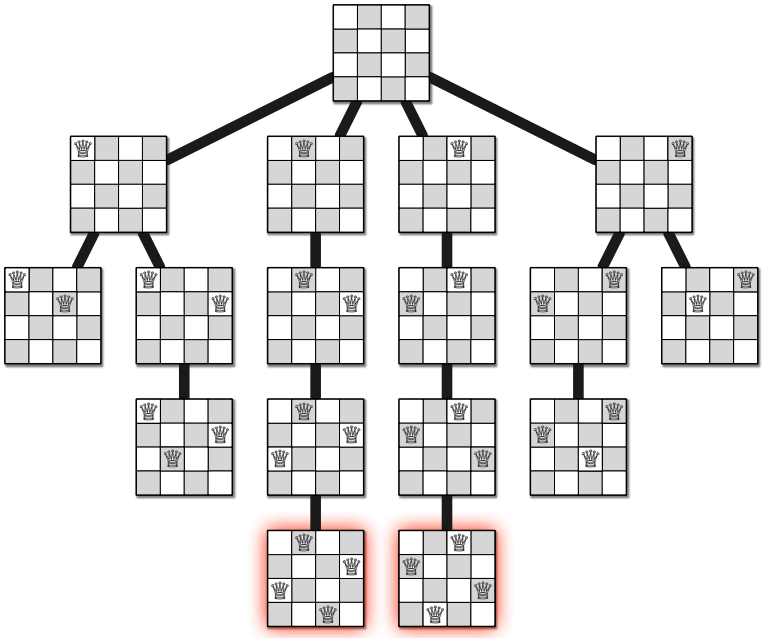


그림 2.3. 4퀸 문제에 대한 가우스와 라키에르 알고리즘의 전체 재귀 트리입니다.

2.2 게임 트리

다음과 같은 간단한 2인용 게임[1]을 생각해 봅시다. 이 게임은 가장자리에 한 줄의 칸이 둘러진 $n \times n$ 정사각형 격자 위에서 진행됩니다. 두 플레이어를 Horace Fahlberg-Remsen과 Vera Rebaudi라고 부릅시다.[2] 각 플레이어는 n개의 토큰을 가지고 있으며,

[1] 저는 이 게임의 이름이 무엇인지 모르며, 규칙을 제대로 전달하고 있는지조차 확신하지 못합니다. 저는 이 게임(또는 이와 비슷한 게임)을 Lenny Pitt에게서 배웠는데, 그는 식당에서 인공 감미료 봉지로 이 게임을 해 보라고 권했습니다.

[2] Constantin Fahlberg와 Ira Remsen은 1878년에 처음으로 사카린을 합성했습니다. 당시 Fahlberg는 Remsen의 연구실에서 콜타르 유도체를 연구하던 박사후연구원이었습니다. 1900년에 Ovidio Rebaudi는

이 토큰들을 보드의 한쪽에서 반대쪽으로 가로질러 옮깁니다. Horace의 토큰은 왼쪽 가장자리에서 각 행에 하나씩 놓인 채로 시작하여 오른쪽으로 수평(***hor***izontally) 이동합니다. 이와 대칭적으로, Vera의 토큰은 위쪽 가장자리에서 각 열에 하나씩 놓인 채로 시작하여 아래쪽으로 수직(***ver***tically) 이동합니다. 두 플레이어는 번갈아 가며 차례를 진행합니다. Horace는 자신의 각 차례에 토큰 하나를 오른쪽으로 한 칸 옮겨 빈칸으로 *이*동하거나, 토큰 하나가 Vera의 토큰 정확히 하나를 뛰어넘어 오른쪽으로 두 칸 떨어진 빈칸으로 *점프*하게 합니다. 규칙에 맞는 이동이나 점프가 없으면 Horace는 그냥 차례를 넘깁니다. 마찬가지로 Vera도 이동이나 점프가 불가능한 경우가 아니라면 자신의 각 차례에 토큰 하나를 아래쪽으로 이동하거나 점프하게 합니다. 자신의 토큰을 모두 보드 가장자리의 간선 밖으로 먼저 내보내는 플레이어가 승리합니다. (보드 위에 토큰이 남아 있는 한 적어도 한 플레이어는 규칙에 맞게 움직일 수 있으며, 따라서 결국 누군가는 승리한다는 사실은 어렵지 않게 증명할 수 있습니다.)

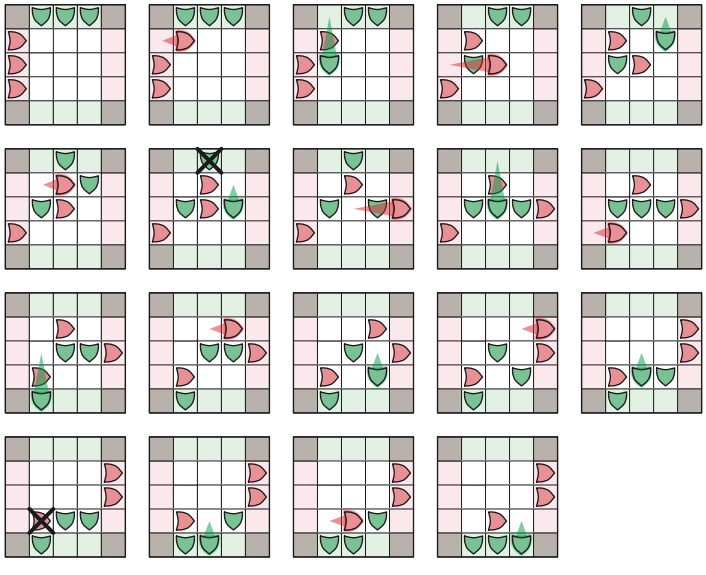


그림 2.4. Vera는 3 × 3 가짜 설탕 봉지 게임에서 이깁니다.

이 게임을 전에 본 적이 없다면[3], 아마 이 게임을 잘하는 방법을 전혀 모를 것입니다. 그럼에도 이 게임을—또는 무작위성이나 숨겨진 정보가 없고 유한 번의 수를 둔 뒤에 끝나는 모든 2인용 게임을—*완벽하게* 플레이할 수 있는 비교적 간단한 백트래킹 알고리즘이 있습니다. 즉, 여러분을 게임 도중에 투입했을 때 또 다른 완벽한 플레이어를 상대로 이기는 것이 *가*능하다면, 이 알고리즘은 이기는 방법을 알려 줄 것입니다.

게임의 ***상태***는 모든 말의 위치와 현재 차례인 플레이어가 누구인지로 이루어집니다. 이러한 상태들을 연결하여 *게임 트리*를 만들 수 있습니다. 게임 트리에는 상태 x의 현재 플레이어가 상태 y로 규칙에 맞게 이동할 수 있을 때, 그리고 오직 그때에만 상태

*ka'a he'ê*에 대한 최초의 화학 분석을 발표했습니다. 이 식물은 과라니족이 1500년 넘게 재배해 온 약용 식물로, 오늘날에는 *Stevia rebaudiana*라는 이름으로 더 널리 알려져 있습니다.

[3] 본 적이 있다면 어디서 보았는지 꼭 알려 주십시오!

x에서 상태 y로 가는 간선이 있습니다. 게임 트리의 루트는 게임의 초기 배치이며, 루트에서 리프까지의 모든 경로는 하나의 완전한 게임입니다.

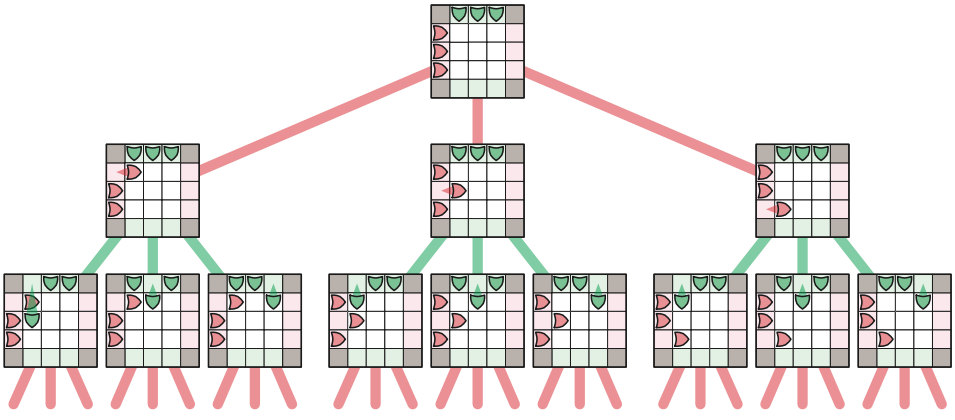


그림 2.5. 가짜 설탕 봉지 게임 트리의 처음 두 단계입니다.

이 게임 트리를 탐색하기 위해, 게임 상태가 *좋은* 상태인지 *나쁜* 상태인지를 다음과 같이 재귀적으로 정의합니다:

- 현재 플레이어가 이미 이겼거나, 현재 플레이어가 상대 플레이어에게 나쁜 상태로 이동할 수 있다면, 그 게임 상태는 *좋은* 상태입니다.

- 현재 플레이어가 이미 졌거나, 둘 수 있는 모든 수가 상대 플레이어에게 좋은 상태로 이어지면 그 게임 상태는 *나쁜* 상태입니다.

같은 내용을 달리 표현하면, 게임 트리에서 리프가 아닌 노드는 나쁜 자식이 적어도 하나 있으면 좋은 노드이고, 모든 자식이 좋은 노드이면 나쁜 노드입니다. 귀납법에 따라, 자기 차례에 게임이 좋은 상태에 있는 플레이어는 상대가 완벽하게 플레이하더라도 게임에서 이길 수 있습니다. 반면에 나쁜 상태에서 시작하는 플레이어는 상대가 실수를 해야만 이길 수 있습니다. 이 재귀적 정의는 1913년에 에른스트 체르멜로(Ernst Zermelo)가 제안했습니다.[4]

이 재귀적 정의로부터 주어진 게임 상태가 좋은지 나쁜지를 판별하는 다음 재귀 백트래킹 알고리즘을 곧바로 떠올릴 수 있습니다. 본질적으로 이 알고리즘은 게임 트리에 대한 깊이 우선 탐색일 뿐입니다. 달리 말하면, 게임 트리가 곧 이 알고리즘의 재귀 트리입니다! 입력이 좋은 게임 상태라면, 이 백트래킹 알고리즘을 간단히 수정하여 좋은 수 하나를(또는 가능한 모든 좋은 수까지도) 찾을 수 있습니다.

[4] 사실 체르멜로는 이보다 더 미묘한 부류의 게임, 즉 상태의 수는 유한하지만 무한히 이어지는 수순을 허용하는 게임을 다루었습니다. (체르멜로는 무한히 계속되는 플레이를 무승부로 정의했습니다.)

```
PlayAnyGame(X, player):
  if player has already won in state X
      return Good
  if player has already lost in state X
      return Bad
  for all legal moves X ⇝ Y
      if PlayAnyGame(Y, ¬player) = Bad
          return Good     《X ⇝ Y는 좋은 수입니다》
  return Bad              《좋은 수가 없습니다》
```

모든 게임 플레이 프로그램은 궁극적으로 이 단순한 백트래킹 전략에 기반합니다. 그러나 대부분의 게임은 상태의 수가 엄청나게 많기 때문에 실제로는 게임 트리 전체를 순회할 수 없습니다. 그 대신 게임 프로그램은 다른 휴리스틱[5]을 사용하여 게임 트리를 *가지치기*합니다. 명백히(또는 "명백히") 좋거나 나쁜 상태, 또는 적어도 다른 상태보다 더 좋거나 더 나쁜 상태를 무시하거나, 트리를 특정 깊이(또는 *플라이*)에서 잘라 내고 더 효율적인 휴리스틱으로 리프를 평가하거나, 이 두 방법을 함께 사용합니다.

2.3 부분집합 합

좀 더 복잡한 문제인 SubsetSum을 생각해 봅시다. 양의 정수의 집합 X와 *대상* 정수 T가 주어졌을 때, 더해서 T가 되는 X의 원소들의 부분집합이 존재합니까? 그러한 부분집합이 둘 이상 있을 수 있다는 점에 유의하십시오. 예를 들어 $X = \{8, 6, 7, 5, 3, 10, 9\}$이고 $T = 15$이면 답은 True입니다. 부분집합 $\{8, 7\}$, $\{7, 5, 3\}$, $\{6, 9\}$, $\{5, 10\}$의 합이 모두 15이기 때문입니다. 반면에 $X = \{11, 6, 5, 1, 7, 13, 12\}$이고 $T = 15$이면 답은 False입니다.

자명한 경우가 두 가지 있습니다. 대상 값 T가 0이면 즉시 True를 반환할 수 있습니다. 공집합은 모든 집합 X의 부분집합이고, 공집합의 원소들을 더하면 0이 되기 때문입니다.[6] 반면 $T < 0$이거나, $T \neq 0$인데 집합 X가 비어 있으면 즉시 False를 반환할 수 있습니다.

일반적인 경우에는 임의의 원소 $x \in X$를 고려합니다. (X가 비어 있는 경우는 이미 처리했습니다.) 합이 T가 되는 X의 부분집합이 존재할 필요충분조건은 다음 명제 중 하나가 참인 것입니다.

- X의 부분집합 중에서 x를 *포함하고* 합이 T인 것이 존재합니다.
- x를 *포함하지 않으며* 합이 T인 X의 부분집합이 존재합니다.

첫 번째 경우에는 합이 $T - x$가 되는 $X \setminus \{x\}$의 부분집합이 반드시 존재해야 하고, 두 번째 경우에는 합이 T가 되는 $X \setminus \{x\}$의 부분집합이 반드시 존재해야 합니다. 따라서 SubsetSum(X, T)를 더 단순한 두 인스턴스인 SubsetSum$(X \setminus \{x\}, T - x)$와 SubsetSum$(X \setminus \{x\}, T)$로 환원하여 풀 수 있습니다. 이렇게 얻은 재귀 알고리즘은 아래와 같습니다.

[5] 휴리스틱이란 작동하지 않는 알고리즘입니다. (실전에서는 예외입니다. 가끔은 말입니다. 아마도 말입니다.)

[6] …그 밖에 무엇이 될 수 있겠습니까?

```
《X의 부분집합 중 합이 T가 되는 것이 있습니까?》
SubsetSum(X, T):
  if T = 0
      return True
  else if T < 0 or X = ∅
      return False
  else
      x ← any element of X
      with ← SubsetSum(X ∖ {x}, T − x)     《재귀하십시오!》
      wout ← SubsetSum(X ∖ {x}, T)         《재귀하십시오!》
      return (with ∨ wout)
```

정확성

이 알고리즘이 올바르다는 것은 귀납법으로 간단히 증명할 수 있습니다. $T = 0$이면 공집합의 원소들의 합이 T이므로 True가 올바른 출력입니다. 그렇지 않고 T가 음수이거나 집합 X가 비어 있으면 합이 T가 되는 X의 부분집합은 존재하지 않으므로 False가 올바른 출력입니다. 그 밖의 경우에 합이 T가 되는 부분집합이 있다면 그 부분집합은 $X[n]$을 포함하거나 포함하지 않으며, 재귀 요정은 이 두 가능성을 각각 올바르게 확인합니다. 이로써 증명이 끝납니다.

분석

알고리즘을 분석하려면 몇 가지 구현 세부 사항을 더 정확하게 기술해야 합니다. 우선, 입력 집합 X가 배열 $X[1..n]$으로 주어진다고 가정합시다.

앞의 재귀 알고리즘에서는 주된 재귀 사례에서 *임의의* 원소 $x \in X$를 선택할 수 있습니다. 순전히 효율성을 위해서는 남은 부분집합 $X \setminus \{x\}$를 빠르게 계산할 수 있는 간결한 형태로 나타낼 수 있도록 원소 x를 고르는 것이 좋습니다. 그러면 재귀 호출을 준비하는 데 드는 오버헤드를 최소로 줄일 수 있습니다. 구체적으로, x를 마지막 원소 $X[n]$으로 정하겠습니다. 그러면 부분집합 $X \setminus \{x\}$는 접두사 $X[1..n-1]$에 저장됩니다. 이 접두사의 완전한 *복사본*을 재귀 호출에 넘기면 시간이 너무 오래 걸립니다—복사본을 만드는 데만 $\Theta(n)$ 시간이 필요합니다—. 따라서 대신 배열에 대한 참조(또는 배열의 시작 주소)와 접두사의 길이라는 두 값만 푸시합니다. (또는 X를 전역 변수로 만들어 모든 재귀 호출에 X에 대한 참조를 넘기지 않아도 되게 할 수도 있습니다.)

```
《X[1..i]의 부분집합 중 합이 T가 되는 것이 있습니까?》
SubsetSum(X, i, T):
  if T = 0
      return True
  else if T < 0 or i = 0
      return False
  else
      with ← SubsetSum(X, i − 1, T − X[i])    《재귀하십시오!》
      wout ← SubsetSum(X, i − 1, T)           《재귀하십시오!》
      return (with ∨ wout)
```

이러한 방식으로 구현하면 우리 알고리즘의 실행 시간 $T(n)$은 점화식 $T(n) \leq 2T(n-1) + O(1)$을 만족합니다. 이 점화식의 해 $T(n) = O(2^n)$은 재귀 트리를 사용하거나, 그보다 더 간단한 "아, 이 점화식은 하노이의 탑에서 이미 풀었지" 방법을 사용하면 쉽게 얻을 수 있습니다. 최악의 경우—예를 들어 T가 X의 모든 원소의 합보다 클 때—이 알고리즘의 재귀 트리는 깊이가 n인 완전 이진 트리가 되며, 알고리즘은 X의 부분집합 2^n개를 모두 고려합니다.

변형

약간만 수정하면 SubsetSum의 여러 변형 문제를 풀 수 있습니다. 예를 들어 그림 2.6은 합이 T가 되는 X의 부분집합이 존재하면 그 부분집합을 실제로 *구성*하고, 그런 부분집합이 존재하지 않으면 오류 값 None을 반환하는 알고리즘을 보여 줍니다. 이 알고리즘은 앞서 살펴본 결정 알고리즘과 정확히 같은 재귀 전략을 사용합니다. 이 알고리즘 역시 $O(2^n)$ 시간에 실행됩니다. 원소 하나를 $O(1)$ 시간에 삽입할 수 있는 집합 자료구조(예를 들어 연결 리스트)를 가정하면 분석이 가장 간단하지만, 실제로는 삽입에 $O(n)$ 시간이 걸리더라도(예를 들어 *정렬된* 연결 리스트) 실행 시간은 여전히 $O(2^n)$입니다. 이와 비슷한 변형을 이용하면 합이 특정 값이 되는 부분집합의 개수를 세거나, 합이 특정 값이 되는 부분집합 중에서 (다른 어떤 기준에 따라) *가장 좋은* 부분집합을 고를 수도 있습니다.

백트래킹으로 푸는 다른 문제들도 대부분 이런 성질을 가집니다. 같은 재귀 전략으로 같은 문제의 여러 변형을 풀 수 있다는 것입니다. 예를 들어, 앞 절에서 설명한 재귀 전략은 주어진 게임 국면이 좋은지 나쁜지를 판별합니다. 이 전략을 조금만 수정하면 판별 결과 대신 좋은 수 하나를 반환하거나 좋은 수 전체의 목록을 반환하게 만들 수 있습니다. 그러므로 백트래킹 알고리즘을 *설계*할 때에는 문제를 가능한 한 가장 단순한 형태로 다루는 것을 목표로 삼아야 합니다. 더 복잡한 정보나 구조를 구하지 말고 수 하나, 더 나아가 불리언 값 하나만 계산해야 합니다.

2.4 일반적인 패턴

백트래킹 알고리즘은 흔히 *일련의 결정*을 내리는 데 사용되며, 그 목표는 특정 제약 조건을 만족하는 재귀적으로 정의된 구조를 만드는 것입니다. 이 목표 구조 자체가 수열인 경우가 많지만, 항상 그런 것은 아닙니다. 예를 들면 다음과 같습니다.

```
《합이 T가 되는 X[1..i]의 부분집합을 반환합니다》
《또는 그러한 부분집합이 존재하지 않으면 NONE》
CONSTRUCTSUBSET(X, i, T):
  if T = 0
      return ∅
  if T < 0 or n = 0
      return NONE
  Y ← CONSTRUCTSUBSET(X, i − 1, T)
  if Y ≠ NONE
      return Y
  Y ← CONSTRUCTSUBSET(X, i − 1, T − X[i])
  if Y ≠ NONE
      return Y ∪ {X[i]}
  return NONE
```

그림 2.6. SUBSETSUM의 구성 버전을 위한 재귀 백트래킹 알고리즘입니다.

- n-퀸 문제의 목표는 어떤 두 퀸도 서로를 공격하지 않도록 각 행에 하나씩 놓인 퀸 위치들의 수열을 구하는 것입니다. 알고리즘은 각 행마다 퀸을 어디에 놓을지 *결정*합니다.
- 게임 트리 문제에서 목표는 각 수가 그 수를 두는 플레이어에게 가능한 한 최선이 되도록 하는 합법적인 수들의 나열입니다. 각 게임 상태에 대해 알고리즘은 가능한 최선의 다음 수를 *결정합니다*.
- SUBSETSUM 문제의 목표는 합이 특정한 값이 되는 입력 원소들의 수열입니다. 알고리즘은 각 입력 원소마다 그 원소를 출력 수열에 포함할지 말지를 *결정*합니다.

(잠깐만요, 왜 *부분집합* 합의 목표가 *수열*을 찾는 것일까요? 이는 의도적인 설계 결정이었습니다. 우리는 입력 집합을 다른 더 형태가 모호한 자료구조가 아니라 배열로 표현함으로써—재귀 알고리즘에서 활용할 수 있는 편리한 순서를 입력 집합에 부여했습니다.)

백트래킹 알고리즘의 각 재귀 호출에서는 ***정확히 하나의*** 결정을 내려야 하며, 그 선택은 이전의 모든 결정과 일관되어야 합니다. 따라서 각 재귀 호출에는 아직 처리하지 않은 입력 데이터의 부분뿐만 아니라, 이미 내린 결정들을 적절히 요약한 정보도 필요합니다. 효율성을 위해 과거 결정의 요약은 가능한 한 작아야 합니다. 예를 들면 다음과 같습니다.

- n-퀸 문제에서는 비어 있는 행의 개수뿐만 아니라 앞서 배치한 모든 퀸의 위치까지 넘겨주어야 합니다. 안타깝게도 여기서는 과거에 내린 결정을 하나도 빠짐없이 기억해야 합니다.
- 게임 트리 문제에서는 다음 차례의 플레이어가 누구인지를 포함한 현재 게임 상태만 넘겨주면 됩니다. 주어진 게임 상태에서 누가 이기는지는 그 상태에 이르게 한 수들과 무관하므로, 과거의 결정에 대해서는 *아무것*도 기억할 필요가 없습니다.[7]

[7]많은 게임이 이 독립성 조건을 위반하는 *것처럼 보입니다*. 예를 들어 체스와 체커의 표준 규칙은 모두 같은 기물 배치가 세 번 나타나면 플레이어가 무승부를 선언할 수 있도록 허용하며, 바둑의 중국식

- SubsetSum 문제에서는 남아 있는 사용 가능한 정수들과 함께 남은 대상 값도 전달해야 합니다. 남은 대상 값은 원래 대상 값에서 이전에 선택한 원소들의 *합*을 뺀 값입니다. 이전에 정확히 어떤 원소들을 선택했는지는 중요하지 않습니다.

새로운 재귀 백트래킹 알고리즘을 설계할 때에는 *알고리즘 도중에* 과거의 결정에 관해 어떤 정보가 필요할지를 *미리* 파악해야 합니다. 이 정보가 자명하지 않다면, 재귀 알고리즘은 원래 풀도록 요청받은 문제보다 더 일반적인 문제를 풀어야 할 수도 있습니다. (이런 종류의 일반화는 이전에도 본 적이 있습니다. 정렬되지 않은 배열의 *중앙값*을 선형 시간에 찾기 위해, *임의의* k에 대해 k번째로 작은 원소를 선택하는 알고리즘을 유도했습니다.)

마지막으로, 우리가 *실제로* 풀어야 하는 재귀 문제가 무엇인지 파악하고 나면, 그 문제를 ***재귀적 무차별 대입***으로 풉니다. 즉, 이전 결정들과 모순되지 않는 다음 결정의 가능성을 모두 시도하고, 나머지는 재귀 요정에게 맡기십시오. 여기서는 영리하게 굴지 마십시오. "명백히" 어리석은 선택이라도 건너뛰지 마십시오. 모든 것을 시도하십시오. 알고리즘을 더 빠르게 만드는 일은 나중에 해도 됩니다.

2.5 텍스트 분할(*Interpunctio Verborum*)

어떤 외국어로 된 텍스트를 나타내는 문자열이 공백이나 문장 부호 없이 주어졌고, 이 문자열을 구성하는 개별 단어들로 나누고 싶다고 가정합시다. 예를 들어, 기원전 62년에 키케로가 루키우스 리키니우스 무레나를 변호하며 행한 유명한 연설의 다음 구절이 고전 라틴어의 표준 표기 방식인 *scriptio continua*(연속 표기)로 주어질 수 있습니다:[8]

PRIMVSDIGNITASINTAMTENVISCIENTIANONPOTEST
ESSERESENIMSVNTPARVAEPROPEINSINGVLISLITTERIS
ATQVEINTERPVNCTIONIBUSVERBORVMOCCVPATAE

라틴어에 능통한 독자라면 이 문자열을 (현대 철자법으로) *Primus dignitas in tam tenui scientia non potest esse; res enim sunt parvae, prope in singulis litteris atque interpunctionibus verborum occupatae*로 읽어 낼 것입니다.[9] 텍스트 분할은 고전 라틴어와 그리스어만의 문제가 아니라 발리어, 버마어, 중국어, 일본어, 자바어, 크메르어, 라오어, 태국어, 티베트어, 베트남어를 비롯한 여러 현대 언어와 문자 체계에서도 나타나는 문제입니다. 이와 비슷한 문제는 구두점이 없는 영어 텍스트를 문장으로

규칙은 이전에 나왔던 돌의 배치를 되풀이하는 것을 아예 금지합니다. 따라서 이러한 게임에서 게임 상태는 형식적으로 기물의 현재 위치뿐 아니라 이전에 둔 수의 전체 기록까지 포함합니다.

[8] 사실·대부분의·고전·라틴어·필사본은·*interpuncts*(중점)라고·불리는·작은·점으로·단어를·구분했습니다. 중점 구두법은 3세기 무렵 *scriptio continua*에 밀려 거의 사라졌습니다. 단어 사이의 공백은 8세기에 아일랜드 수도사들이 도입했으며, 이후 몇 세기에 걸쳐 유럽 전역으로 서서히 퍼져 나갔습니다. *Scriptio continua*는 21세기 초 영어에서도 URL과 해시태그의 형태로 살아남아 있습니다. #octotherps4lyfe

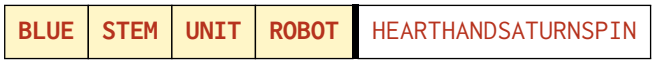
[9] 대략 옮기면 다음과 같습니다. "우선, 이토록 하찮은 지식에 품위란 있을 수 없습니다. 이는 사소한 것으로, 대개 낱글자와 단어 사이에 점을 찍는 일에 관한 것일 뿐입니다." 키케로는 자신의 친구(!)이자 저명한 법률가였던 세르비우스 술피키우스 루푸스의 법률 전문성을 대놓고 조롱하고 있었습니다. 루푸스는 집정관 선거에서 무레나에게 패한 뒤 무레나를 뇌물 수수 혐의로 고발했습니다. 무레나는 거의 확실히 유죄였음에도, 키케로의 신랄한 변론에 일부 힘입어 무죄 판결을 받았습니다. #librapondo #nunquamestfidelis

나누는 일,[10] 조판을 위해 텍스트를 줄 단위로 나누는 일, 음성 및 필기 인식, 곡선 단순화, 그리고 여러 유형의 시계열 분석에서도 발생합니다. 설명을 위해, 여기서는 현대 영어 알파벳 글자의 나열을 현대 영어 단어로 분할하는 문제만 다루겠습니다.

물론 어떤 문자열은 여러 가지 방식으로 분할될 수 있습니다. 예를 들어 BOTHEARTHANDSATURNSPIN은 영어 단어로 BOTH · EARTH · AND · SATURN · SPIN으로도, BOT · HEART · HANDS · AT · URNS · PIN으로도 분해될 수 있으며, 이 밖에도 가능한 방식이 많습니다. 우선은 극도로 단순한 분할 문제를 생각해 봅시다. 문자열이 주어졌을 때, 이 문자열을 영어 단어로 분할하는 방법이 *하나라도* 있습니까?

문제를 구체적으로 (그리고 특정 언어에 얽매이지 않도록) 만들기 위해, 문자열 w를 입력으로 받아 w가 "단어"이면 True를, w가 "단어"가 아니면 False를 반환하는 서브루틴 IsWord(w)를 사용할 수 있다고 가정합시다. 예를 들어 입력 문자열을 회문들로 분해하려 한다면 "단어"는 "회문"과 같은 뜻이므로, IsWord(ROTATOR) = True이지만 IsWord(PALINDROME) = False입니다.

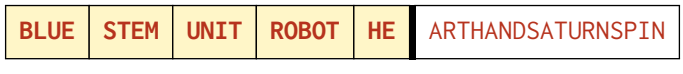
SubsetSum 문제와 마찬가지로 *입력* 구조는 수열이며, 이번에는 숫자 대신 문자로 이루어져 있습니다. 따라서 입력 문자를 왼쪽에서 오른쪽으로 차례대로 소비하는 결정 과정을 생각하는 것이 자연스럽습니다. 마찬가지로 *출력* 구조는 단어의 수열이므로, 출력 단어를 왼쪽에서 오른쪽으로 차례대로 생성하는 과정을 생각하는 것이 자연스럽습니다. 따라서 분할 과정의 한가운데로 바로 들어가 보면, 다음과 같은 그림을 떠올릴 수 있습니다:



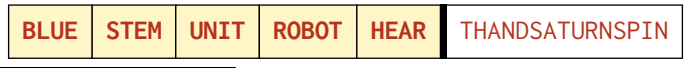
여기서 검은 막대는 지금까지 내린 결정, 즉 처음 17개의 글자를 네 개의 단어로 나눈 결정과 입력 문자열 중 아직 처리하지 않은 부분을 구분합니다.

우리가 상상하는 과정의 다음 단계는 출력 수열의 다음 단어가 어디서 끝나는지 ***결정***하는 것입니다. 이 구체적인 예에서는 다음 출력 단어로 가능한 경우가 네 가지입니다—HE, HEAR, HEART, HEARTH입니다. 이 선택지 중 어느 것이 입력 문자열의 완전한 분할과 일치하는지, 아니면 그런 선택지가 하나도 없는지 우리는 *전혀 알지 못합니다*. 여기서 "똑똑하게" 굴면서 어떤 선택지가 좋은지 *알아내려고* 할 수도 있지만, 그러려면 *생각*을 해야 합니다! 그 대신 무차별 대입으로 모든 가능성을 "멍청하게" 시도해 보고, 실제 작업은 모두 재귀 요정에게 맡깁시다.

- 먼저 HE를 다음 단어로 *잠정적으로* 받아들이고, 나머지 결정은 재귀 요정에게 맡기십시오.



- 그런 다음 HEAR를 다음 단어로 *잠정적으로* 받아들이고, 나머지 결정은 재귀 요정에게 맡기십시오.

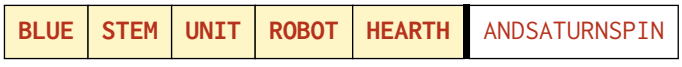


[10] 성 아우구스티누스의 *De doctrina Christiana*는 구두점을 추가하여 라틴어 성서의 모호함을 없애는 일에 한 장 전체를 할애합니다.

- 그런 다음 HEART를 다음 단어로 *잠정적*으로 받아들이고, 나머지 결정은 재귀 요정에게 맡기십시오.

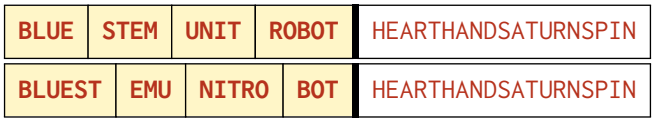


- 마지막으로, HEARTH를 다음 단어로 *잠정적*으로 받아들이고, 나머지 결정은 재귀 요정에게 맡기십시오.



재귀 요정이 적어도 한 번 성공을 보고하기만 하면 우리도 성공을 보고합니다. 반면에 재귀 요정이 *한 번도* 성공을 보고하지 않으면—특히 가능한 다음 단어의 집합이 비어 있으면—우리는 실패를 보고합니다.

과거에 내린 결정 중 어느 것도 지금 어떤 선택지가 가능한지에 영향을 주지 않습니다. 중요한 것은 아직 처리하지 않은 문자들로 이루어진 접미사뿐입니다. 특히 서로 다른 여러 과거 결정의 순서가 같은 접미사에 이를 수 있지만, 어느 경우든 그 접미사에 대해 남는 선택지의 집합은 정확히 같습니다.



따라서 검은 막대 왼쪽에 있는 *모든 것*을 버림으로써 재귀 과정을 나타낸 그림을 단순화할 수 있습니다:



이제 우리에게는 단순하고 자연스러운 백트래킹 전략이 남습니다. *첫 번째 출력 단어를 선택하고, 입력 문자열의 나머지 부분을 재귀적으로 분할합니다.*

완전한 재귀 알고리즘을 얻으려면 기저 사례가 필요합니다. 입력 문자열의 끝에 도달하면 다음 단어가 없으므로 우리의 재귀 전략은 더 이상 작동하지 않습니다. 다행히도 빈 문자열을 0개의 단어로 분할하는 방법은 단 하나뿐입니다!

이 모든 조각을 하나로 합치면 다음과 같은 간단한 재귀 알고리즘을 얻습니다:

```
SPLITTABLE(A[1..n]):
  if n = 0
      return TRUE
  for i ← 1 to n
      if ISWORD(A[1..i])
          if SPLITTABLE(A[i+1..n])
              return TRUE
  return FALSE
```

인덱스 정식화

실제로는 배열을 입력 매개변수로 전달하는 방식이 상당히 느리므로, 재귀적 부분 문제를 기술하는 더 간결한 방법을 꼭 찾아야 합니다. *알고리즘을 설계할 때에는* 원래의 입력 배열을 전역 변수로 취급한 다음, 명시적인 부분 배열 대신 배열 인덱스를 사용하여 문제와 알고리즘을 다시 정식화하는 것이 대단히 유용합니다.

이 문자열 분할 문제에서 모든 재귀 호출의 인자는 항상 원래 입력 배열의 ***접미사*** $A[i..n]$입니다. 따라서 입력 배열 $A[1..n]$을 전역 변수로 취급하면, 재귀 문제를 다음과 같이 다시 정식화할 수 있습니다:

> 인덱스 i가 주어졌을 때, 접미사 $A[i..n]$의 분할을 구하십시오.

알고리즘을 설명하려면 두 가지 불리언 함수가 필요합니다.

- 임의의 인덱스 i와 j에 대해, 부분 문자열 $A[i..j]$가 단어일 때, 그리고 오직 그때에만 $\textsc{IsWord}(i,j) = \textsc{True}$라고 합시다. (이 함수는 주어진다고 가정합니다.)
- 임의의 인덱스 i에 대해, 접미사 $A[i..n]$을 단어들로 분할할 수 있을 때 그리고 오직 그때에만 $\mathit{Splittable}(i) = \textsc{True}$라고 합시다. (이것이 우리가 구현해야 하는 함수입니다.)

예를 들어, $\textsc{IsWord}(1,n) = \textsc{True}$일 필요충분조건은 입력 문자열 전체가 하나의 단어인 것이고, $\mathit{Splittable}(1) = \textsc{True}$일 필요충분조건은 입력 문자열 전체를 분할할 수 있는 것입니다. 앞서 세운 재귀 전략으로부터 다음 점화식을 얻습니다.

$$\mathit{Splittable}(i) = \begin{cases} \textsc{True} & \text{만약} i > n \\ \displaystyle\bigvee_{j=i}^{n} \big(\textsc{IsWord}(i,j) \wedge \mathit{Splittable}(j+1)\big) & \text{그렇지 않으면} \end{cases}$$

이것은 앞서 본 알고리즘과 *정확히* 같은 알고리즘이며, 바뀐 것은 표기법뿐입니다. 점화식을 의사코드로 다시 쓰면 그 유사성이 훨씬 더 분명하게 드러납니다:

```
《접미사 A[i..n]은 분할 가능(Splittable)합니까?》
SPLITTABLE(i):
  if i > n
      return TRUE
  for j ← i to n
      if ISWORD(i, j)
          if SPLITTABLE(j + 1)
              return TRUE
  return FALSE
```

사소한 표기상의 차이처럼 보일 수도 있지만, 배열 표기법 대신 인덱스 표기법을 사용하는 것은 중요한 습관입니다. 이 습관은 실제로 백트래킹 알고리즘의 속도를 높이는 데뿐만 아니라, 다음 장에서 다룰 동적 계획법 알고리즘을 개발하는 데에도 중요합니다.

♥분석

대부분의 백트래킹 알고리즘의 최악의 경우 실행 시간이 지수적이라는 사실은 그리 놀라운 일이 아닙니다. 이러한 알고리즘 중 상당수는 정확한 실행 시간을 분석하려면 이 책의 범위를 벗어나는 기법이 필요합니다. 다행히도 *이 책에서* 접하게 될 백트래킹 알고리즘은 대부분 더 효율적인 알고리즘으로 가는 과정에서 나오는 중간 결과일 뿐이므로, 정확한 최악의 경우 실행 시간은 사실 그다지 중요하지 않습니다. (먼저 동작하게 만들고, 그다음에 빠르게 만드십시오.)

하지만 그냥 재미 삼아 재귀 알고리즘 SplitTable의 실행 시간을 분석해 봅시다. IsWord가 무엇을 하는지 모르므로 IsWord를 한 번 호출하는 데 시간이 얼마나 걸리는지 알 수 없습니다. 따라서 실행 시간을 IsWord의 호출 횟수로 분석할 수밖에 없습니다.[11] SplitTable은 입력 문자열의 모든 접두사에 대해 IsWord를 호출하고, 출력 문자열의 모든 접미사에 대해 자기 자신을 재귀적으로 호출할 수도 있습니다. 따라서 SplitTable의 "실행 시간"은 다음과 같은 무시무시해 보이는 점화식을 따릅니다

$$T(n) \leq \sum_{i=0}^{n-1} T(i) + O(n)$$

이것은 보기만큼 그렇게 어렵지 않으며, 특히 요령을 한 번 보고 나면 더욱 그렇습니다.

첫째, $O(n)$ 항을 알려지지 않은(그리고 결국 중요하지 않은) 어떤 상수 α에 대한 명시적인 식 αn으로 바꿉니다. 둘째, 알고리즘이 가능한 모든 재귀 호출을 실제로 수행한다고 보수적으로 가정합니다.[12] 그러면 다음과 같이 $T(n-1)$에 대한 점화식을 빼서 "전체 이력" 점화식을 "제한된 이력" 점화식으로 변환할 수 있습니다:

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} T(i) + \alpha n \\ T(n-1) &= \sum_{i=0}^{n-2} T(i) + \alpha(n-1) \\ \implies T(n) - T(n-1) &= T(n-1) + \alpha \end{aligned}$$

이 마지막 점화식은 $T(n) = 2T(n-1) + \alpha$로 단순화됩니다. 이 시점에서 우리는 $T(n) = O(2^n)$이라고 자신 있게 추측할 수 있습니다(또는 재귀 트리를 통해 유도하거나, 하노이의 탑 분석에서 기억해 낼 수도 있습니다). 실제로 이 상한은 원래의 전체 이력 점화식으로부터 귀납법으로 증명하기 어렵지 않습니다.

게다가 이 분석은 빈틈이 없습니다. 길이 n인 문자열을 분할하는 방법은 정확히 2^{n-1}가지입니다—각 입력 문자는 단어를 끝내거나 끝내지 않는데, 마지막 입력 문자만은 예외로 항상 마지막 단어를 끝냅니다. 최악의 경우 우리의 SplitTable 알고리즘은 이 2^{n-1}가지 가능성을 하나하나 모두 탐색합니다.

---

[11] 사실 IsWord가 *다항* 시간에 실행되기만 하면 SplitTable은 $O(2^n)$ 시간에 실행됩니다.

[12] 대부분의 글자 문자열은 영어 단어가 아니므로 이 가정은 영어 *단어* 분할에 대해서는 지나치게 보수적입니다. 그러나 영어 *단어*의 나열을 문법적으로 올바른 영어 *문장*으로 분할하는 비슷한 문제에 대해서는 보수적이지 *않습니다*. 예를 들어, 임의의 양의 정수 n에 대해 "buffalo"라는 단어를 n번 나열한 것, "police"라는 단어를 n번 나열한 것, 또는 "can"이라는 단어를 n번 나열한 것을 생각해 보십시오. (물랭루주에서는, 다른 춤들이 금속 원통에 보존할 수 있는 춤들이 교도소 화장실 쓰레기통에서 벌어지는 춤들을 해고할 기회를 가집니다.)

변형

이제 기본적인 재귀 패턴을 갖추었으므로, SubsetSum 문제에서 했던 것과 마찬가지로 이를 이용하여 분할 문제의 여러 가지 변형을 풀 수 있습니다. 여기서는 한 가지 예만 설명하겠습니다. 더 많은 변형은 연습 문제에서 다룹니다. 늘 그렇듯이, 우리 문제의 원래 입력은 배열 $A[1..n]$입니다.

문자열을 둘 이상의 방식으로 단어들의 나열로 분할할 수 있다면, 어떤 기준에 따라 *가장 좋은* 분할을 찾고 싶을 수 있습니다. 반대로 입력 문자열을 단어들로 분할할 수 없다면, 단순히 실패를 보고하는 대신 찾을 수 있는 가장 좋은 분할을 계산하고 싶을 수 있습니다. 이 두 목표를 모두 달성하기 위해, 문자열을 입력으로 받아 수치 값을 반환하는 두 번째 함수 Score를 사용할 수 있다고 가정합시다. 예를 들어 더 길거나 더 흔한 단어에는 더 높은 점수를, 더 짧거나 더 생소한 단어에는 더 낮은 점수를, 사소한 철자 오류에는 약간의 음수 점수를, 명백히 단어가 아닌 문자열에는 더 큰 음수 점수를 부여할 수 있습니다. 우리의 목표는 각 조각의 점수 합을 최대화하는 분할을 찾는 것입니다.

임의의 인덱스 i에 대해, $MaxScore(i)$를 접미사 $A[i..n]$의 분할이 가질 수 있는 최대 점수라고 합시다. 우리가 계산해야 하는 값은 $MaxScore(1)$입니다. 이 함수는 다음 점화식을 만족합니다.

$$MaxScore(i) = \begin{cases} 0 & \text{만약} i > n \\ \max\limits_{i \le j \le n} \big(\textsc{Score}(A[i..j]) + MaxScore(j+1) \big) & \text{그렇지 않으면} \end{cases}$$

이것은 본질적으로 앞에서 *Splittable*에 대해 세운 점화식과 같은 점화식입니다. 유일한 차이는 불리언 연산 ∨와 ∧가 수치 연산 max와 +로 바뀌었다는 점입니다.

2.6 최장 증가 부분 수열

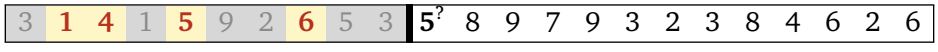
임의의 수열 S에 대해, S의 ***부분 수열***이란 S에서 0개 이상의 원소를 삭제하되 남은 원소들의 순서는 바꾸지 않고 얻은 또 다른 수열입니다. 부분 수열의 원소들이 S에서 연속해 있을 필요는 없습니다. 예를 들어, 어느 도시에서든 큰 도로를 따라 운전하면 신호등이 있는 교차로들의 *수열*을 지나가게 되지만, 실제로 멈춰야 하는 곳은 그 교차로들 가운데 신호등이 빨간불인 교차로들로 이루어진 *부분 수열*뿐입니다. 운이 아주 좋다면 한 번도 멈추지 않을 수도 있습니다. 즉, 빈 수열도 S의 부분 수열입니다. 반대로 운이 아주 나쁘다면 모든 교차로에서 멈춰야 할 수도 있습니다. 즉, S는 그 자체의 부분 수열입니다.

또 다른 예로, 문자열 BENT, ACKACK, SQUARING, SUBSEQUENT는 모두 문자열 SUBSEQUENCEBACKTRACKING의 부분 수열이며, 빈 문자열과 문자열 SUBSEQUENCEBACKTRACKING 전체도 마찬가지입니다. 하지만 문자열 QUEUE, EQUUS, TALLYHO는 부분 수열이 아닙니다. 원소들이 원래 수열에서 연속해 있는 부분 수열을 ***부분 문자열***이라고 합니다. 예를 들어 MASHER와 LAUGHTER는 둘 다 MANSLAUGHTER의 부분 수열이지만, 부분 문자열은 LAUGHTER뿐입니다.

이제 *정수*의 수열이 주어지고, 원소들이 증가하는 순서로 놓인 가장 긴 부분 수열을 찾아야 한다고 가정해 봅시다. 좀 더 구체적으로 말하면, 입력은 정수 배열 $A[1..n]$

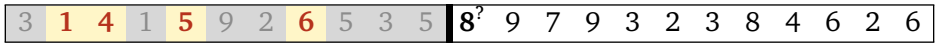
이고, 모든 k에 대해 $A[i_k] < A[i_{k+1}]$을 만족하는 인덱스 수열 $1 \le i_1 < i_2 < \cdots < i_\ell \le n$ 중 가능한 한 가장 긴 것을 계산해야 합니다.

이 ***최장 증가 부분 수열***을 만드는 한 가지 자연스러운 방법은 1부터 n까지 순서대로 각 인덱스 j에 대해 $A[j]$를 부분 수열에 포함할지 여부를 *결정*하는 것입니다. 이 결정 과정의 한가운데로 뛰어들어 보면, 다음과 같은 그림을 떠올릴 수 있습니다.



앞서 살펴본 텍스트 분할 예제에서와 마찬가지로, 검은 막대는 지금까지 내린 결정과 아직 처리하지 않은 입력 부분을 구분합니다. 이미 포함하기로 결정한 수는 강조 표시되어 굵게 나타나고, 이미 제외하기로 결정한 수는 회색으로 흐리게 표시됩니다. (포함하기로 결정한 수들이 증가하고 있다는 점에 주목하십시오!) 우리 알고리즘은 검은 막대 바로 뒤에 있는 수를 포함할지 말지를 결정해야 합니다.

이 예에서는 5를 절대 포함할 *수 없습니다*. 포함하면 선택한 수들이 더 이상 증가하는 순서가 아니게 되기 때문입니다. 그러니 다음 결정으로 건너뜁시다:

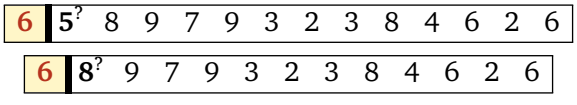


이제 8을 *포함할 수는* 있지만, *포함해야 하는지*는 분명하지 않습니다. 우리의 백트래킹 알고리즘은 "영리하게" 굴려고 하기보다 단순한 무차별 대입을 사용할 것입니다.

- 먼저 8을 *잠정적*으로 포함하고, 나머지 결정은 재귀 요정에게 맡기십시오.
- 그런 다음 8을 *잠정적*으로 제외하고, 나머지 결정은 재귀 요정이 내리도록 맡기십시오.

두 선택 중 더 긴 증가 부분 수열로 이어지는 쪽이 올바른 선택입니다. (이는 SUBSET-SUM을 풀 때 사용했던 것과 정확히 같은 재귀 패턴입니다.)

이제 핵심 질문을 살펴보겠습니다. *과거의 결정에 대해 무엇을 기억해야 합니까?* 결과로 얻는 부분 수열이 증가하는 순서일 때에만 $A[j]$를 포함할 수 있습니다. $A[1..j-1]$에서 이전에 선택한 수들이 증가하는 순서라고 (귀납적으로!) 가정하면, $A[j]$가 $A[1..j-1]$에서 마지막으로 선택한 수보다 클 때, 그리고 오직 그때에만 $A[j]$를 포함할 수 있습니다. 따라서 과거에 대해 필요한 유일한 정보는 ***지금까지 마지막으로 선택한 수***입니다. 이제 필요 없는 것을 모두 지워서 그림을 수정할 수 있습니다:



따라서 우리의 재귀 전략이 *실제로* 풀고 있는 문제는 다음과 같습니다:

> 정수 *prev*와 배열 $A[1..n]$이 주어졌을 때, 모든 원소가 *prev*보다 큰 A의 최장 증가 부분 수열을 찾으십시오.

늘 그렇듯이 우리의 재귀 전략에는 기저 사례가 필요합니다. 현재 전략은 배열의 끝에 도달하면 더 이상 작동하지 않습니다. 고려할 "다음 수"가 없기 때문입니다. 그러나 빈 배열에는 부분 수열이 정확히 하나 있는데, 바로 *빈* 수열입니다. 공허하게

참인 명제로서, 빈 수열의 모든 원소는 여러분이 원하는 어떤 값보다도 크며, 빈 수열의 모든 원소 쌍은 증가하는 순서로 나타납니다. 따라서 빈 배열의 최장 증가 부분 수열의 길이는 0입니다.

그 결과로 얻은 재귀 알고리즘은 다음과 같습니다:

```
LISbigger(prev, A[1..n]):
  if n = 0
      return 0
  else if A[1] ≤ prev
      return LISbigger(prev, A[2..n])
  else
      skip ← LISbigger(prev, A[2..n])
      take ← LISbigger(A[1], A[2..n]) + 1
      return max{skip, take}
```

좋습니다. 하지만 호출 스택을 통해 배열을 주고받는 것은 비용이 크다는 점을 기억하십시오. 배열 $A[1..n]$이 전역 변수라고 가정하고, 모든 것을 배열 인덱스로 다시 표현해 봅시다. 정수 *prev*는 일반적으로 배열 원소 $A[i]$이고, 남은 배열은 항상 원래 입력 배열의 접미사 $A[j..n]$입니다. 따라서 우리의 재귀 *문제*를 다음과 같이 다시 정식화할 수 있습니다:

> 두 인덱스 i와 $j (i < j)$가 주어질 때, $A[j..n]$의 부분 수열 가운데 모든 원소가 $A[i]$보다 큰 최장 증가 부분 수열을 구하십시오.

LISbigger(i, j)를 모든 원소가 $A[i]$보다 큰 $A[j..n]$의 최장 증가 부분 수열의 *길이*라고 합시다. 우리의 재귀적 전략으로부터 다음 점화식을 얻습니다:

$$LISbigger(i,j) = \begin{cases} 0 & \text{만약} j > n \\ LISbigger(i, j+1) & \text{만약} A[i] \geq A[j] \\ \max\left\{ \begin{array}{c} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{array} \right\} & \text{그렇지 않으면} \end{cases}$$

또는 의사코드가 더 편하시다면 다음과 같습니다.

```
LISbigger(i, j):
  if j > n
      return 0
  else if A[i] ≥ A[j]
      return LISbigger(i, j + 1)
  else
      skip ← LISbigger(i, j + 1)
      take ← LISbigger(j, j + 1) + 1
      return max{skip, take}
```

마지막으로, 이 재귀 전략을 원래 문제, 즉 *다른 제약 조건이 없는* 배열의 최장 증가 부분 수열을 찾는 문제와 연결해야 합니다. 가장 간단한 방법은 배열의 맨 앞에 인위적인 보초 값 $-\infty$를 추가하는 것입니다.

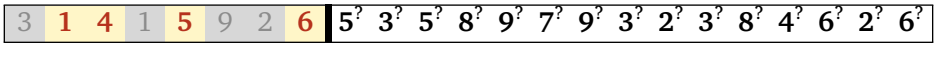
```
LIS(A[1..n]):
  A[0] ← −∞
  return LISbigger(0, 1)
```

LISbigger의 실행 시간은 하노이 점화식 $T(n) \le 2T(n-1) + O(1)$을 만족하며, 이는 늘 그렇듯 $T(n) = O(2^n)$임을 뜻합니다. 사실 이 실행 시간은 전혀 놀랄 일이 아닙니다. 최악의 경우 이 알고리즘은 입력 배열의 2^n개 부분 수열을 하나하나 모두 검사하기 때문입니다.

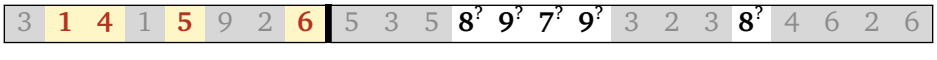
2.7 최장 증가 부분 수열, 두 번째 접근

최장 증가 부분 수열을 찾는 데 쓸 수 있는 백트래킹 전략은 이것만이 아닙니다. *입력* 수열을 한 번에 원소 하나씩 살펴보는 대신, *출력* 수열을 한 번에 원소 하나씩 구성해 볼 수도 있습니다. 즉, "$A[i]$가 출력 수열의 다음 원소입니까?" 라고 묻는 대신, "출력 수열에 다음 원소가 있다면, 그것은 어디에 있습니까?" 라고 직접 물을 수 있습니다.

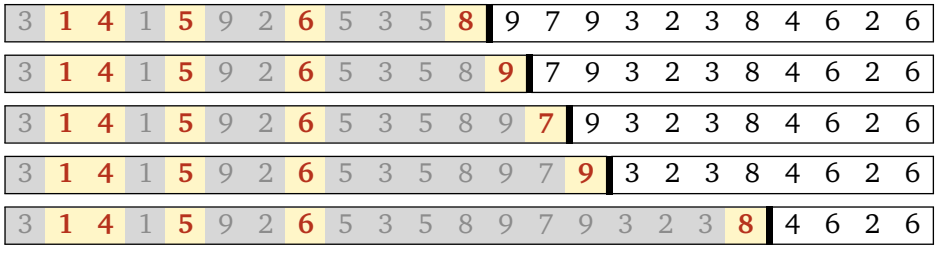
이 전략을 진행하는 도중의 한 시점을 살펴보면, 다음과 같은 상황을 마주할 수 있습니다. 검은 막대 바로 왼쪽에 있는 6을 출력 수열에 포함하기로 방금 결정했고, 이제 막대 오른쪽의 원소 중 어느 것을 다음에 포함할지 정해야 한다고 가정합시다.



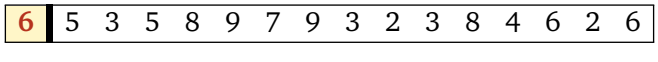
물론 오른쪽에는 6보다 큰 수만 포함할 수 있습니다. 그렇지 않으면 출력 수열이 증가하지 않게 됩니다.



하지만 그 더 큰 수들 중 *어느* 것이 최선의 선택인지 우리는 전혀 알 수 없으며, 최선의 선택을 영리하게 *알아내려고* 애쓰는 것은 너무 많은 수고가 들 뿐만 아니라 어차피 곤란한 상황만 초래할 뿐입니다. 대신 우리는 무차별 대입으로 모든 가능성을 열거하고, 각각을 평가하는 일은 재귀 요정에게 맡깁니다.



다음 원소로 고려할 수 있는 수들의 부분집합은 *오직* 마지막으로 포함하기로 결정한 수에만 의존합니다. 따라서 막대 왼쪽에 있는 모든 것을 버리되 마지막으로 포함하기로 결정한 수는 *제외함*으로써 결정 과정에 대한 그림을 단순화할 수 있습니다.



남은 수열은 원래 입력 배열의 접미사일 뿐입니다. 따라서 입력 배열 $A[1..n]$을 전역 변수로 생각하면, 우리의 재귀 문제를 인덱스를 사용하여 다음과 같이 형식적으로 표현할 수 있습니다:

> 인덱스 i가 주어졌을 때, $A[i]$로 시작하는 $A[i..n]$의 최장 증가 부분 수열을 찾으십시오.

LISfirst(i)를 $A[i]$로 시작하는 $A[i..n]$의 최장 증가 부분 수열의 길이라고 합시다. 이제 우리의 재귀 백트래킹 전략을 다음과 같은 재귀적 정의로 정식화할 수 있습니다:

$$LISfirst(i) = 1 + \max\{LISfirst(j) \mid j > i \text{그리고} A[j] > A[i]\}$$

자연수의 집합을 다루고 있으므로 $\max\emptyset = 0$으로 정의합니다. 그러면 모든 $j > i$에 대해 $A[j] \le A[i]$이면 자동으로 *LISfirst*$(i) = 1$이 성립하며, 특히 *LISfirst*$(n) = 1$입니다. 이것들이 우리 점화식의 기저 사례입니다.

이 재귀적 정의는 다음과 같이 의사코드로 표현할 수도 있습니다.

```
LISFIRST(i):
  best ← 0
  for j ← i + 1 to n
      if A[j] > A[i]
          best ← max{best, LISFIRST(j)}
  return 1 + best
```

마지막으로, 이 재귀 알고리즘을 원래 문제—첫 번째 원소를 모르는 상태에서 최장 증가 부분 수열을 찾는 문제—와 다시 연결해야 합니다. 이때 쓸 수 있는 자연스러운 접근법 하나는 가능한 모든 첫 번째 원소를 무차별 대입으로 시도하는 것입니다. 이와 동등하게, 배열의 맨 앞에 보초 원소 $-\infty$를 추가하고, 그 보초로 시작하는 최장 증가 부분 수열을 찾은 다음, 마지막으로 보초를 무시할 수도 있습니다.

```
LIS(A[1..n]):
  best ← 0
  for i ← 1 to n
      best ← max{best, LISFIRST(i)}
  return best
```

```
LIS(A[1..n]):
  A[0] ← −∞
  return LISFIRST(0) − 1
```

2.8 최적 이진 탐색 트리

마지막 예제는 재귀 백트래킹과 분할 정복 전략을 결합합니다. 이진 탐색 트리에서 성공적인 탐색의 실행 시간은 대상 노드의 조상 수에 비례한다는 사실을 상기하십시오.[13] 그 결과, 최악의 경우 탐색 시간은 트리의 깊이에 비례합니다. 따라서 최악의 경우 탐색 시간을 최소화하려면 트리의 높이가 가능한 한 작아야 합니다. 이 기준에 따르면 이상적인 트리는 완벽하게 균형 잡힌 트리입니다.

[13] 노드 v의 *조상*은 그 노드 자신이거나 v의 부모의 조상입니다. v의 *진(proper)* 조상은 v의 부모이거나 v의 부모의 진 조상입니다.

그러나 이진 탐색 트리의 많은 응용에서는 한 번의 탐색에 드는 최악의 경우 비용보다 여러 번의 탐색에 드는 총비용을 최소화하는 것이 더 중요합니다. x가 y보다 더 자주 탐색되는 대상이라면, 트리의 전체 깊이가 늘어나더라도 x의 깊이가 y의 깊이보다 작은 트리를 만들어 시간을 절약할 수 있습니다. 일부 항목이 다른 항목보다 훨씬 더 자주 탐색된다면 완벽하게 균형 잡힌 트리는 최선의 선택이 *아닙니다*. 실제로 깊이가 $\Omega(n)$인 완전히 불균형한 트리가 오히려 최선의 선택일 수도 있습니다!

이러한 상황에서 다음 문제를 생각할 수 있습니다. *키*의 정렬된 배열 $A[1..n]$과 이에 대응하는 ***접근 빈도***의 배열 $f[1..n]$이 주어졌다고 가정합시다. 각 키 $A[i]$에 대해 정확히 $f[i]$번의 탐색이 이루어진다고 가정할 때, 우리가 할 일은 *전체* 탐색 시간을 최소화하는 이진 탐색 트리를 만드는 것입니다.

이 문제를 어떻게 풀지 생각하기 전에, 먼저 우리가 최적화하려는 함수에 대한 좋은 재귀적 정의부터 세워야 합니다! 이제 n개의 노드를 가진 이진 탐색 트리 T도 주어졌다고 가정합시다. $v_1, v_2, \ldots, v_n$을 정렬된 순서대로 번호를 매긴 T의 노드라고 하여, 각 노드 v_i가 대응하는 키 $A[i]$를 저장하도록 합시다. 그러면 상수 배를 무시할 때, 모든 이진 탐색을 수행하는 데 드는 총비용은 다음 식으로 주어집니다:

$$Cost(T, f[1..n]) := \sum_{i=1}^{n} f[i] \cdot \#T\text{에서 } v_i\text{의 조상들} \qquad (*)$$

이제 v_r가 T의 루트라고 가정합시다. 정의에 따라 v_r는 T의 모든 노드의 조상입니다. $i < r$이면, 루트를 제외한 v_i의 모든 조상은 T의 왼쪽 부분 트리에 있습니다. 마찬가지로 $i > r$이면, 루트를 제외한 v_i의 모든 조상은 T의 오른쪽 부분 트리에 있습니다. 따라서 비용 함수를 다음과 같이 세 부분으로 나눌 수 있습니다:

$$\begin{aligned} Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] &+ \sum_{i=1}^{r-1} f[i] \cdot \#left(T)\text{에서 } v_i\text{의 조상} \\ &+ \sum_{i=r+1}^{n} f[i] \cdot \#right(T)\text{에서 } v_i\text{의 조상들} \end{aligned}$$

두 번째와 세 번째 합은 $Cost(T, f[1..n])$에 대한 원래 정의 $(*)$와 정확히 같은 꼴입니다. 이제 간단히 대입하면 $Cost$에 대한 점화식을 얻습니다:

$$\begin{aligned} Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] &+ Cost(left(T), f[1..r-1]) \\ &+ Cost(right(T), f[r+1..n]) \end{aligned}$$

이 점화식의 기저 사례는 늘 그렇듯이 $n = 0$이며, 빈 트리에서 아무 탐색도 수행하지 않는 비용은 0입니다.

이제 우리가 할 일은 이 비용 함수를 최소화하는 트리 T_{opt}를 계산하는 것입니다. 어떤 마법 같은 방법으로 T_{opt}의 루트가 v_r이라는 사실을 알고 있다고 가정해 봅시다. 그러면 $Cost(T, f)$의 재귀적 정의에 의해 왼쪽 부분 트리 $left(T_{\text{opt}})$는 키 $A[1..r-1]$과 접근 빈도 $f[1..r-1]$에 대한 최적 탐색 트리여야 함이 곧바로 따라 나옵니다. 마찬가지로 오른쪽 부분 트리 $right(T_{\text{opt}})$는 키 $A[r+1..n]$과 접근 빈도 $f[r+1..n]$에

대한 최적 탐색 트리여야 합니다. **루트에 저장할 올바른 키를 고르기만 하면, 나머지 최적 트리는 재귀 요정이 구성해 줄 것입니다.**

더 일반적으로, 빈도 구간 $f[i..k]$에 대한 최적 탐색 트리의 총비용을 $OptCost(i,k)$로 나타냅시다. 이 함수는 다음 점화식을 따릅니다.

$$OptCost(i,k) = \begin{cases} 0 & \text{만약} i > k \\ \displaystyle\sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{그렇지 않으면} \end{cases}$$

기저 사례는 공집합에서 0번의 탐색을 수행하는 데 드는 최소 비용이 0임을 올바르게 나타냅니다! 우리의 원래 문제는 $OptCost(1,n)$을 계산하는 것입니다.

이 재귀적 정의는 $OptCost(1,n)$을 계산하는 재귀 백트래킹 알고리즘으로 기계적으로 옮길 수 있습니다. 당연하게도 이 알고리즘의 실행 시간은 지수 시간입니다. 다음 장에서 실행 시간을 다항 시간으로 줄이는 방법을 살펴볼 것이므로, 정확한 실행 시간을 계산하는 것은 큰 의미가 없습니다…

♥분석

…물론 그런 것을 즐기신다면 이야기가 다릅니다. 그저 재미 삼아, 이 백트래킹 알고리즘이 실제로 얼마나 느린지 알아봅시다. 실행 시간은 다음 점화식을 만족합니다

$$T(n) = \sum_{k=1}^{n} \big(T(k-1) + T(n-k)\big) + O(n).$$

$O(n)$ 항은 전체 탐색 횟수 $\sum_{i=1}^{n} f[i]$를 계산하는 데서 나옵니다. 네, 정말 보기 흉한 점화식입니다. 하지만 앞에서 사용한 것과 똑같은 뻘셈 기법으로 풀 수 있습니다. $O()$ 표기를 명시적인 상수로 바꾸고, 항들을 다시 묶어 동일한 항끼리 모은 다음, 합을 없애기 위해 $T(n-1)$에 대한 점화식을 빼고, 다시 한번 항들을 묶습니다.

$$\begin{aligned} T(n) &= 2\sum_{k=0}^{n-1} T(k) + \alpha n \\ T(n-1) &= 2\sum_{k=0}^{n-2} T(k) + \alpha(n-1) \\ T(n) - T(n-1) &= 2T(n-1) + \alpha \\ T(n) &= 3T(n-1) + \alpha \end{aligned}$$

어, 막상 보니 그렇게 나쁘지 않습니다. 재귀 트리 방법을 쓰면 $\boldsymbol{T(n) = O(3^n)}$이라는 해를 곧바로 얻을 수 있습니다(아니면 그냥 추측한 뒤 귀납법으로 확인해도 됩니다).

이 분석은 우리의 재귀 알고리즘이 가능한 모든 이진 탐색 트리를 살펴보지 *않는다*는 것을 뜻합니다! 정점이 n개인 이진 탐색 트리의 개수는 다음 점화식을 만족합니다

$$N(n) = \sum_{r=1}^{n-1} \big(N(r-1) \cdot N(n-r)\big),$$

이 점화식의 닫힌 형태의 해는 $N(n) = \Theta(4^n/\sqrt{n})$입니다. (아니요, 이는 자명하지 않습니다.) 우리 알고리즘은 각 루트마다 최적의 왼쪽 부분 트리와 오른쪽 부분 트리를 *독립적*으로 탐색하므로 시간을 상당히 절약합니다. 이진 탐색 트리를 전부 열거한다면 왼쪽 부분 트리와 오른쪽 부분 트리의 가능한 모든 *쌍*을 고려해야 합니다. $N(n)$에 대한 점화식에 곱이 나타나는 것은 이 때문입니다.

연습문제

1. SubsetSum 문제를 다음과 같이 일반화한 문제들에 대한 재귀 알고리즘을 설명하십시오.

   (a) 양의 정수로 이루어진 배열 $X[1..n]$과 정수 T가 주어질 때, 원소들의 합이 T가 되는 X의 부분집합의 *개수*를 계산하십시오.

   (b) 양의 정수로 이루어진 두 배열 $X[1..n]$과 $W[1..n]$, 그리고 정수 T가 주어집니다. 여기서 각 $W[i]$는 대응하는 원소 $X[i]$의 *가중치*를 나타냅니다. 원소들의 합이 T인 X의 부분집합 중 *가중치가 최대*인 것을 계산하십시오. 합이 T가 되는 X의 부분집합이 없으면 알고리즘은 $-\infty$를 반환해야 합니다.

2. 텍스트 분할 문제의 다음 변형들을 푸는 재귀 알고리즘을 설명하십시오. 문자 배열을 입력으로 받아 그 문자열이 "단어"일 때, 그리고 그때에만 True를 반환하는 서브루틴 IsWord가 있다고 가정하십시오.

   (a) 문자 배열 $A[1..n]$이 주어질 때, A를 단어들로 분할하는 방법의 수를 계산하십시오. 예를 들어 문자열 ARTISTOIL이 주어지면 ARTIST·OIL과 ART·IS·TOIL의 두 가지 분할이 있으므로 알고리즘은 2를 반환해야 합니다.

   (b) 문자 배열 $A[1..n]$과 $B[1..n]$이 주어졌을 때, A와 B를 같은 인덱스에서 단어들로 분할할 수 있는지 판정하십시오. 예를 들어, 문자열 BOTHEARTHANDSATURNSPIN과 PINSTARTRAPSANDRAGSLAP은 다음과 같이 같은 인덱스에서 단어들로 분할할 수 있습니다:

   BOT·HEART·HAND·SAT·URNS·PIN
   PIN·START·RAPS·AND·RAGS·LAP

   (c) 문자로 이루어진 두 배열 $A[1..n]$과 $B[1..n]$이 주어질 때, A와 B를 같은 인덱스에서 단어들로 분할할 수 있는 서로 다른 방법의 수를 계산하십시오.

3. 정수 n에 대한 *덧셈 사슬*은 1로 시작하여 n으로 끝나는 증가하는 정수 수열로서, 첫 번째 항 이후의 각 항이 그보다 앞선 두 항의 합인 수열입니다. 더 엄밀하게 말하면, 정수 수열 $x_0 < x_1 < x_2 < \cdots < x_\ell$이 n에 대한 덧셈 사슬일 필요충분조건은 다음과 같습니다.

   - $x_0 = 1$,
   - $x_\ell = n$이고,

- 모든 인덱스 $k > 0$에 대해 $x_k = x_i + x_j$를 만족하는 인덱스 $i \le j < k$가 존재합니다.

덧셈 사슬의 길이 ℓ은 원소의 개수에서 1을 뺀 값이며, 첫 번째 항목은 굳이 세지 않습니다. 예를 들어 $\langle 1, 2, 3, 5, 10, 20, 23, 46, 92, 184, 187, 374\rangle$은 374에 대한 길이 11의 덧셈 사슬입니다.

(a) 주어진 양의 정수 n에 대해 최소 길이 덧셈 사슬을 계산하는 재귀 백트래킹 알고리즘을 설명하십시오. 여러분 스스로의 호기심을 채우려는 경우가 아니라면, 알고리즘의 실행 시간을 분석하거나 최적화하려 ***하지 마십시오***. 실행 시간이 n에 대해 지수적이더라도 올바른 알고리즘이면 만점을 받기에 충분합니다. *[힌트: 이 문제는 텍스트 분할보다는 n 퀸 문제에 훨씬 더 가깝습니다.]*

♥(b) 주어진 양의 정수 n에 대한 최소 길이 덧셈 사슬을 *n에 대해 준지수적인 시간에* 계산하는 재귀 백트래킹 알고리즘을 설명하십시오. *[힌트: 고대 이집트의 밧줄 매듭꾼들, 인더스강 유역의 운율학자들, 그리고 러시아 농부들이 얻은 결과가 도움이 될 수 있습니다.]*

4. (a) $A[1..m]$과 $B[1..n]$을 임의의 두 배열이라고 합시다. A와 B의 *공통 부분 수열*은 A의 부분 수열이면서 동시에 B의 부분 수열인 수열입니다. A와 B의 *최장* 공통 부분 수열의 길이를 구하는 함수 $lcs(A, B)$의 간단한 재귀적 정의를 제시하십시오.

(b) $A[1..m]$과 $B[1..n]$을 임의의 두 배열이라고 합시다. A와 B의 *공통 초수열*이란 A와 B를 모두 부분 수열로 포함하는 또 다른 수열입니다. A와 B의 *최단* 공통 초수열의 길이를 구하는 함수 $scs(A, B)$에 대한 간단한 재귀적 정의를 제시하십시오.

(c) 숫자들의 수열 $X[1..n]$에 대해, $1 < i < n$인 인덱스 i가 존재하여 접두사 $X[1..i]$는 증가하고 접미사 $X[i..n]$은 감소할 때, 이 수열을 ***바이토닉***이라고 부릅니다. 임의의 정수 배열 A의 가장 긴 바이토닉 부분 수열의 길이를 반환하는 함수 $lbs(A)$에 대한 간단한 재귀적 정의를 제시하십시오.

(d) 모든 짝수 i에 대해 $X[i] < X[i+1]$이고 모든 홀수 i에 대해 $X[i] > X[i+1]$이면, 수열 $X[1..n]$을 *진동하는* 수열이라고 부릅시다. 정수로 이루어진 임의의 배열 A에서 가장 긴 진동 부분 수열의 길이를 구하는 함수 $los(A)$의 간단한 재귀적 정의를 제시하십시오.

(e) 임의의 정수 배열 A의 최단 진동 초수열의 길이를 반환하는 함수 $sos(A)$에 대한 간단한 재귀적 정의를 제시하십시오.

(f) 모든 i에 대해 $2 \cdot X[i] < X[i-1] + X[i+1]$이면 수열 $X[1..n]$을 *볼록하다*고 합시다. 임의의 정수 배열 A의 가장 긴 볼록 부분 수열의 길이를 반환하는 함수 $lxs(A)$에 대한 간단한 재귀적 정의를 제시하십시오.

5. 다음 각 문제에서 입력은 두 배열 $X[1..k]$와 $Y[1..n]$으로 이루어지며, 여기서 $k \le n$입니다.

(a) X가 Y의 부분 수열인지 판별하는 재귀 백트래킹 알고리즘을 설명하십시오. 예를 들어, 문자열 PPAP는 문자열 PENPINEAPPLEAPPLEPEN의 부분 수열입니다.

(b) X가 더 이상 부분 수열이 되지 않도록 Y에서 제거해야 하는 기호의 최소 개수를 구하는 재귀 백트래킹 알고리즘을 설명하십시오. 이는 X의 초수열이 *아닌* Y의 가장 긴 부분 수열을 찾는 것과 같습니다. 예를 들어 문자열 PENPINEAPPLEAPPLEPEN에서 기호 두 개를 제거하면 문자열 PPAP는 더 이상 부분 수열이 아닙니다.

♥(c) X가 Y 안에서 서로 *겹치지 않는* 두 부분 수열로 나타나는지 판별하는 재귀 백트래킹 알고리즘을 설명하십시오. 예를 들어 문자열 PPAP는 문자열 PENPINEAPPLEAPPLEPEN 안에서 서로 겹치지 않는 두 부분 수열로 나타납니다.

호기심을 채우려는 경우가 아니라면 알고리즘의 실행 시간을 분석하지 ***마십시오***. 세 알고리즘 모두 지수 시간에 실행됩니다. 이는 나중에 개선할 것이므로 정확한 실행 시간은 그다지 중요하지 않습니다.

6. 이 문제에서는 추가적인 균형 제약 조건을 만족하는 최적 이진 탐색 트리의 비용을 구하는 백트래킹 알고리즘을 설계해야 합니다. 입력은 탐색 키의 정렬된 배열 $A[1..n]$과 빈도수의 배열 $f[1..n]$으로 이루어지며, 여기서 $f[i]$는 $A[i]$에 대한 탐색 횟수입니다. 이는 2.8절에서 설명한 비용 함수와 정확히 같습니다. 하지만 이제 여러분이 할 일은 몇 가지 추가 제약 조건을 만족하는 최적 트리를 계산하는 것입니다.

(a) ***AVL 트리***는 최초로 고안된 스스로 균형을 유지하는 균형 이진 탐색 트리로, 1962년 게오르기 아델손-벨스키(Georgy Adelson-Velsky)와 예브게니 란디스(Evgenii Landis)가 처음 기술하였습니다. AVL 트리는 모든 노드 v에 대해 v의 왼쪽 부분 트리의 높이와 v의 오른쪽 부분 트리의 높이가 최대 1만큼 차이 나는 이진 탐색 트리입니다.

주어진 탐색 키와 빈도의 집합에 대해 최적 AVL 트리를 구성하는 재귀 백트래킹 알고리즘을 설명하십시오.

(b) ***대칭 이진 B-트리***(symmetric binary B-tree)는 또 다른 자가 균형 이진 트리로, 1972년 루돌프 바이어(Rudolf Bayer)가 처음 기술했습니다. 1978년 레오 기바스(Leo Guibas)와 밥 세지윅(Bob Sedgwick)이 이를 다소 단순하게 재정식화한 이후로는 ***레드-블랙 트리***(red-black tree)라는 이름으로 더 잘 알려져 있습니다. 레드-블랙 트리는 다음과 같은 제약 조건이 추가된 이진 탐색 트리입니다.

- 모든 노드는 빨간색이거나 검은색입니다.
- 모든 빨간 노드의 부모는 검은 노드입니다.
- 루트에서 리프까지의 모든 경로에는 같은 개수의 검은색 노드가 있습니다.

주어진 탐색 키 집합과 각 키의 빈도에 대해 최적 레드-블랙 트리를 구성하는 재귀 백트래킹 알고리즘을 설명하십시오.

(c) ***AA 트리***는 1993년 Arne Andersson이 제안하였고, 2000년 Mark Allen Weiss가 약간 단순화하면서 이름을 붙였습니다. 2006년 밥 세지윅(Bob Sedgewick)이 (다른 재균형 알고리즘을 사용하여) 대칭적으로 재정식화한 이후 AA 트리는 *왼쪽으로 기운 레드-블랙 트리*라고도 불립니다. AA 트리는 다음과 같은 제약 조건이 하나 더 붙은 레드-블랙 트리입니다.

- 빨간색인 왼쪽 자식은 없습니다.[14]

주어진 탐색 키와 빈도의 집합에 대해 최적의 AA 트리를 구성하는 재귀 백트래킹 알고리즘을 설명하십시오.

여러분 자신의 호기심을 채우려는 경우가 아니라면 알고리즘의 실행 시간을 분석하지 ***마십시오***. 세 알고리즘 모두 지수 시간에 실행됩니다. 이는 나중에 개선할 것이므로 정확한 실행 시간은 그다지 중요하지 않습니다.

백트래킹 연습 문제를 더 풀어 보려면 다음 장을 참고하십시오!

[14] Sedgwick의 재정식화에서는 빨간색인 오른쪽 자식이 없어야 합니다. 뭐, 아무래도 상관없습니다. Andersson과 Sedgwick은 달걀을 어느 쪽 끝부터 먼저 먹어야 하는지에 대해서는 이상하리만치 아무 말도 하지 않습니다.

Potes enim videre in hac margine, qualiter hoc operati fuimus, scilicet quod iunximus primum numerum cum secundo, videlicet 1 cum 2; et secundum cum tercio; et tercium cum quarto; et quartum cum quinto, et sic deinceps….
[여기 여백에서 우리가 이것을 어떻게 계산했는지 볼 수 있습니다. 분명히 우리는 첫 번째 수를 두 번째 수와, 즉 1을 2와 합쳤고, 두 번째 수를 세 번째 수와, 세 번째 수를 네 번째 수와, 네 번째 수를 다섯 번째 수와 합쳤으며, 이런 식으로 계속하였습니다….]

— 레오나르도 피사노, *Liber Abaci* (1202)

과거를 기억하지 못하는 사람들은 과거를 되풀이할 운명에 처합니다.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

배움의 경험이 무엇인지 아십니까?
배움의 경험이란 이렇게 말해 주는 그런 것입니다.
"방금 하신 그 일 아시지요? 그거, 하지 마십시오."

— 더글러스 애덤스, *The Salmon of Doubt* (2002)

3

동적 계획법

3.1 마트라브리타(Mātrāvṛtta)

재귀의 가장 초기 사례 가운데 하나는 2000년도 더 전에 인도에서 시의 운율, 곧 운율학을 연구하는 과정에서 나타났습니다. 고전 산스크리트 시는 음절(*akṣara*)을 두 종류로 구분합니다. 바로 *가벼운* 음절(*laghu*)과 *무거운* 음절(*guru*)입니다. *mātrāvṛtta* 또는 *mātrāchandas* 등으로 불리는 한 부류의 운율에서는 시의 각 행이 정해진 수의 "박"(*mātrā*)으로 이루어지며, 가벼운 음절은 한 박, 무거운 음절은 두 박 동안 이어집니다. *mātrā-vṛtta*에 대한 체계적인 연구는 학자 Piṅgala가 기원전 600년에서 기원전 200년 사이에 쓴 *Chandaḥśāstra*까지 거슬러 올라갑니다. Piṅgala는 4박 운율이 ——, — • •, • — •, • • —, • • • •의 정확히 다섯 가지임을 관찰했습니다. (여기서 각 "—"는 긴 음절을, 각 "•"는 짧은 음절을 나타냅니다.)[1]

Piṅgala의 문헌은 주어진 박자 수를 갖는 운율의 개수를 세는 체계적인 규칙을 *암시*하지만,[2] 그 규칙이 명시적으로 서술되기까지는 약 천 년이 걸렸습니다. 7세기에

[1] 모스 부호에서 "dah"는 "dit"보다 세 배 길게 이어지지만, 각 "dit" 또는 "dah" 뒤에는 "dit"과 같은 길이의 쉼이 따라옵니다. 따라서 각 "dit-쉼"은 *laghu akṣara*이고 각 "dah-쉼"은 *guru akṣara*이며, 부호의 길이가 네 *mātrā*인 문자는 정확히 다섯 개(M, D, R, U, H)입니다.

[2] *Chandaḥśāstra*에는 주어진 *음절* 수를 갖는 모든 운율을 나열하는 체계적인 규칙이 *두 가지* 들어 있으며, 이는 대략 수를 이진법으로 (그리스인처럼) 왼쪽에서 오른쪽으로 쓰거나 (이집트인처럼) 오른쪽에서

Virahāṇka라는 또 다른 인도 학자가 Piṅgala의 저작에 대한 주석을 썼는데, 그 주석에서 그는 n박자 운율의 개수가 $(n-2)$박자 운율의 개수와 $(n-1)$박자 운율의 개수의 합이라는 사실을 관찰했습니다. 좀 더 현대적인 표기법으로 나타내면, Virahāṇka의 관찰은 n박자 운율의 총 개수 $M(n)$에 대한 다음 점화식을 함의합니다:

$$M(n) = M(n-2) + M(n-1)$$

$M(0) = 1$이고(빈 운율은 하나뿐입니다) $M(1) = 1$이라는 것(한 박자짜리 운율은 짧은 음절 하나로 이루어진 것 하나뿐입니다)은 어렵지 않게 알 수 있습니다.

같은 점화식은 Virahāṇka로부터 약 500년 뒤 유럽에서 다시 등장했습니다. 피사의 레오나르도가 1202년에 쓴 논저 *Liber Abaci*에서였으며, 이 책은 "산법(algorism)"을 다룬 초기 유럽 저작 가운데 가장 큰 영향을 끼친 책 중 하나입니다. 스티글러의 명명 법칙에 완벽하게 들어맞게도,[3] 오늘날의 *피보나치 수*는 Virahāṇka의 점화식을 사용하되 기저 사례만 달리하여 정의됩니다:

$$F_n = \begin{cases} 0 & \text{만약} n = 0 \\ 1 & \text{만약} n = 1 \\ F_{n-1} + F_{n-2} & \text{그렇지 않으면} \end{cases}$$

특히, 모든 n에 대해 $M(n) = F_{n+1}$입니다.

백트래킹은 느릴 수 있습니다

피보나치 수의 재귀적 정의로부터 피보나치 수를 계산하는 재귀 알고리즘을 곧바로 얻을 수 있습니다. 다음은 같은 알고리즘을 의사코드로 작성한 것입니다:

```
RecFibo(n):
  if n = 0
      return 0
  else if n = 1
      return 1
  else
      return RecFibo(n − 1) + RecFibo(n − 2)
```

안타깝게도 이 단순한 재귀 알고리즘은 끔찍하게 느립니다. 재귀 호출을 제외하면 전체 알고리즘에 필요한 것은 상수 개의 단계, 즉 비교 한 번과 경우에 따라 덧셈 한

왼쪽으로 쓰는 방식에 해당합니다. 같은 문헌에는 반복 제곱으로 2^n(n개의 음절을 갖는 운율의 개수)을 계산하는 재귀 알고리즘과, (논란의 여지는 있지만) 이항 계수(짧은 음절 k개를 포함하여 전체 n개의 음절을 갖는 운율의 개수)를 계산하는 재귀 알고리즘도 들어 있습니다.

[3] "*어떤 과학적 발견도 최초 발견자의 이름을 따서 명명되지 않는다.*" 이 법칙에 이름을 붙인 1980년 논문에서 통계학자 스티븐 스티글러는 이 법칙을 처음 제안한 사람이 사회학자 로버트 K. 머튼이라고 농담 삼아 주장했습니다. 그러나 비슷한 말은 이미 이전에도 있었습니다. 1970년대에는 블라디미르 아르놀트가("발견이 올바른 사람의 공으로 돌아가는 일은 드물다."), 1968년에는 칼 보이어가("역사의 뮤즈 클리오는 정리에 이름을 붙이는 데 종종 변덕스럽다!"), 1917년에는 앨프리드 노스 화이트헤드가("중요한 것은 모두 그것을 발견하지 않은 누군가가 이미 말한 바 있다."), 그리고 심지어 1966년에는 스티븐의 아버지 조지 스티글러까지("이론에 올바른 사람의 이름이 붙은 사례를 만나게 된다면, 그 사실은 반드시 주목받을 것이다.") 그런 말을 했습니다. 이 책에서 우리는 스티글러의 법칙을 보여 주는 다른 예를 *많이* 보게 될 것입니다.

번뿐입니다. RecFibo에 대한 재귀 호출의 횟수를 $T(n)$이라고 합시다. 이 함수는 다음 점화식을 만족합니다

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + 1,$$

이는 피보나치 수 자체의 점화식과 놀라울 정도로 비슷해 보입니다! $T(n)$의 처음 몇 개 값을 적어 보면 닫힌 형태의 해 $\boldsymbol{T(n) = 2F_{n+1} - 1}$을 짐작할 수 있으며, 이는 귀납법으로 검증할 수 있습니다(힌트, 힌트). 따라서 이 알고리즘으로 F_n을 계산하는 데에는 그냥 F_n까지 세는 것보다 약 두 배의 시간이 걸립니다. 이 책의 범위를 벗어나는 방법들[4]에 따르면 $F_n = \Theta(\phi^n)$이며, 여기서 $\phi = (\sqrt{5}+1)/2 \approx 1.61803$은 이른바 *황금비*입니다. 요컨대, 이 재귀 알고리즘의 실행 시간은 n에 대해 지수적입니다.

사실 이 지수적 증가는 다음과 같이 직접 확인할 수 있습니다. RecFibo의 재귀 트리를 리프에 0과 1만 있는, 덧셈으로 이루어진 이진 트리로 생각해 봅시다. 최종 출력이 F_n이므로 리프 중 정확히 F_n개는 값이 1이어야 합니다. 이 리프들은 RecRibo(1) 호출을 나타냅니다. 간단한 귀납적 논증(힌트, 힌트)으로 RecFibo(0)이 정확히 F_{n-1}번 호출된다는 것을 알 수 있습니다. (점근적 한계만 필요하다면 RecFibo(0)의 호출 횟수가 RecFibo(1)의 호출 횟수 이하라는 점만 관찰하면 충분합니다.) 따라서 재귀 트리에는 정확히 $F_n + F_{n-1} = F_{n+1} = O(F_n)$개의 리프가 있으며, 이 트리는 정 이진 트리이므로 노드는 모두 합쳐 $2F_{n+1} - 1 = O(F_n)$개입니다.

메모(라)이제이션: 모든 것을 기억하기

재귀 알고리즘이 느린 명백한 이유는 같은 피보나치 수를 몇 번이고 거듭하여 계산하기 때문입니다. RecFibo(n)을 한 번 호출하면 RecFibo($n-1$)에 대한 재귀 호출이 한 번, RecFibo($n-2$)에 대한 재귀 호출이 두 번, RecFibo($n-3$)에 대한 재귀 호출이 세 번, RecFibo($n-4$)에 대한 재귀 호출이 다섯 번 일어나며, 일반적으로 임의의 정수 $0 \le k < n$에 대해 RecFibo($n-k$)에 대한 재귀 호출이 F_{k-1}번 일어납니다. 각 호출은 어떤 피보나치 수를 처음부터 다시 계산하고 있습니다.

재귀 호출의 결과를 기록해 두었다가 나중에 필요할 때 다시 찾아보면 재귀 알고리즘의 속도를 상당히 높일 수 있습니다. 오늘날 *메모이제이션*(memoization, 네, R이 없습니다)으로 알려진 이 최적화 기법은 보통 1967년 도널드 미치(Donald Michie)의 업적으로 알려져 있지만, 본질적으로 같은 기법을 이미 1959년에 아서 새뮤얼(Arthur Samuel)이 제안했습니다.[5]

[4] 백트래킹 점화식을 푸는 방법에 관한 노트는 http://algorithms.wtf 를 참고하십시오.

[5] 미치는 프로그래밍 언어가 배열과 함수를 따로 지원하는 대신 그가 "메모 함수(memo function)"라고 부른 추상화를 지원해야 한다고 제안했습니다. 메모 함수는 표준 함수("규칙(rule)")와 사전("암기(rote)")으로 이루어집니다. 메모 함수는 어떤 함숫값을 처음 계산할 때마다 그 값을 자신의 사전에 "기억(memorise)"합니다(네, 이번에는 R이 있습니다). 미치는 새뮤얼이 체커 게임 트리를 재귀적으로 평가하는 속도를 높이려고 "암기 학습(rote learning)"을 사용한 데서 영감을 얻었습니다. 미치는 자신의 더 일반적인 제안을 "프로그래머가 원하는 어떤 함수든 '새뮤얼화(Samuelize)' 할 수 있게 해 주는 것"이라고 설명합니다. (제가 아는 한, 미치 자신은 "memoisation"이라는 용어를 한 번도 사용하지 않았습니다.) 메모이제이션은 그보다 더 이전에 클로드 섀넌(Claude Shannon)이 1950년에 설계하고 제작한 미로 탐색 로봇 "테세우스(Theseus)"에서도 사용되었습니다.

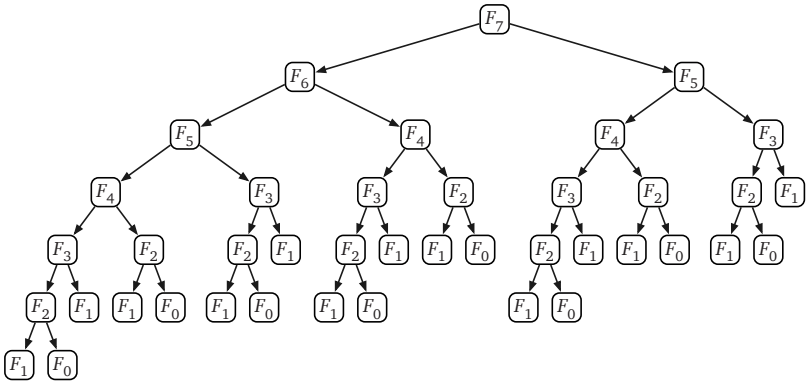


그림 3.1. F_7을 계산하는 재귀 트리입니다. 화살표는 재귀 호출을 나타냅니다.

```
MemFibo(n):
  if n = 0
      return 0
  else if n = 1
      return 1
  else
      if F[n] is undefined
          F[n] ← MemFibo(n − 1) + MemFibo(n − 2)
      return F[n]
```

메모이제이션이 알고리즘의 실행 시간을 줄여 준다는 것은 분명합니다. 그렇다면 얼마나 줄여 줄까요? MemFibo가 수행하는 재귀 호출을 실제로 따라가 보면, 배열 $F[\,]$가 아래에서부터 위로 채워진다는 것을 알 수 있습니다. 먼저 $F[2]$, 그다음 $F[3]$, 이런 식으로 $F[n]$까지 채워집니다. 이 패턴은 귀납법으로 검증할 수 있습니다. 각 항목 $F[i]$는 바로 앞 항목 $F[i-1]$이 채워진 뒤에야 채워집니다. 재귀 호출에 드는 시간을 무시하면, 각 피보나치 수 F_i에 대한 점화식을 계산하는 데에는 상수 시간만 필요합니다. 그런데 설계상 F_i에 대한 점화식은 각 인덱스 i마다 단 한 번만 계산됩니다. 따라서 MemFibo는 $O(n)$번의 덧셈만 수행한다는 결론을 얻습니다. 이는 단순한 재귀 알고리즘에 비해 *지수적인* 개선입니다!

동적 계획법: 신중하게 채우기

배열 $F[\,]$가 어떤 순서로 채워지는지 알고 나면, 더 복잡한 재귀 알고리즘이 우연히 그 순서대로 배열을 채워 주기를 기대하는 대신, 그 순서대로 배열을 *의도적으로* 채우는 단순한 for 루프로 메모이제이션된 점화식을 대체할 수 있습니다.

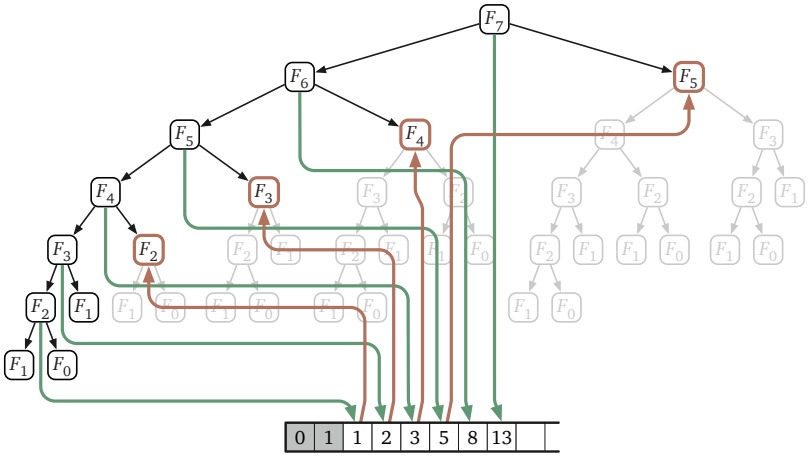


그림 3.2. 메모이제이션으로 가지치기한 F_7의 재귀 트리입니다. 아래로 향하는 초록색 화살표는 메모이제이션 배열에 값을 쓰는 것을 나타내고, 위로 향하는 빨간색 화살표는 메모이제이션 배열에서 값을 읽는 것을 나타냅니다.

```
IterFibo(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

이제 시간 분석은 바로 할 수 있습니다. IterFibo는 분명히 ***O(n)번의 덧셈***을 사용하고 $O(n)$개의 정수를 저장합니다.

이것은 우리가 처음으로 다루는 명시적인 ***동적 계획법*** 알고리즘입니다. 동적 계획법 패러다임은 1950년대 중반 RAND 연구소에서 일하던 리처드 벨먼(Richard Bellman)이 정형화하고 널리 알렸지만, 벨먼이 이 기법을 처음 사용한 사람은 결코 아닙니다. 특히 피보나치 수를 구하는 이 반복 알고리즘은 이미 Virahāṇka와 12세기의 후대 산스크리트 운율학자들이 제안했으며, 13세기 초에는 피보나치가 다시 제안했습니다![6]

이 일이 있고 여러 해가 지난 뒤, 벨먼은 "동적 계획법"이라는 이름을 일부러 골랐다고 주장했습니다. 수학 연구와 조금이라도 비슷해 보이는 것이라면 무엇이든 노골적으로 적대시하던 군 상관들에게 자기 연구의 수학적 성격을 숨기려는 의도였다는

[6] 더 일반적인 동적 계획법 기법은 1930년대 후반과 1940년대 초반에 여러 차례 독립적으로 사용되었습니다. 예를 들어 피에르 마세(Pierre Massé)는 비시 정권 시기의 프랑스에서 수력 발전 댐 운영을 최적화하는 데 동적 계획법 알고리즘을 사용했습니다. 존 폰 노이만(John von Neumann)과 오스카어 모르겐슈테른(Oskar Morgenstern)은 완전 정보를 갖춘 임의의 2인 게임(예를 들어 체커)의 승자를 결정하는 동적 계획법 알고리즘을 개발했습니다. 앨런 튜링(Alan Turing)과 그의 동료들은 블레츨리 파크에서 수행한 암호 해독 작업의 일환으로 비슷한 방법을 사용했습니다. 마세의 연구와 폰 노이만 및 모르겐슈테른의 연구는 모두 벨먼이 "동적 계획법"이라는 용어를 만들기 6년 전인 1944년에 처음 발표되었습니다. 튜링의 "반버리즘(Banburismus)"의 세부 내용은 1980년대 중반까지 기밀로 유지되었습니다.

것입니다.[7] 여기서 "programming"이라는 단어는 코드 작성을 가리키는 것이 아니라, 보통 표를 채워 나가는 방식으로 이루어지는 *계획* 또는 *일정 수립*이라는 더 오래된 의미를 가리킵니다. 예를 들어 스포츠 프로그램과 공연 프로그램은 중요한 행사들의 일정표(광고 포함)이고, 텔레비전 편성(programming)은 비어 있는 각 시간대를 방송 프로그램(과 광고)으로 채우는 일이며, 학위 프로그램은 수강해야 할 강의들의 일정표(광고 포함)입니다. 공군은 벨먼을 비롯한 연구자들에게 자금을 지원하여 훈련 및 병참 일정, 곧 그들의 표현으로는 "프로그램"을 짜는 방법을 개발하게 했습니다. "dynamic"이라는 단어는 벨먼과 동료들이 최적화하려 했던 다단계의, 시간에 따라 변하는 과정을 가리키는 말이었을 뿐 아니라, 제2차 세계 대전(WW II) 이후 미국의 미래지향적인 '하면 된다' 시대정신™과 공명할 마케팅 유행어이기도 했습니다.[8] 벨먼의 열성적인 전파에 힘입은 바도 있어, 오늘날 동적 계획법은 경제학, 로봇공학, 제어 이론을 비롯한 여러 분야에서 다단계 계획을 위한 표준 도구로 자리 잡았습니다.

결국 모든 것을 다 기억하지는 마십시오

많은 동적 계획법 알고리즘에서는 계산 전 과정에 걸쳐 중간 결과를 모두 보관할 필요가 없습니다. 예를 들어, 배열에서 가장 최근의 두 원소만 유지하면 우리 알고리즘 IterFibo의 공간 요구량을 크게 줄일 수 있습니다:

```
IterFibo2(n):
  prev ← 1
  curr ← 0
  for i ← 1 to n
      next ← curr + prev
      prev ← curr
      curr ← next
  return curr
```

(이 알고리즘은 IterFibo2(0)이 올바른 값 0을 반환하도록 비표준적이지만 일관된 기저 사례 $F_{-1} = 1$을 사용합니다.) 공간 절약은 실제로 절대적으로 중요할 수 있지만, 이 책에서는 공간 문제를 중점적으로 다루지 않겠습니다.

[7] 찰스 어윈 윌슨은 12년 동안 제너럴 모터스 사장을 지낸 뒤 1953년 1월에 국방부 장관이 되었습니다. "엔진 찰리"는 국방부를 훨씬 더 산업 기업처럼 운영하겠다는 목표를 분명히 내세웠습니다. 그리고 취임 첫해에 국방부를 개편하고 예산을 크게 줄였습니다. 벨먼은 1984년 자서전에서 윌슨을 다음과 같이 묘사했습니다.

> 워싱턴에 윌슨이라는 아주 흥미로운 신사가 있었습니다. 그는 국방부 장관이었는데, 실제로 "*연구*"라는 단어에 병적인 공포와 혐오를 품고 있었습니다. 저는 이 표현을 가볍게 쓰는 것이 아니라 정확한 의미로 쓰고 있습니다. 누군가 그의 앞에서 "*연구*"라는 말을 쓰면 그는 얼굴에 피가 몰려 벌겋게 달아올랐고, 난폭해지기까지 했습니다. 그렇다면 "*수학적*"이라는 말에 대해서는 그가 어떻게 느꼈을지 짐작하실 수 있을 것입니다. …저는 제가 RAND 연구소 안에서 실제로 수학을 하고 있다는 사실로부터 윌슨과 공군을 가려 줄 무언가를 해야 한다고 느꼈습니다. 어떤 명칭, 어떤 이름을 고를 수 있었겠습니까?

하지만 벨먼이 "동적 계획법"이라는 용어를 처음 출판물에 사용한 것은 이미 윌슨이 취임하기 몇 달 전인 1952년이었으므로, 이 이야기는 적어도 *약간은* 윤색된 것입니다.

[8] …그리고 어쩌면 찰스 로먼이 1928년에 만들어 낸, 찰스 아틀라스의 유명한 운동 시리즈를 상징하는 브랜드명 "Dynamic-Tension"을 변주한 것일 수도 있습니다. 해변의 영웅!

3.2 ♥여담: 훨씬 더 빠른 피보나치 수

앞의 알고리즘은 단순하고 매력적이지만, 피보나치 수를 계산하는 가장 빠른 알고리즘은 *아닙니다*. 피보나치 점화식을 다음과 같이 행렬로 재정식화한 것을 활용하면 더 빠른 알고리즘을 유도할 수 있습니다:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x+y \end{bmatrix}$$

다시 말해, 2차원 벡터에 행렬 $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$을 곱하는 것은 IterFibo2의 안쪽 루프를 한 번 반복하는 것과 정확히 같은 효과를 냅니다. 따라서 행렬을 n번 곱하는 것은 루프를 n번 반복하는 것과 같습니다.

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}.$$

따라서 n번째 피보나치 수를 구하려면 행렬 $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$의 n제곱만 계산하면 됩니다. 반복제곱법을 사용하면[9] 무언가의 n제곱을 계산하는 데 $O(\log n)$번의 곱셈만 필요합니다. 여기서는 "무언가"가 2×2 행렬이므로, 이는 $O(\log n)$번의 2×2 행렬 곱셈을 뜻하며, 각 행렬 곱셈은 상수 번의 정수 곱셈과 덧셈으로 환원됩니다. 따라서 F_n을 단 ***O*(log *n*) *번의 정수 산술 연산***만으로 계산할 수 있습니다.

모든 정수 m과 n에 대해 (귀납법으로!) 성립하는 항등식 $F_n = F_m F_{n-m-1} + F_{m+1} F_{n-m}$을 사용해도 같은 속도 향상을 얻을 수 있습니다. 특히 이 항등식으로부터 1898년에 에두아르 뤼카(Édouard Lucas)가 처음 제안한, 인접한 피보나치 수 쌍에 대한 다음 상호 점화식이 유도됩니다:

$$\begin{aligned} F_{2n-1} &= F_{n-1}^2 + F_n^2 \\ F_{2n} &= F_n(F_{n-1} + F_{n+1}) = F_n(2F_{n-1} + F_n) \end{aligned}$$

(이 상호 점화식은 행렬 제곱 알고리즘에서 직접 유도할 수도 있습니다.) 이 점화식들은 그대로 다음 알고리즘으로 옮길 수 있습니다:

```
《F_{n-1}, F_n 쌍을 계산합니다》
FastRecFibo(n) :
  if n = 1
      return 0, 1
  m ← ⌊n/2⌋
  hprv, hcur ← FastRecFibo(m)        《F_{m-1}, F_m》
  prev ← hprv² + hcur²                《F_{2m-1}》
  curr ← hcur · (2 · hprv + hcur)     《F_{2m}》
  next ← prev + curr                  《F_{2m+1}》
  if n is even
      return prev, curr
  else
      return curr, next
```

[9] Piṅgala가 *Chandaḥśāstra*의 다른 부분에서 2의 거듭제곱에 대해 제안한 방식입니다

표준적인 재귀 트리 기법을 적용하면 이 알고리즘은 $O(\log n)$번의 정수 산술 연산만 수행한다는 것을 알 수 있습니다.

이는 표준 반복 알고리즘보다 지수적으로 빠른 것이며, 그 표준 반복 알고리즘도 이미 우리의 원래 재귀 알고리즘보다 지수적으로 빨랐습니다. 그렇지 않습니까?

워워! 그렇게 서두르지 마십시오!

음, 꼭 그렇지는 않습니다. 피보나치 수는 지수적으로 빠르게 커집니다. n번째 피보나치 수의 길이는 십진수로 약 $n \log_{10} \phi \approx n/5$자리, 이진수로는 약 $n \log_2 \phi \approx 2n/3$비트입니다. 따라서 F_n을 로그 *시간*에 계산하는 것은 불가능합니다 — 답을 적는 데만도 $\Omega(n)$ 시간이 필요하기 때문입니다!

이 역설처럼 보이는 상황에서 벗어나는 방법은 ***임의 정밀도 산술 연산을 상수 시간에 수행할 수는 없다***는 점을 관찰하는 것입니다. 두 n자리 수를 곱하는 데 필요한 시간을 $M(n)$이라고 합시다. FastRecFibo의 실행 시간은 점화식 $T(n) = T(\lfloor n/2 \rfloor) + M(n)$을 만족하며, 재귀 트리를 이용하면 이 점화식의 해는 $T(n) = \boldsymbol{O(M(n))}$입니다. (2019년 기준으로) 알려진 가장 빠른 정수 곱셈 알고리즘은 $O(n \log n)$ 시간에 실행되므로, 이는 (2019년 기준으로) 알려진 가장 빠른 피보나치 수 계산 알고리즘의 실행 시간이기도 합니다.

이 알고리즘은 앞서 본 "선형 시간" 반복 알고리즘보다 느립니까? 사실은 그렇지 않습니다—덧셈도 공짜가 아니기 때문입니다! 두 n자리 수를 더하는 데에는 $O(n)$ 시간이 걸리므로, 반복 알고리즘 IterFibo와 IterFibo2는 실제로 $\boldsymbol{O(n^2)}$ ***시간***에 실행됩니다. (왜 그런지 아시겠습니까?) 따라서 FastRecFibo는 반복 알고리즘보다 상당히 빠르지만, *지수적*으로 빠르지는 않습니다.

원래의 재귀 알고리즘에서는 임의 정밀도 연산에 드는 추가 비용이 엄청나게 많은 재귀 호출 횟수에 묻혀 버립니다. 올바른 점화식은 $T(n) = T(n-1) + T(n-2) + O(n)$이며, 이 점화식의 해는 여전히 $T(n) = O(\phi^n)$입니다.

3.3 *Interpunctio Verborum Redux*

다음 동적 계획법 알고리즘으로, 이전 장에서 다룬 텍스트 분할 문제를 살펴봅시다. 문자열 $A[1..n]$과, 주어진 문자열이 단어인지(그것이 무엇을 의미하든) 판별하는 서브루틴 IsWord가 주어졌을 때, A를 단어들의 나열로 분할할 수 있는지 알고자 합니다.

우리는 접미사 $A[i..n]$을 단어들의 수열로 분할할 수 있을 때, 그리고 오직 그때에만 True를 반환하는 함수 $Splittable(i)$를 정의하여 이 문제를 해결했습니다. 우리가 계산해야 하는 값은 $Splittable(1)$입니다. 이 함수는 다음 점화식을 만족합니다.

$$Splittable(i) = \begin{cases} \text{True} & \text{만약} i > n \\ \displaystyle\bigvee_{j=i}^{n} \big(IsWord(i,j) \ \wedge \ Splittable(j+1) \big) & \text{그렇지 않으면} \end{cases}$$

여기서 $IsWord(i, j)$는 $IsWord(A[i..j])$의 약칭입니다. 이 점화식은 최악의 경우 IsWord 서브루틴을 $O(2^n)$번 호출하는 재귀 백트래킹 알고리즘으로 곧바로 옮겨집니다.

하지만 임의로 고정된 문자열 $A[1..n]$에 대해, 재귀 함수 $Splittable(i)$를 호출하는 서로 다른 방법은 1과 $n+1$ 사이의 각 i 값마다 하나씩, 단 n가지뿐이며, IsWord(i, j)를 호출하는 서로 다른 방법은 $1 \le i \le j \le n$을 만족하는 각 쌍 (i, j)마다 하나씩, 단 $O(n^2)$가지뿐입니다. 그런데 왜 우리는 고작 다항식 분량의 것을 계산하는 데 지수 시간을 쓰고 있는 것입니까?

각 재귀적 부분 문제는 1과 $n+1$ 사이의 정수 하나로 지정되므로, 함수 $Splittable$을 배열 $SplitTable[1..n+1]$에 메모이제이션할 수 있습니다. 각 부분 문제 $Splittable(i)$는 $j > i$인 부분 문제 $Splittable(j)$의 결과에만 의존하므로, 메모이제이션된 재귀 알고리즘은 배열을 인덱스의 *감소* 순서로 채웁니다. 이 순서대로 배열을 의도적으로 채우면 그림 3.3에 나타난 동적 계획법 알고리즘을 얻습니다. 이 알고리즘은 IsWord를 $O(n^2)$번 호출하며, 이는 앞서 살펴본 백트래킹 알고리즘에 비해 지수적으로 개선된 것입니다.

```
FastSplittable(A[1..n]):
  SplitTable[n + 1] ← True
  for i ← n down to 1
      SplitTable[i] ← False
      for j ← i to n
          if IsWord(i, j) and SplitTable[j + 1]
              SplitTable[i] ← True
  return SplitTable[1]
```

그림 3.3. Interpunctio verborum velox (빠른 텍스트 분할)

3.4 패턴: 똑똑한 재귀

한마디로 말해, 동적 계획법은 *반복 없는 재귀*입니다. 동적 계획법 알고리즘은 중간 부분 문제들의 해를 저장하는데, *항상 그런 것은 아니지만* 흔히 일종의 배열이나 표에 저장합니다. 많은 알고리즘 수강생들(그리고 강사들과 교과서들)이 올바른 점화식을 찾는 *훨씬* 더 중요한(그리고 더 어려운) 작업 대신 표에—표는 쉽고 익숙하기 때문에—집중하는 실수를 저지릅니다. 올바른 점화식을 메모이제이션하기만 한다면 명시적인 표는 사실 필요하지 않지만, 점화식이 틀렸다면 우리는 그야말로 완전히 망한 것입니다.

동적 계획법의 핵심은 표 채우기가 *아닙니다*.
핵심은 영리한 재귀입니다!

동적 계획법 알고리즘은 서로 구별되는 두 단계로 나누어 개발하는 것이 가장 좋습니다.

1. **문제를 재귀적으로 정식화하십시오.** 더 작은 부분 문제들의 답을 이용하여 전체 문제에 대한 재귀적 공식이나 알고리즘을 작성하십시오. 이 단계가 어려운 부분입니다. 완전한 재귀적 정식화는 두 부분으로 이루어집니다:

   (a) **명세.** 재귀적으로 풀고자 하는 문제를 일관되고 정확한 영어로 기술하십시오—그 문제를 *어떻게* 풀지가 아니라, 풀고자 하는 문제가 *무엇*인지를 기술해야 합니다. 이러한 명세가 없으면 여러분의 해법이 올바른지 판단하는 것은 원리적으로도 불가능합니다.

   (b) **풀이.** *정확히* 같은 문제의 더 작은 인스턴스들에 대한 답을 이용하여, 전체 문제에 대한 명확한 재귀 공식이나 알고리즘을 제시하십시오.

2. **점화식의 해를 상향식으로 구성하십시오.** 점화식의 기저 사례에서 시작하여, 중간 부분 문제들을 올바른 순서로 고려해 나가며 최종 해에 도달하는 알고리즘을 작성하십시오. 이 단계는 비교적 기계적인 몇 가지 작은 단계로 나눌 수 있습니다.

   (a) **부분 문제를 식별하십시오.** 어떤 초기 입력에서 시작했을 때, 재귀 알고리즘이 자기 자신을 호출할 수 있는 서로 다른 방식은 모두 무엇입니까? 예를 들어, RecFibo의 인자는 항상 0과 n 사이의 정수입니다.

   (b) **메모이제이션 자료구조를 선택하십시오.** 단계 (a)에서 찾아낸 *모든* 부분 문제의 해를 저장할 수 있는 자료구조를 찾으십시오. 이는 *항상 그렇지는 않지만* 대개 다차원 배열입니다.

   (c) **의존성을 파악하십시오.** 기저 사례를 제외하면 모든 부분 문제는 다른 부분 문제에 의존합니다—어떤 부분 문제에 의존합니까? 자료구조를 그림으로 그리고, 임의의 원소 하나를 고른 다음, 그 원소가 의존하는 다른 각 원소에서 그 원소로 화살표를 그리십시오. 그런 다음 그 그림을 형식화하십시오.

   (d) **적절한 평가 순서를 찾으십시오.** 각 부분 문제가 자신이 의존하는 부분 문제들보다 *뒤에* 오도록 부분 문제들의 순서를 정하십시오. 먼저 기저 사례를 고려하고, 그다음 기저 사례에만 의존하는 부분 문제들을 고려하는 식으로 진행하여, 최종적으로 원래의 최상위 문제까지 쌓아 올려야 합니다. 이전 단계에서 파악한 의존 관계는 부분 문제들 위의 부분 순서를 정의합니다. 여러분은 그 부분 순서의 선형 확장을 찾아야 합니다. ***주의하십시오!***

   (e) **공간과 실행 시간을 분석하십시오.** 서로 다른 부분 문제의 개수가 메모이제이션을 적용한 알고리즘의 공간 복잡도를 결정합니다. 전체 실행 시간을 계산하려면 *더 깊은 재귀 호출은 이미 메모이제이션되어 있다고 가정하고* 가능한 모든 부분 문제의 실행 시간을 합산하십시오. 사실 이 작업은 (a) 단계 직후에 바로 할 수 있습니다.

   (f) **알고리즘을 작성하십시오.** 부분 문제를 어떤 순서로 고려해야 하는지 알고, 각 부분 문제를 어떻게 풀어야 하는지도 알고 있습니다. 그러니 그대로 하십시오! 자료구조가 배열이라면, 이는 보통 원래의 점화식을 몇 겹의 중첩된 for 루프로 감싸고 재귀 호출을 배열 조회로 바꾸는 것을 의미합니다.

물론 이 각 단계가 올바르다는 것을 증명해야 합니다. 점화식이 틀렸거나 답을 잘못된 순서로 쌓아 올리려 한다면 알고리즘은 제대로 동작하지 않습니다!

3.5 경고: 탐욕은 어리석습니다

정말 운이 좋다면 점화식이나 표 같은 것들을 모두 건너뛰고 *탐욕* 알고리즘으로 문제를 풀 수 있습니다. 탐욕 알고리즘은 백트래킹 알고리즘처럼 일련의 결정을 통해 해를 구성합니다. 하지만 어떤 재귀적 부분 문제도 풀지 *않고* 그 결정을 직접 내립니다. 이 접근법은 매우 자연스러워 보이지만 거의 통하지 않습니다. 탐욕 알고리즘으로 올바르게 풀 수 있는 최적화 문제는 상당히 드뭅니다. 그런데도 백트래킹이나 동적 계획법으로 풀어야 하는 많은 문제에서 많은 학생은 처음에 탐욕 전략을 적용하려고 합니다.

예를 들어, 텍스트 분할 문제에 대한 탐욕 알고리즘은 입력 문자열에서 단어가 되는 가장 짧은(또는 원한다면 가장 긴) 접두사를 찾아 그 접두사를 분할의 첫 번째 단어로 받아들인 다음, 입력 문자열의 나머지 접미사를 재귀적으로 분할할 수 있습니다. 마찬가지로, 최장 증가 부분 수열 문제에 대한 탐욕 알고리즘은 입력 배열에서 가장 작은 원소를 찾아 그 원소를 대상 부분 수열의 시작으로 받아들인 다음, 그 원소의 오른쪽에서 최장 증가 부분 수열을 재귀적으로 찾을 수 있습니다. 이것들이 어리석은 꼼수처럼 들린다면 스스로를 칭찬하십시오. 이것들은 올바른 해법에 *근접하지*조차 않습니다.

누구나 로그와 빅오 표기법에 관한 온갖 규칙 바로 아래에, 다음 문장을 손등에 문신으로 새겨 두어야 합니다:

탐욕 알고리즘은 절대 통하지 않습니다!
대신 동적 계획법을 사용하십시오!

아니, 절대로 없습니까?

아니요, 절대 아닙니다!

아니, *절대*로 말입니까?

음… 거의 없습니다.[10]

탐욕적 접근법은 너무나도 유혹적이지만 올바른 경우는 매우 드물기 때문에, 저는 어떤 알고리즘 강의에서든, 심지어 평소에 정확성 증명을 요구하지 않는 강의에서도 (아니, 어쩌면 *특히* 그런 강의에서) 다음 방침을 강력히 권합니다.[11]

정확성에 대한 *형식적* 증명이 없다면, 알고리즘이 옳더라도
어떤 숙제나 시험에서든 *어떤* 탐욕 알고리즘에 대해서도
***어떠한* 점수도 받을 수 없습니다.**

[10] 그것들이 제대로 작동하는 일은 거의, 거의 없습니다! 그러니 *Pinafore*호의 엄격한 함장님을 위해 만세 삼창을 외치고, 한 번 더 외치십시오! 그러니 *Pinafore*호의 함장님을 위해 만세 삼창을 외치고, 한 번 더 외치십시오!

[11] 제 알고리즘 강의에 이 방침을 도입하자 잘못된 탐욕 알고리즘이 나오는 빈도가 크게 줄었고, 그 덕분에 학생들의 성적이 크게 올랐습니다.

게다가 학생들이 탐욕 알고리즘을 제출하고 싶은 유혹을 느끼는 문제의 대다수는 사실 동적 계획법으로 푸는 것이 가장 좋습니다. 그래서 저는 제 알고리즘 수업을 듣는 학생들에게 항상 다음과 같이 조언합니다.

"greeDY"(탐욕)라는 단어를 쓸 때마다—혹은 *생각*만 하더라도—,
여러분의 잠재의식은 DYnamic programming, 즉 동적 계획법을 사용하라고
말하고 있는 것입니다.

탐욕 알고리즘으로 올바르게 풀 수 있는 문제라 하더라도, 대개는 백트래킹이나 동적 계획법 알고리즘을 먼저 개발하는 편이 더 생산적입니다. 먼저 동작하게 만들고, 그 다음에 빠르게 만드십시오. 탐욕 알고리즘이 올바름을 증명하는 기법은 다음 장에서 살펴보겠습니다.

3.6 최장 증가 부분 수열

이전 장에서 살펴본 또 다른 문제는 주어진 수 배열 $A[1..n]$의 최장 증가 부분 수열의 길이를 계산하는 문제였습니다. 우리는 이 문제를 풀기 위해 서로 다른 두 가지 재귀 백트래킹 알고리즘을 개발했습니다. 두 알고리즘 모두 최악의 경우 $O(2^n)$ 시간에 실행되며, 두 알고리즘 모두 동적 계획법을 이용하면 속도를 크게 높일 수 있습니다.

첫 번째 점화식: 이것이 다음입니까?

첫 번째 백트래킹 알고리즘은 함수 $LISbigger(i, j)$를 계산했습니다. 이 함수는 모든 원소가 $A[i]$보다 큰 $A[j..n]$의 최장 증가 부분 수열의 길이로 정의했습니다. 이 함수에 대해 다음과 같은 점화식을 유도했습니다:

$$LISbigger(i,j) = \begin{cases} 0 & \text{만약} j > n \\ LISbigger(i, j+1) & \text{만약} A[i] \geq A[j] \\ \max\left\{ \begin{array}{c} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{array} \right\} & \text{그렇지 않으면} \end{cases}$$

원래 문제를 풀려면 배열에 보초값 $A[0] = -\infty$를 추가하고 $LISbigger(0, 1)$을 계산하면 됩니다.

각 재귀적 부분 문제는 두 인덱스 i와 j로 식별되므로, 고려해야 할 서로 다른 재귀적 부분 문제는 $O(n^2)$개뿐입니다. 이 부분 문제들의 결과는 2차원 배열 $\boldsymbol{LISbigger[0..n, 1..n]}$에 메모이제이션할 수 있습니다.[12] 게다가 재귀 호출을 제외하면 각 부분 문제는 $O(1)$ 시간에 풀 수 있으므로, 최종 동적 계획법 알고리즘은 $\boldsymbol{O(n^2)}$ *시간*에 실행될 것으로 예상할 수 있습니다.

메모이제이션을 사용하는 재귀 알고리즘이 이 배열을 어떤 순서로 채우는지는 곧바로 드러나지 않습니다. 점화식에서 알 수 있는 것은 그림 3.4의 왼쪽에 나타낸 것처럼

[12] 사실 항상 $i < j$이므로 이 배열의 절반만 있으면 됩니다. 그러나 이 책에서 상수 배를 신경 쓴다고 하더라도(신경 쓰지 않습니다), 지금은 그것을 걱정할 때가 아닙니다. 먼저 작동하게 만들고, 그다음에 더 좋게 만드십시오.

각 항목 $LISbigger[i,j]$가 다음 열의 항목 $LISbigger[i,j+1]$과 $LISbigger[j,j+1]$보다 *나중에* 채워진다는 것뿐입니다.

다행히도 이 부분적인 정보만으로도 *유효한* 평가 순서를 얻기에 충분합니다. 표를 오른쪽에서 왼쪽으로 한 번에 한 열씩 채우면, 표의 어떤 항목에 도달하든 그 항목이 의존하는 항목들은 이미 계산되어 있습니다. 이것이 재귀 알고리즘이 사용할 순서는 아닐 수도 있지만, 제대로 동작하므로 이 순서를 사용하겠습니다. 그림 3.4의 오른쪽 그림은 이 평가 순서를 보여 주며, 이중 화살표는 바깥쪽 루프를, 단일 화살표는 안쪽 루프를 나타냅니다. 이 경우 각 열을 채우는 순서는 중요하지 않으므로 단일 화살표는 양방향입니다.

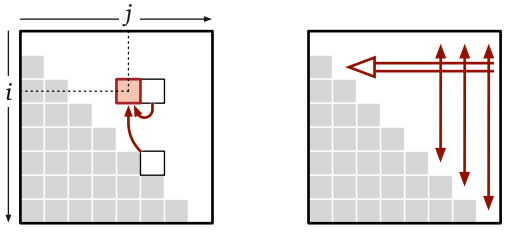


그림 3.4. 최장 증가 부분 수열의 부분 문제 의존 관계와 올바른 평가 순서

이제 끝났습니다! 동적 계획법 알고리즘의 의사코드는 아래와 같습니다. 예상대로 이 알고리즘은 분명히 $O(n^2)$ 시간에 실행됩니다. 필요하다면 표의 가장 최근 두 열인 $LISbigger[\cdot,j]$와 $LISbigger[\cdot,j+1]$만 유지하여 공간 한계를 $O(n^2)$에서 $O(n)$으로 줄일 수 있습니다.[13]

```
FastLIS(A[1..n]):
  A[0] ← −∞                                  《보초값 추가》
  for i ← 0 to n                             《기저 사례》
      LISbigger[i, n + 1] ← 0
  for j ← n down to 1
      for i ← 0 to j − 1                     《…등등》
          keep ← 1 + LISbigger[j, j + 1]
          skip ← LISbigger[i, j + 1]
          if A[i] ≥ A[j]
              LISbigger[i, j] ← skip
          else
              LISbigger[i, j] ← max{keep, skip}
  return LISbigger[0, 1]
```

두 번째 점화식: 다음은 무엇입니까?

두 번째 백트래킹 알고리즘은 함수 $LISfirst(i)$를 계산했습니다. 이 함수는 $A[i]$로 시작하는 $A[i..n]$의 최장 증가 부분 수열의 길이로 정의했습니다. 이 함수에 대해

[13] 보십시오, 아직은 상수 배에 대해 걱정하지 말라고 말씀드렸습니다!

다음 점화식을 유도했습니다:

$$LISfirst(i) = 1 + \max\{LISfirst(j) \mid j > i\text{그리고}A[j] > A[i]\}$$

여기서는 $\max\emptyset = 0$이라고 가정하므로, $LISfirst(n) = 1$과 같은 기저 사례가 점화식에서 자동으로 도출됩니다. 원래 문제를 풀려면 배열에 보초값 $A[0] = -\infty$를 추가하고 $LISfirst(0) - 1$을 계산하면 됩니다.

이 경우 재귀적 부분 문제는 하나의 인덱스 i로 나타낼 수 있으므로, 점화식을 1차원 배열 $LISfirst[1..n]$에 메모이제이션할 수 있습니다. 각 항목 $LISfirst[i]$는 $j > i$인 항목 $LISfirst[j]$에만 의존하므로, 인덱스가 감소하는 순서로 배열을 채울 수 있습니다. 각 $LISfirst[i]$를 계산하려면 $j > i$인 모든 인덱스에 대해 $LISfirst[j]$를 고려해야 하지만, 그 인덱스 j들을 특정한 순서로 고려할 필요는 없습니다. 이렇게 얻은 동적 계획법 알고리즘은 $O(n^2)$ *시간*에 실행되며 $O(n)$ 공간을 사용합니다.

```
FastLIS2(A[1..n]):
  A[0] = -∞                                  《보초값 추가》
  for i ← n downto 0
      LISfirst[i] ← 1
      for j ← i + 1 to n                     《…등등》
          if A[j] > A[i] and 1 + LISfirst[j] > LISfirst[i]
              LISfirst[i] ← 1 + LISfirst[j]
  return LISfirst[0] - 1                     《보초는 세지 마십시오》
```

3.7 편집 거리

두 문자열 사이의 ***편집 거리***는 한 문자열을 다른 문자열로 변환하는 데 필요한 문자 삽입, 문자 삭제, 문자 치환의 최소 횟수입니다. 예를 들어, FOOD와 MONEY 사이의 편집 거리는 최대 4입니다:

$$\underline{\text{F}}\text{OOD} \rightarrow \text{MO}\underline{\text{O}}\text{D} \rightarrow \text{MON}\underset{\wedge}{\underline{\text{D}}} \rightarrow \text{MONE}\underline{\text{D}} \rightarrow \text{MONEY}$$

이 거리 함수는 1965년 블라디미르 레벤슈타인(부호 이론 연구), 1968년 타라스 빈추크(음성 인식 연구), 1972년 스타니스와프 울람(생물학적 서열 연구)이 각각 독립적으로 제안했습니다. 이런 이유로 편집 거리는 때때로 ***레벤슈타인 거리*** 또는 ***울람 거리***라고 불립니다(그런데 이상하게도 "빈추크 거리"라고는 결코 불리지 않습니다).

두 문자열을 위아래로 정렬하면 이 편집 과정을 시각적으로 나타낼 수 있습니다. 이때 삽입이 있을 때마다 첫 번째 단어에 빈칸을 두고, 삭제가 있을 때마다 두 번째 단어에 빈칸을 둡니다. 서로 *다른* 두 문자가 있는 열은 치환에 해당합니다. 이렇게 나타내면 편집 단계의 수는 같은 문자가 두 번 들어 있지 *않은* 열의 수와 같습니다.

```
F  O  O     D
M  O  N  E  Y
```

FOOD를 세 단계 만에 MONEY로 변환할 수 없다는 것은 꽤 명백하므로, FOOD와 MONEY 사이의 편집 거리는 정확히 4입니다. 안타깝게도 일반적으로는 편집 수열이 가능

한 한 가장 짧은지 판단하기가 그리 쉽지 않습니다. 예를 들어, 다음 정렬은 문자열 ALGORITHM(알고리즘)과 ALTRUISTIC(이타적인) 사이의 거리가 *최대* 6임을 보여 줍니다. 이것이 우리가 할 수 있는 최선입니까?

```
A L G O R   I     T H M
A L   T R U I S T I C
```

재귀 구조

편집 거리를 계산하는 동적 계획법 알고리즘을 개발하려면 먼저 문제를 재귀적으로 정식화해야 합니다. 편집 수열을 나타내는 우리의 정렬 표현에는 "최적 부분 구조"라는 핵심적인 성질이 있습니다. 두 문자열에 대한 최단 편집 수열의 간격 표현이 주어졌다고 가정합시다. **마지막 열을 제거하면, 남은 열들은 남은 접두사들에 대한 최단 편집 수열을 나타내야 합니다.** 이 관찰은 귀류법으로 쉽게 증명할 수 있습니다. 만약 접두사들에 더 짧은 편집 수열이 있다면, 마지막 열을 다시 붙였을 때 원래 문자열들에 대한 더 짧은 편집 수열을 얻게 됩니다. 따라서 마지막 열에서 무슨 일이 일어나야 하는지만 알아내면, 최적 간격 표현의 나머지는 재귀 요정이 알아낼 수 있습니다.

달리 말하면, 우리가 찾는 정렬은 (특별한 이유 없이) 오른쪽에서 왼쪽 순서로 나열한 일련의 편집 연산을 나타냅니다. 편집 거리 문제를 풀려면 출력 정렬의 각 열마다 하나씩, 일련의 결정을 내려야 합니다. 이러한 일련의 결정을 내리는 도중에는 한 문자열의 접미사를 다른 문자열의 접미사와 이미 정렬해 둔 상태입니다.

ALGOR	I		T	H	M
ALTRU	I	S	T	I	C

정렬의 비용은 단지 일치하지 않는 열의 개수이므로, 남은 결정들은 우리가 이미 선택한 편집 연산에 의존하지 않습니다. 남은 결정들은 아직 정렬하지 않은 접두사에만 의존합니다.

ALGOR
ALTRU

따라서 임의의 두 입력 문자열 $A[1..m]$과 $B[1..n]$에 대해 편집 거리 문제를 다음과 같이 재귀적으로 정식화할 수 있습니다. 임의의 인덱스 i와 j에 대해 $\boldsymbol{Edit(i,j)}$가 접두사 $A[1..i]$와 $B[1..j]$ 사이의 편집 거리를 나타낸다고 합시다. 우리가 계산해야 하는 값은 $Edit(m,n)$입니다.

점화식

i와 j가 모두 양수일 때, $A[1..i]$와 $B[1..j]$의 최적 정렬에서 마지막 열이 될 수 있는 경우는 정확히 세 가지입니다.

- **삽입:** 윗줄의 마지막 항목이 비어 있습니다. 이 경우 편집 거리는 $Edit(i,j-1)+1$과 같습니다. $+1$은 마지막 삽입의 비용이며, 재귀 식은 나머지 정렬의 최소 비용을 나타냅니다.

ALGOR	
ALTR	U

- **삭제**: 맨 아래 행의 마지막 칸이 비어 있습니다. 이 경우 편집 거리는 $Edit(i-1,j)+1$과 같습니다. +1은 마지막 삭제에 드는 비용이고, 재귀 식은 나머지 정렬의 최소 비용을 나타냅니다.

ALGO	R
ALTRU	

- **치환**: 두 행 모두 마지막 열에 문자가 있습니다. 이 두 문자가 서로 다르면 편집 거리는 $Edit(i-1,j-1)+1$과 같습니다. 이 두 문자가 같으면 치환에 비용이 들지 않으므로 편집 거리는 $Edit(i-1,j-1)$입니다.

ALGO	R
ALTR	U

ALGO	R
ALT	R

이 일반적인 경우 분석은 $i=0$이거나 $j=0$이면 성립하지 않지만, 이러한 경계 경우는 직접 쉽게 처리할 수 있습니다.

- 빈 문자열을 길이가 j인 문자열로 변환하려면 j번의 삽입이 필요하므로 $Edit(0,j)=j$입니다.
- 길이가 i인 문자열을 빈 문자열로 변환하려면 i번의 삭제가 필요하므로 $Edit(i,0)=i$입니다.

검증 차원에서 확인해 보면, 이 두 기저 사례는 모두 빈 문자열과 빈 문자열 사이의 편집 거리가 0임을 올바르게 나타냅니다!

따라서 $Edit$ 함수는 다음 점화식을 만족한다는 결론을 얻습니다:

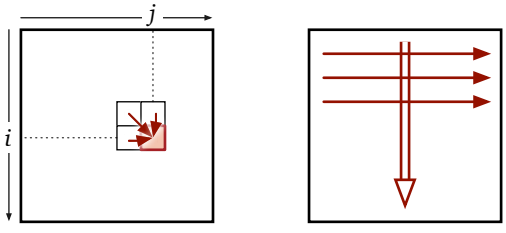
$$Edit(i,j)=\begin{cases} i & \text{만약} j=0 \\ j & \text{만약} i=0 \\ \min\begin{Bmatrix} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i]\neq B[j]] \end{Bmatrix} & \text{그렇지 않으면} \end{cases}$$

동적 계획법

이제 점화식을 얻었으니, 늘 따르던 기계적인 절차에 따라 이를 동적 계획법 알고리즘으로 바꿀 수 있습니다.

- **부분 문제**: 각 재귀적 부분 문제는 두 인덱스 $0\le i\le m$와 $0\le j\le n$으로 식별됩니다.
- **메모이제이션 구조**: 따라서 $Edit(i,j)$의 가능한 모든 값을 2차원 배열 $Edit[0..m,0..n]$에 메모이제이션할 수 있습니다.
- **의존 관계**: 각 항목 $Edit[i,j]$는 인접한 세 항목 $Edit[i-1,j]$, $Edit[i,j-1]$, $Edit[i-1,j-1]$에만 의존합니다.

- **평가 순서:** 이 배열을 표준 행 우선 순서로, 즉 위에서 아래로 한 행씩, 각 행은 왼쪽에서 오른쪽으로 채우면, 배열의 어떤 항목에 도달하든 그 항목이 의존하는 모든 항목은 이미 준비되어 있습니다. (이것이 사용할 수 있는 *유일한* 평가 순서는 아니지만, 제대로 작동하므로 이 순서를 사용합시다.)



- **공간과 시간:** 메모이제이션 구조는 $O(mn)$ 공간을 사용합니다. 각 항목 $Edit[i,j]$는 선행 항목들을 알고 나면 $O(1)$ 시간에 계산할 수 있으므로, 전체 알고리즘은 ***$O(mn)$ 시간에 실행됩니다.***

이렇게 해서 얻은 동적 계획법 알고리즘은 다음과 같습니다.

```
EditDistance(A[1..m], B[1..n]):
  for j ← 0 to n
      Edit[0, j] ← j
  for i ← 1 to m
      Edit[i, 0] ← i
      for j ← 1 to n
          ins ← Edit[i, j − 1] + 1
          del ← Edit[i − 1, j] + 1
          if A[i] = B[j]
              rep ← Edit[i − 1, j − 1]
          else
              rep ← Edit[i − 1, j − 1] + 1
          Edit[i, j] ← min{ins, del, rep}
  return Edit[m, n]
```

이 알고리즘은 흔히 1974년에 이를 기술한 Robert Wagner와 Michael Fischer의 것으로 알려져 있습니다. 그러나 Stigler의 명명 법칙에 충실하게도, 이와 동일하거나 더 일반적인 알고리즘이 1968년 Taras Vintsyuk, 1970년 V. M. Velichko와 N. G. Zagoruyko, 1972년 David Sankoff, 1974년 Peter Sellers, 그리고 거의 확실히 다른 여러 사람에 의해 각각 독립적으로 발견되었습니다.[14] 흥미롭게도, 이 저자들 중 누구도 Levenshtein이나 Ulam을 인용하지 않습니다!

[14] 이 알고리즘은 1970년의 Saul Needleman과 Christian Wunsch의 것으로 *잘못* 알려지기도 합니다. "Needleman-Wunsch 알고리즘"은 보통 두 문자열의 최장 공통 부분 수열(또는 이와 동치로, 삽입과 삭제만 허용되는 편집 거리)을 $O(mn)$ 시간에 계산하는 표준 동적 계획법 알고리즘을 가리키지만, 그 명명 또한 잘못되었습니다! 실제로 Needleman과 Wunsch의 알고리즘은 다른 점화식을 사용하여 (가중치가 있는) 최장 공통 부분 수열을 (간격 비용이 있는 경우도 포함하여) $O(m^2n^2)$ 시간에 계산합니다. Sankoff는 자신의 $O(mn)$ 시간 알고리즘이 Needleman과 Wunsch의 알고리즘을 개선한 것이라고 명시적으로 기술합니다.

입력 문자열 ALGORITHM(알고리즘)과 ALTRUISTIC에 대한 메모이제이션 표는 아래와 같습니다. 굵은 숫자는 두 문자열의 문자가 서로 같은 위치를 나타냅니다. ALGORITHM과 ALTRUISTIC 사이의 편집 거리는 실제로 6입니다!

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

이 표의 화살표는 각 항목을 실제로 정의하는 선행 항목(들)이 무엇인지 나타냅니다. 화살표의 각 방향은 서로 다른 편집 연산에 대응합니다. 가로 방향은 삭제, 세로 방향은 삽입, 대각선 방향은 치환입니다. 굵은 빨간색 대각선 화살표는 글자를 자기 자신으로 바꾸는 "무료" 치환을 나타냅니다. 이 표의 왼쪽 위 모서리에서 오른쪽 아래 모서리까지 이어지는 화살표 경로는 모두 두 문자열 사이의 최적 편집 수열을 나타냅니다. 예시 메모이제이션 배열에는 왼쪽 위 모서리에서 오른쪽 아래 모서리까지 이어지는 방향 경로가 정확히 세 개 있으며, 다음 페이지에 나와 있듯이 각 경로는 ALGORITHM을 ALTRUISTIC으로 바꾸는 서로 다른 여섯 번의 편집 수열을 나타냅니다.

```
A L G O R I   T H M
A L T R U I S T I C

A L G O R   I   T H M
A L   T R U I S T I C

A L G O R   I   T H M
A L T   R U I S T I C
```

우리의 EditDistance 알고리즘은 실제로 표에 화살표를 계산하거나 저장하지 않지만, 표의 어느 항목으로 들어오는 화살표(들)든 수치 값으로부터 $O(1)$ 시간에 즉석에서 재구성할 수 있습니다. 따라서 표를 모두 채우고 나면, $O(n+m)$의 추가 시간에 최단 편집 수열을 재구성할 수 있습니다.

3.8 부분집합 합

앞에서 살펴본 *부분집합 합* 문제는 양의 정수로 이루어진 배열 $X[1..n]$의 부분집합 중에 합이 주어진 정수 T가 되는 것이 있는지 묻는 문제였습니다. 이전 장에서 우리는 재귀적인 부분집합 합 알고리즘을 개발했으며, 이 알고리즘은 다음과 같이 다시 정리할 수 있습니다. 원래의 입력 배열 $X[1..n]$을 고정하고 다음과 같은 불리언 함수를 정의합니다.

$SS(i,t)$ = True⋯의 어떤 부분집합이 ⋯ 경우에, 그리고 오직 그 경우에만$X[i..n]$합은 다음과 같

$SS(1,T)$를 계산해야 합니다. 이 함수는 다음 점화식을 만족합니다:

$$SS(i,t) = \begin{cases} \textsc{True} & \text{만약} t = 0 \\ \textsc{False} & \text{만약} t < 0 \text{또는} i > n \\ SS(i+1,t) \ \vee \ SS(i+1,t-X[i]) & \text{그렇지 않으면} \end{cases}$$

이 점화식은 늘 쓰는 정형화된 절차에 따라 동적 계획법 알고리즘으로 바꿀 수 있습니다.

- **부분 문제**: 각 부분 문제는 $1 \le i \le n+1$을 만족하는 정수 i와 정수 $t \le T$로 기술됩니다. 그러나 $t < 0$인 부분 문제는 자명하므로, 이를 메모이제이션하는 것은 다소 어리석어 보입니다.[15] 실제로 점화식을 수정하여 그러한 부분 문제가 아예 생기지 않도록 할 수 있습니다:

$$SS(i,t) = \begin{cases} \textsc{True} & \text{만약} t = 0 \\ \textsc{False} & \text{만약} i > n \\ SS(i+1,t) & \text{만약} t < X[i] \\ SS(i+1,t) \ \vee \ SS(i+1,t-X[i]) & \text{그렇지 않으면} \end{cases}$$

- **자료구조**: 점화식을 2차원 배열 $S[1..n+1, 0..T]$에 메모이제이션할 수 있습니다. 여기서 $S[i,t]$에는 $SS(i,t)$의 값이 저장됩니다.
- **평가 순서**: 각 항목 $S[i,t]$는 최대 두 개의 다른 항목에 의존하며, 두 항목 모두 $SS[i+1,\cdot]$ 형태입니다. 따라서 바깥쪽 루프에서는 행을 아래에서 위로 살펴보고, 안쪽 루프에서는 각 행의 원소를 임의의 순서로 살펴보는 방식으로 배열을 채울 수 있습니다.

[15] 그렇습니다. 저는 지금 섣부른 최적화를 하지 말라는 제 규칙을 스스로 어기고 있습니다.

- **공간과 시간:** 메모이제이션 구조는 $O(nT)$ 공간을 사용합니다. $S[i+1,t]$와 $S[i+1,t-X[i]]$를 이미 알고 있다면 $S[i,t]$를 상수 시간에 계산할 수 있으므로, 이 알고리즘은 ***O(nT) 시간***에 실행됩니다.

이렇게 얻은 동적 계획법 알고리즘은 다음과 같습니다.

```
FastSubsetSum(X[1..n], T):
  S[n+1, 0] ← True
  for t ← 1 to T
      S[n+1, t] ← False
  for i ← n downto 1
      S[i, 0] = True
      for t ← 1 to X[i] − 1
          S[i, t] ← S[i+1, t]      ⟪t < 0인 경우를 피하십시오⟫
      for t ← X[i] to T
          S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

이 알고리즘의 최악의 경우 실행 시간 $O(nT)$는 T가 작을 때 $O(2^n)$ 시간이 걸리는 재귀 백트래킹 알고리즘보다 크게 개선된 것입니다.[16] 그러나 대상 합 T가 2^n보다 상당히 크다면, 이 반복 알고리즘은 실제로 단순한 재귀 알고리즘보다 더 느립니다. 재귀 알고리즘이 전혀 고려하지 않는 부분 문제를 푸느라 시간을 낭비하기 때문입니다. 동적 계획법이 *항상* 개선을 가져오는 것은 아닙니다![17]

3.9 최적 이진 탐색 트리

이전 장에서 마지막으로 살펴본 문제는 최적 이진 탐색 트리 문제였습니다. 입력은 탐색 키들이 정렬된 배열 $A[1..n]$과 빈도수 배열 $f[1..n]$이며, 여기서 $f[i]$는 $A[i]$를 탐색할 횟수입니다. 우리가 할 일은 모든 탐색의 총비용이 가능한 한 작아지도록 이 집합에 대한 이진 탐색 트리를 구성하는 것입니다.

빈도 배열 f를 고정하고, $OptCost(i,k)$가 부분 배열 $A[i..k]$에 대한 최적 탐색 트리의 총 탐색 시간을 나타낸다고 합시다. 우리는 함수 $OptCost$에 대해 다음과 같은 점화식을 유도했습니다.

$$OptCost(i,k) = \begin{cases} 0 & \text{만약 } i > k \\ \displaystyle\sum_{j=i}^{k} f[j] + \min_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{그렇지 않으면} \end{cases}$$

이 점화식으로 결국 무엇을 하게 될지는 아마 짐작하실 수 있겠지만, 먼저 저 보기 흉한 합부터 없애 봅시다.

[16] 부분집합 합 문제가 NP-난해이기는 하지만, 이 시간 한계가 P=NP임을 의미하는 것은 *아닙니다*. T가 반드시 입력 크기의 다항 함수로 제한되는 것은 아니기 때문입니다.

[17] 메모 함수를 제안한 1967년의 연구 메모(!)에서 Donald Michie는 "필요하지 않을 함수 값을 표로 만드는 것은 공간 낭비이고, 같은 값을 두 번 이상 다시 계산하는 것은 시간 낭비이다."라고 썼습니다. 그러나 사실 필요하지 않을 함수 값을 표로 만드는 것은 *시간* 낭비이기도 합니다!

임의의 인덱스 쌍 $i \le k$에 대해, $F(i,k)$가 구간 $A[i..k]$에 있는 모든 키의 빈도수 총합을 나타낸다고 합시다:

$$F(i,k) := \sum_{j=i}^{k} f[j]$$

이 함수는 다음과 같은 간단한 점화식을 만족합니다:

$$F(i,k) = \begin{cases} f[i] & \text{만약} i = k \\ F(i,k-1) + f[k] & \text{그렇지 않으면} \end{cases}$$

$F(i,k)$의 가능한 모든 값은 $O(n^2)$ 시간에 계산할 수 있습니다. 짐작하셨겠지만, 바로 동적 계획법을 사용하면 됩니다! 늘 하던 기계적인 단계를 따르면 다음과 같은 동적 계획법 알고리즘을 얻습니다:

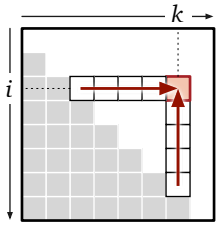
```
InitF(f[1..n]):
  for i ← 1 to n
      F[i, i-1] ← 0
      for k ← i to n
          F[i,k] ← F[i,k-1] + f[k]
```

이 짧은 알고리즘을 초기화 서브루틴으로 사용하겠습니다. 이 초기화 덕분에 원래의 *OptCost* 점화식을 다음과 같이 단순화할 수 있습니다.

$$OptCost(i,k) = \begin{cases} 0 & \text{만약} i > k \\ F[i,k] + \min\limits_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i,r-1) \\ \quad + OptCost(r+1,k) \end{array} \right\} & \text{그렇지 않으면} \end{cases}$$

이제 크랭크를 돌려 봅시다.

- **부분 문제:** 각 재귀적 부분 문제는 $1 \le i \le n+1$이고 $0 \le k \le n$인 두 정수 i와 k로 지정됩니다.
- **메모이제이션:** *OptCost*의 가능한 모든 값을 2차원 배열 $OptCost[1..n+1, 0..n]$에 저장할 수 있습니다. (실제로는 $j \ge i-1$인 항목 $OptCost[i,j]$만 쓰이지만, 상관없습니다.)
- **의존 관계:** 각 항목 $OptCost[i,k]$는 $i \le j \le k$를 만족하는 모든 j에 대해 항목 $OptCost[i,j-1]$과 $OptCost[j+1,k]$에 의존합니다. 다시 말해, 각 표 항목은 바로 왼쪽이나 바로 아래에 있는 모든 항목에 의존합니다.



다음 서브루틴은 $OptCost[i,k]$ 항목을 채웁니다. 이때 이 항목이 의존하는 모든 항목은 이미 계산되어 있다고 가정합니다.

```
ComputeOptCost(i, k):
  OptCost[i, k] ← ∞
  for r ← i to k
      tmp ← OptCost[i, r − 1] + OptCost[r + 1, k]
      if OptCost[i, k] > tmp
          OptCost[i, k] ← tmp
  OptCost[i, k] ← OptCost[i, k] + F[i, k]
```

- **평가 순서:** 배열을 채우는 순서는 적어도 세 가지가 있습니다. 대부분의 학생이 가장 먼저 떠올리는 방법은 자명한 기저 사례 $OptCost[i, i-1]$에서 시작하여 최종 답 $OptCost[1, n]$을 향해 나아가면서 표를 한 번에 대각선 하나씩 훑는 것입니다. 다음과 같습니다.

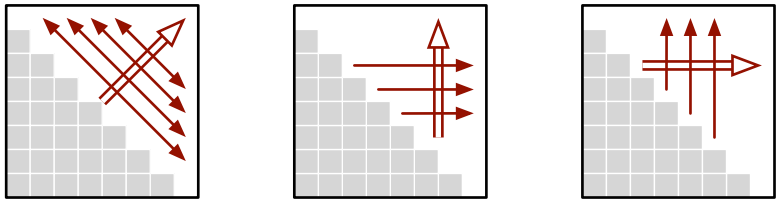
```
OptimalBST(f[1..n]):
  InitF(f[1..n])
  for i ← 1 to n + 1
      OptCost[i, i − 1] ← 0
  for d ← 0 to n − 1
      for i ← 1 to n − d        《…등등》
          ComputeOptCost(i, i + d)
  return OptCost[1, n]
```

또한 배열을 아래에서 위로 한 행씩 순회하면서 각 행을 왼쪽에서 오른쪽으로 순회할 수도 있고, 왼쪽에서 오른쪽으로 한 열씩 순회하면서 각 열을 아래에서 위로 순회할 수도 있습니다.

```
OptimalBST2(f[1..n]):
  InitF(f[1..n])
  for i ← n + 1 downto 1
      OptCost[i, i − 1] ← 0
      for j ← i to n
          ComputeOptCost(i, j)
  return OptCost[1, n]
```

```
OptimalBST3(f[1..n]):
  InitF(f[1..n])
  for j ← 0 to n + 1
      OptCost[j + 1, j] ← 0
      for i ← j downto 1
          ComputeOptCost(i, j)
  return OptCost[1, n]
```

이전과 마찬가지로, 이중선 화살표로 바깥쪽 루프를 나타내고 단일선 화살표로 안쪽 루프를 나타내어 이러한 평가 순서를 그림으로 보일 수 있습니다. 첫 번째 평가 순서에 있는 양방향 화살표는 안쪽 루프의 순서가 상관없다는 것을 나타냅니다.



- **시간과 공간:** 메모이제이션 구조는 $O(n^2)$ 공간을 사용합니다. 어떤 평가 순서를 택하든 각 항목 $OptCost[i,k]$를 계산하는 데 $O(n)$ 시간이 필요하므로, 전체 알고리즘은 $\boldsymbol{O(n^3)}$ *시간*에 실행됩니다.

늘 그렇듯이, 최종 공간 및 시간 한계는 원래의 점화식으로부터 곧바로 예측할 수도 있었습니다:

$$OptCost(i,k) = \begin{cases} 0 & \text{만약} i > k \\ F[i,k] + \min\limits_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{그렇지 않으면} \end{cases}$$

$OptCost$ 함수는 인수가 두 개이고 각 인수는 대략 n가지의 서로 다른 값을 가질 수 있으므로, 아마도 크기가 $O(n^2)$인 자료구조가 필요할 것입니다. 반면 점화식의 본문에는 *세 개*의 변수(i, k, r)가 있고 각 변수는 대략 n가지의 서로 다른 값을 가질 수 있으므로, 모든 값을 계산하는 데 $O(n^3)$ 시간이 걸릴 것입니다.

3.10 트리에서의 동적 계획법

지금까지 살펴본 동적 계획법 예제는 모두 재귀적 부분 문제의 결과를 저장하기 위해 다차원 배열을 사용했습니다. 하지만 다음 예제에서 보듯이, 배열이 항상 가장 적합한 자료구조인 것은 아닙니다.

그래프에서 ***독립 집합***이란 서로 간에 간선이 하나도 없는 정점들의 부분집합입니다. 임의의 그래프에서 가장 큰 독립 집합을 찾는 것은 극도로 어렵습니다. 사실 이 문제는 12장에서 다룰 대표적인 NP-난해 문제 중 하나입니다. 하지만 몇몇 특수한 부류의 그래프에서는 가장 큰 독립 집합을 빠르게 찾을 수 있습니다. 특히 입력 그래프가 정점이 n개인 *트리*일 때는 실제로 가장 큰 독립 집합을 $O(n)$ 시간에 계산할 수 있습니다.

트리 T가 주어졌다고 가정합시다. 일반성을 잃지 않고 T가 루트 트리라고 가정합니다. 즉, T에는 *루트*라고 부르는 특별한 노드가 있으며, 모든 간선은 암묵적으로 이 정점에서 멀어지는 방향을 가집니다. (T가 루트 없는 트리, 즉 연결된 비순환 무방향 그래프라면 임의의 정점을 루트로 선택할 수 있습니다.) w에서 루트까지의 유일한 경로가 v를 포함하면 정점 w를 정점 v의 *자손*이라고 부릅니다. 동치로, v의 자손은 v 자신과 v의 자식들의 자손입니다. v*를 루트로 하는 부분 트리*는 v의 모든 자손과 그들 사이의 간선으로 이루어집니다.

T의 임의의 노드 v에 대해, $MIS(v)$를 v를 루트로 하는 부분 트리에서 가장 큰 독립 집합의 크기라고 합시다. 이 부분 트리에서 v 자신을 제외하는 독립 집합은 모두 v의

자식들을 루트로 하는 부분 트리들에 있는 독립 집합들의 합집합입니다. 반면, v를 *포함하는* 독립 집합은 반드시 v의 자식을 모두 제외하므로, v의 손자들을 루트로 하는 부분 트리들에 있는 독립 집합들을 포함합니다. 따라서 함수 MIS는 다음 점화식을 따릅니다. 여기서 비표준 표기 $w \downarrow v$는 "w가 v의 자식임"을 의미합니다:

$$MIS(v) = \max\left\{\sum_{w\downarrow v} MIS(w),\ \ 1 + \sum_{w\downarrow v}\sum_{x\downarrow w} MIS(x)\right\}$$

r이 T의 루트일 때, $MIS(r)$을 계산해야 합니다.

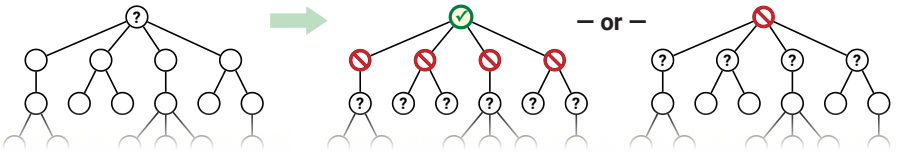


그림 3.5. 트리에서 최대 독립 집합 계산하기

이 점화식을 메모이제이션하려면 어떤 자료구조를 사용해야 하겠습니까? 가장 자연스러운 선택은 ***트리 T 그 자체입니다!*** 구체적으로 말하면, T의 각 정점 v에 대해 $MIS(v)$의 결과를 새 필드 $v.MIS$에 저장합니다. (원칙적으로는 대신 배열을 사용할 수도 있지만, 그러면 각 노드와 그에 대응하는 배열 항목 사이를 오가는 포인터가 필요하므로 굳이 그렇게 할 이유가 있겠습니까?)

부분 문제를 어떤 순서로 고려하는 것이 좋겠습니까? 임의의 노드 v에 연관된 부분 문제는 v의 자식과 손자에 연관된 부분 문제들에 의존합니다. 따라서 모든 정점을 그 부모보다 먼저 방문하기만 한다면 원하는 어떤 순서로든 노드를 방문할 수 있으며, 특히 표준적인 ***후위*** 순회를 사용할 수 있습니다.

이 알고리즘의 실행 시간은 얼마입니까? 각 노드 v에 연관된 비재귀적 시간은 v의 자식과 손자의 수에 비례하며, 이 수는 정점마다 크게 다를 수 있습니다. 하지만 분석을 거꾸로 뒤집어 볼 수 있습니다. 각 정점은 자신의 부모와 조부모에게 상수 시간만큼을 기여합니다! 각 정점에는 부모가 최대 하나, 조부모가 최대 하나뿐이므로 이 알고리즘은 $O(n)$ ***시간***에 실행됩니다.

다음은 이렇게 해서 얻은 동적 계획법 알고리즘입니다. 네, 여전히 재귀적입니다. 트리 후위 순회를 구현하는 가장 자연스러운 방법이 재귀이기 때문입니다.

```
TreeMIS(v):
  skipv ← 0
  for each child w of v
      skipv ← skipv + TreeMIS(w)
  keepv ← 1
  for each grandchild x of v
      keepv ← keepv + x.MIS
  v.MIS ← max{keepv, skipv}
  return v.MIS
```

T의 노드들에 대해 서로 다른 두 함수를 정의하면 훨씬 더 간단한 선형 시간 알고리즘을 유도할 수 있습니다:

- v를 루트로 하는 부분 트리에서 v를 *포함하는* 가장 큰 독립 집합의 *크기*를 $MISyes(v)$로 나타냅시다.
- v를 루트로 하는 부분 트리의 독립 집합 중에서 v를 *제외하는* 가장 큰 독립 집합의 크기를 $MISno(v)$로 나타냅시다.

이번에도 $\max\{MISyes(r), MISno(r)\}$를 계산해야 합니다. 여기서 r은 T의 루트입니다. 처음 두 함수는 다음과 같은 상호 점화식을 만족합니다.

$$MISyes(v) = 1 + \sum_{w \downarrow v} MISno(w)$$

$$MISno(v) = \sum_{w \downarrow v} \max\{MISyes(w), MISno(w)\}$$

이번에도 각 정점에 새로운 필드 두 개를 정의하여 이 함수들을 트리 자체에 메모이제이션할 수 있습니다. 간단한 후위 트리 순회로 모든 노드에서 두 함수를 $O(n)$ *시간*에 계산할 수 있습니다. 다음 알고리즘은 v에서 두 함수 값을 모두 메모이제이션할 뿐만 아니라, 두 값 중 더 큰 값을 반환하기도 합니다.

```
TreeMIS2(v):
  v.MISno ← 0
  v.MISyes ← 1
  for each child w of v
      v.MISno ← v.MISno + TreeMIS2(w)
      v.MISyes ← v.MISyes + w.MISno
  return max{v.MISyes, v.MISno}
```

안쪽 루프의 두 번째 줄에서는 바로 앞 줄의 재귀 호출이 메모이제이션해 둔 값 $w.MISno$를 사용합니다.

연습문제

다음의 모든 연습 문제에서—그리고 더 일반적으로는 ***어떤*** 새로운 동적 계획법 알고리즘을 개발하든—3.4절에서 설명한 단계를 따를 것을 강력히 권장합니다. 특히, 실제로 풀고 있는 재귀적 부분 문제를 영어로 명확하게 명시한 내용까지 포함하여 완전한 재귀적 해법을 갖추기 전에는 표나 for 루프에 대해 생각을 *시작*조차 하지 마십시오.[18] **먼저 동작하게 만들고, 그다음에 빠르게 만드십시오.**

수열/배열

1. 전생에 여러분은 사라진 남극 식민지 나디리아에서 계산원으로 일하며 하루의 대부분을 손님들에게 거스름돈을 내주는 데 보냈습니다. 남극에서는 종이가 매우

[18] 제 알고리즘 수업에서는 바탕이 되는 재귀적 부분 문제를 영어로 명시하지 *않은* 동적 계획법 풀이는 나머지 부분이 완벽하더라도 자동으로 *0점*을 받습니다. 이 방침을 도입한 이후 학생들의 성적이 크게 향상되었는데, 학생들이 틀린(또는 앞뒤가 맞지 않는) 동적 계획법 알고리즘을 제출하는 횟수가 크게 줄었기 때문입니다.

희귀하고 귀중한 자원이었기 때문에, 계산원은 거스름돈을 줄 때마다 가능한 한 가장 적은 수의 지폐를 사용하도록 법으로 정해져 있었습니다. 건국자 중 한 사람의 수비학적 취향 덕분에, 드림달러라고 불리는 나디리아의 화폐는 $1, $4, $7, $13, $28, $52, $91, $365의 액면가로 발행되었습니다.[19]

♠(a) 탐욕적 거스름돈 알고리즘은 대상 금액을 넘지 않는 가장 큰 지폐를 반복해서 선택합니다. 예를 들어 탐욕 알고리즘으로 $122를 만들려면 먼저 $91 지폐를 선택하고, 그다음 $28 지폐를 선택하고, 마지막으로 $1 지폐 세 장을 선택합니다. 이 탐욕 알고리즘이 가능한 최소 개수보다 더 많은 Dream-Dollar 지폐를 사용하는 예를 제시하십시오. *[힌트: 손으로 직접 따져 보는 것보다 작은 프로그램을 작성하는 편이 더 쉬울 수 있습니다.]*

(b) 정수 k가 주어졌을 때, k 드림 달러를 만드는 데 필요한 지폐의 최소 개수를 계산하는 재귀 알고리즘을 설명하고 분석하십시오. (알고리즘을 빠르게 만드는 데는 신경 쓰지 마십시오. 정확하기만 하면 됩니다.)

(c) 정수 k가 주어졌을 때, k 드림달러를 만드는 데 필요한 지폐의 최소 개수를 계산하는 동적 계획법 알고리즘을 설명하십시오. (이 알고리즘은 빨라야 합니다.)

2. 다음에 제시하는 텍스트 분할 문제의 변형들을 풀 수 있는 효율적인 알고리즘을 설명하십시오. 문자 배열을 입력으로 받아 그 문자열이 "단어"일 때, 그리고 그 때에만 True를 반환하는 서브루틴 IsWord가 있다고 가정합니다. IsWord의 호출 횟수에 상한을 구하는 방식으로 알고리즘을 분석하십시오.

(a) 문자 배열 $A[1..n]$이 주어졌을 때, A를 단어들로 분할하는 방법의 수를 계산하십시오. 예를 들어 문자열 ARTISTOIL이 주어지면, 알고리즘은 ARTIST·OIL과 ART·IS·TOIL의 두 가지 분할이 있으므로 2를 반환해야 합니다.

(b) 두 문자 배열 $A[1..n]$과 $B[1..n]$이 주어질 때, A와 B를 같은 인덱스에서 단어들로 분할할 수 있는지 판정하십시오. 예를 들어, 문자열 BOTHEARTHANDSATURNSPIN과 PINSTARTRAPSANDRAGSLAP은 다음과 같이 같은 인덱스에서 단어들로 분할할 수 있습니다:

BOT·HEART·HAND·SAT·URNS·PIN
PIN·START·RAPS·AND·RAGS·LAP

(c) 두 문자 배열 $A[1..n]$과 $B[1..n]$이 주어졌을 때, A와 B를 같은 인덱스에서 단어들로 분할할 수 있는 서로 다른 방법의 수를 계산하십시오.

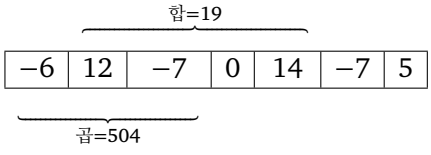
3. 양수, 음수 또는 0일 수 있으며 반드시 정수인 것은 *아닌* 수들의 배열 $A[1..n]$이 주어졌다고 가정합시다.

[19] 드림달러의 다양한 액면가 지폐 이미지를 포함하여 나디리아의 역사와 문화에 대한 더 자세한 내용은 http://moneyart.biz/dd/ 를 참고하십시오.

(a) 연속 부분 배열 $A[i..j]$에 속한 원소들의 합 중 가장 큰 값을 구하는 알고리즘을 설계하고 분석하십시오.

(b) 연속 부분 배열 $A[i..j]$에 있는 원소들의 *곱* 중 가장 큰 값을 찾는 알고리즘을 설명하고 분석하십시오.

예를 들어, 배열 $[-6, 12, -7, 0, 14, -7, 5]$가 입력으로 주어지면 첫 번째 알고리즘은 19를 반환해야 하고, 두 번째 알고리즘은 504를 반환해야 합니다.



원소가 하나인 배열 $[-374]$가 입력으로 주어지면, 첫 번째 알고리즘은 0을 반환해야 하고 두 번째 알고리즘은 1을 반환해야 합니다. (빈 구간도 엄연히 구간입니다!) 분석의 편의를 위해, 임의의 두 수를 비교하거나 더하거나 곱하는 데 $O(1)$ 시간이 걸린다고 가정하십시오.

[힌트: (a)번은 적어도 1980년대 중반부터 컴퓨터 과학 면접의 단골 질문이었습니다. 웹에서 올바른 풀이를 많이 찾을 수 있으며, 이 문제는 심지어 위키백과 문서까지 따로 있습니다! 하지만 적어도 2016년 당시에는 제가 웹에서 찾은 (b)번 풀이 가운데 상당수가 필요 이상으로 느리거나 실제로 틀린 것이었습니다.]

4. 이 연습문제에서는 최대 부분 배열 문제(문제 3)의 여러 변형을 살펴봅니다. 모든 경우에 입력은 실수(양수, 음수 또는 0일 수 있음)로 이루어진 배열 $A[1..n]$과, 경우에 따라 추가로 주어지는 정수 $X \geq 0$으로 이루어집니다.

   (a) *순환 배열:* A가 *순환* 배열이라고 가정합니다. 이 설정에서 "연속 부분 배열"은 구간 $A[i..j]$일 수도 있고, 접미사 뒤에 접두사가 이어진 $A[i..n] \cdot A[1..j]$일 수도 있습니다. A의 연속 부분 배열 중 합이 가장 큰 것을 찾는 알고리즘을 설명하고 분석하십시오.

   (b) *긴 부분 배열만:* A의 연속 부분 배열 중 *길이가 최소* X이면서 합이 가장 큰 것을 찾는 알고리즘을 설명하고 분석하십시오. ($X \leq n$이라고 가정합니다.)

   (c) *짧은 부분 배열만:* A의 연속 부분 배열 중 *길이가 최대* X인 것들 가운데 합이 가장 큰 부분 배열을 찾는 알고리즘을 설명하고 분석하십시오.

   (d) *The Price Is Right:* 합이 X *이하*인 A의 연속 부분 배열 중에서 합이 가장 큰 것을 찾는 알고리즘을 설명하고 분석하십시오.

   (e) 배열 A의 모든 수가 음이 아닐 때, 문제 4(d)를 더 빠르게 푸는 알고리즘을 설명하십시오.

5. 이 연습 문제에서는 여러 종류의 최적 *부분 수열*을 찾는 효율적인 알고리즘을 개발합니다. 부분 수열이란 어떤 수열에서 원소들의 부분집합을 뽑아내되 그 순서는 그대로 유지하여 얻은 것을 말합니다. 부분 수열의 원소들이 원래 수열에서 연속해

있을 필요는 없습니다. 예를 들어 문자열 C, DAMN, YAIOAI, DYNAMICPROGRAMMING은 모두 문자열 DYNAMICPROGRAMMING의 부분 수열입니다.

[힌트: 이 문제들 중 정확히 하나만 탐욕 알고리즘을 사용하여 $O(n)$ 시간에 풀 수 있습니다.]

(a) $A[1..m]$과 $B[1..n]$을 임의의 두 배열이라고 합시다. A와 B의 ***공통 부분 수열***이란 A와 B 모두의 부분 수열이 되는 또 다른 수열입니다. A와 B의 *최장* 공통 부분 수열의 길이를 계산하는 효율적인 알고리즘을 설명하십시오.

(b) $A[1..m]$과 $B[1..n]$을 임의의 두 배열이라고 합시다. A와 B의 ***공통 초수열***이란 A와 B를 모두 부분 수열로 포함하는 또 다른 수열입니다. A와 B의 *최단* 공통 초수열의 길이를 계산하는 효율적인 알고리즘을 설명하십시오.

(c) 수 수열 $X[1..n]$에 대해 $1 < i < n$인 인덱스 i가 존재하여 접두사 $X[1..i]$는 증가하고 접미사 $X[i..n]$은 감소할 때, 이 수열을 ***바이토닉***하다고 합시다. 임의의 정수 배열 A에서 가장 긴 바이토닉 부분 수열의 길이를 계산하는 효율적인 알고리즘을 설명하십시오.

(d) 수열 $X[1..n]$이 모든 짝수 i에 대해 $X[i] < X[i+1]$이고 모든 홀수 i에 대해 $X[i] > X[i+1]$이면, 이 수열을 ***진동하는*** 수열이라고 부릅시다. 임의의 정수 배열 A에서 가장 긴 진동하는 부분 수열의 길이를 계산하는 효율적인 알고리즘을 설명하십시오.

(e) 임의의 정수 배열 A에 대해 가장 짧은 진동 초수열의 길이를 계산하는 효율적인 알고리즘을 설명하십시오.

(f) 모든 i에 대해 $2 \cdot X[i] < X[i-1] + X[i+1]$을 만족하는 수열 $X[1..n]$을 ***볼록***하다고 부릅시다. 임의의 정수 배열 A에서 가장 긴 볼록 부분 수열의 길이를 계산하는 효율적인 알고리즘을 설명하십시오.

(g) 수열 $X[1..n]$의 각 원소가 바로 앞 두 원소의 평균보다 크면, 즉 모든 $i > 2$에 대해 $2 \cdot X[i] > X[i-1] + X[i-2]$이면 이 수열을 ***약하게 증가***한다고 합니다. 임의의 정수 배열 A에서 가장 긴 약한 증가 부분 수열의 길이를 계산하는 효율적인 알고리즘을 설명하십시오.

(h) 모든 $i > 2$에 대해 $X[i] > X[i-2]$를 만족하는 수열 $X[1..n]$을 ***이중 증가*** 수열이라고 부릅시다. (다시 말해, 이중 증가 수열은 두 증가 수열의 원소를 하나씩 번갈아 완벽하게 섞은 것입니다.) 임의의 정수 배열 A에서 가장 긴 이중 증가 부분 수열의 길이를 계산하는 효율적인 알고리즘을 설명하십시오.

(i) 수로 이루어진 수열 $X[1..n]$이 모든 i에 대해 $X[i] < X[i+1]$을 만족하면 *증가*한다고 했던 것을 떠올리십시오. 주어진 두 정수 배열의 *최장 공통 증가 부분 수열*의 길이를 계산하는 효율적인 알고리즘을 설명하십시오. 예를 들어, $\langle 1, 4, 5, 6, 7, 9\rangle$는 수열 $\langle 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3\rangle$과 $\langle 1, 4, 1, 4, 2, 1, 3, 5, 6, 2, 3, 7, 3, 0, 9, 5\rangle$의 최장 공통 증가 부분 수열입니다.

6. 두 문자열 X와 Y의 *셔플*은 X와 Y 각각의 문자 순서를 그대로 유지하면서 두 문자열의 문자들을 섞어 만든 새로운 문자열입니다. 예를 들어, 문자열 BANANAANANAS

는 여러 가지 서로 다른 방식으로 문자열 BANANA와 ANANAS의 셔플이 됩니다.

$$\text{BANANA}_{\text{ANANAS}} \qquad \text{BAN}_{\text{ANA}}\text{ANA}_{\text{NAS}} \qquad \text{B}_{\text{AN}}\text{AN}_{\text{A}}\text{A}_{\text{NA}}\text{NA}_{\text{S}}$$

마찬가지로, 문자열 PRODGYRNAMAMMIINCG와 DYPRONGARMAMMICING은 둘 다 DYNAMIC과 PROGRAMMING의 셔플입니다:

$$\text{PRO}^{\text{D}}\text{G}^{\text{Y}}\text{R}^{\text{NAM}}\text{A}^{\text{MMI}}\text{I}^{\text{NC}}\text{G} \qquad \text{DY}^{\text{PRO}}\text{N}^{\text{GA}}\text{R}^{\text{M}}\text{A}^{\text{MM}}\text{I}^{\text{C}}\text{ING}$$

(a) 세 문자열 $A[1..m]$, $B[1..n]$, $C[1..m+n]$이 주어졌을 때, C가 A와 B의 셔플인지 판별하는 알고리즘을 설명하고 분석하십시오.

(b) X와 Y의 ***매끄러운*** 셔플이란 X와 Y의 셔플 중에서 어느 문자열의 기호도 세 개 이상 연속으로 사용하지 않는 셔플을 말합니다. 예를 들면,

- $\text{PR}_{\text{D}}\text{O}_{\text{Y}}\text{G}_{\text{NA}}\text{R}_{\text{A}}\text{M}^{\text{MM}}\text{I}_{\text{C}}\text{NG}$는 문자열 DYNAMIC과 PROGRAMMING의 매끄러운 셔플입니다.
- $\text{DY}^{\text{PR}}\text{N}^{\text{OGR}}\text{A}_{\text{A}}\text{M}^{\text{MM}}\text{I}_{\text{C}}\text{ING}$은 DYNAMIC과 PROGRAMMING의 셔플이지만, 매끄러운 셔플은 아닙니다(부분 문자열 OGR과 ING 때문입니다).
- $\text{XX}_{\text{X}}\text{X}_{\text{X}}\text{X}_{\text{XX}}\text{XX}_{\text{XX}}\text{XX}_{\text{X}}\text{X}_{\text{X}}\text{XX}$은(는) 문자열 XXXXXXX와(과) XXXXXXXXXXX의 매끄러운 셔플입니다.
- 문자열 XXXX와 XXXXXXXXXXXX의 매끄러운 셔플은 존재하지 않습니다.

세 문자열 X, Y, Z가 주어졌을 때 Z가 X와 Y의 매끄러운 셔플인지 판정하는 알고리즘을 설명하고 분석하십시오.

7. 다음 각 문제에서 입력은 두 배열 $X[1..k]$와 $Y[1..n]$으로 이루어지며, 여기서 $k \le n$입니다.

(a) X가 Y의 부분 수열인지 판정하는 알고리즘을 설명하고 분석하십시오. 예를 들어, 문자열 PPAP는 문자열 PENPINEAPPLEAPPLEPEN의 부분 수열입니다.

(b) X가 더 이상 부분 수열이 되지 않도록 Y에서 제거해야 하는 기호의 최소 개수를 구하는 알고리즘을 설명하고 분석하십시오. 다시 말해, 여러분의 알고리즘은 X의 초수열이 *아닌* Y의 최장 부분 수열을 찾아야 합니다. 예를 들어, 문자열 PENPINEAPPLEAPPLEPEN에서 기호 두 개를 제거하면 문자열 PPAP는 더 이상 부분 수열이 아닙니다.

♥(c) X가 Y 안에서 *서로소인* 두 부분 수열로 나타나는지 판별하는 알고리즘을 설명하고 분석하십시오. 예를 들어, 문자열 PPAP는 문자열 PENPINEAPPLEAPPLEPEN에서 서로소인 두 부분 수열로 나타납니다.

(d) 입력에 양수, 음수 또는 0일 수 있는 수들로 이루어진 세 번째 배열 $C[1..n]$도 포함되어 있으며, $C[i]$는 $Y[i]$의 *비용*이라고 가정합시다. Y의 부분 수열로 나타나는 X의 출현 중 비용이 최소인 것을 계산하는 알고리즘을 설명하고 분석하십시오. 즉, 모든 인덱스 j에 대해 $I[j] < I[j+1]$이고 $X[I[j]] = Y[j]$이며, 총비용 $\sum_{j=1}^{k} C[j]$가 가능한 한 작은 배열 $I[1..k]$를 찾고자 합니다.

(e) X가 Y의 부분 수열로 나타나는 (겹칠 수도 있는) 모든 경우의 총 횟수를 계산하는 알고리즘을 설명하고 분석하십시오. 분석을 위해 임의의 두 정수를 $O(1)$ 시간에 더할 수 있다고 가정합니다. 예를 들어, 문자열 PPAP는 문자열 PENPINEAPPLEAPPLEPEN의 부분 수열로 정확히 23번 나타납니다. X와 Y의 모든 문자가 같다면, 여러분의 알고리즘은 $\binom{n}{k}$를 반환해야 합니다.

(f) 두 ℓ비트 정수를 더하는 데 $O(\ell)$ 시간이 걸린다면, (d)에서 여러분이 제시한 알고리즘의 실행 시간은 얼마입니까?

8. 입력 문자열 $T[1..n]$에서 정방향과 역방향으로 모두 나타나는 가장 긴 연속 부분 문자열의 길이를 구하는 효율적인 알고리즘을 설명하고 분석하십시오. 정방향 부분 문자열과 역방향 부분 문자열은 서로 겹쳐서는 안 됩니다. 다음은 몇 가지 예입니다.

- 입력 문자열 ALGORITHM이 주어지면 알고리즘은 0을 반환해야 합니다.
- 입력 문자열 RECURSION이 주어지면, 알고리즘은 부분 문자열 R에 해당하는 1을 반환해야 합니다.
- 입력 문자열 REDIVIDE가 주어지면, 알고리즘은 부분 문자열 EDI에 해당하는 3을 반환해야 합니다. (정방향 부분 문자열과 역방향 부분 문자열은 서로 겹쳐서는 안 됩니다!)
- 입력 문자열 DYNAMICPROGRAMMINGMANYTIMES가 주어지면, 여러분의 알고리즘은 부분 문자열 YNAM에 해당하는 4를 반환해야 합니다. (특히, 부분 수열 YNAMIR에 해당하는 6을 반환해서는 *안 됩니다*).

9. 회문(palindrome)은 I, DEED, RACECAR, AMANAPLANACATACANALPANAMA처럼 뒤집어도 원래와 정확히 같은 문자열입니다.

(a) 주어진 문자열의 부분 수열 중 회문이기도 한 *최장 부분 수열*의 길이를 구하는 알고리즘을 기술하고 분석하십시오.

예를 들어, 문자열 MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM의 가장 긴 회문 부분 수열은 MHYMRORMYHM입니다. 따라서 이 문자열이 입력으로 주어지면 여러분의 알고리즘은 11을 반환해야 합니다.

(b) 주어진 문자열의 초수열 중 회문이기도 한 *최단 초수열*의 길이를 구하는 알고리즘을 설명하고 분석하십시오. 예를 들어 TWENTYONE의 최단 회문 초수열은 TWENTOYOTNEWT이므로, 문자열 TWENTYONE이 입력으로 주어지면 여러분의 알고리즘은 13을 반환해야 합니다.

(c) 모든 문자열은 회문(palindrome)들의 수열로 분해할 수 있습니다. 예를 들어, 문자열 BUBBASEESABANANA(“Bubba sees a banana.”)는 다음과 같은

방법들(그리고 그 밖의 65가지 방법)로 회문들로 나눌 수 있습니다:

BUB • BASEESAB • ANANA
B • U • BB • ASEESA • B • ANANA
BUB • B • A • SEES • ABA • N • ANA
B • U • BB • A • S • EE • S • A • B • A • NAN • A
B • U • B • B • A • S • E • E • S • A • B • A • N • A • N • A

주어진 입력 문자열을 구성하는 회문의 최소 개수를 구하는 효율적인 알고리즘을 설명하고 분석하십시오. 예를 들어 입력 문자열 BUBBASEESABANANA가 주어지면 여러분의 알고리즘은 3을 반환해야 합니다.

(d) 주어진 문자열을 길이가 k 이상인 회문들로 분할할 수 있는 가장 큰 정수 k를 구하는 효율적인 알고리즘을 설명하고 분석하십시오. 예를 들면 다음과 같습니다.

- 문자열 PALINDROME(회문)이 주어지면, 여러분의 알고리즘은 1을 반환해야 합니다.
- 문자열 BUBBASEESABANANA가 주어지면, 알고리즘은 분할 BUB • BASEESAB • ANANA에 따라 3을 반환해야 합니다.
- 같은 기호 n개로 이루어진 문자열이 주어지면, 알고리즘은 n을 반환해야 합니다.

(e) 주어진 문자열을 회문(palindrome)들로 분해하는 서로 다른 방법의 수를 구하는 효율적인 알고리즘을 설명하고 분석하십시오. 예를 들면 다음과 같습니다.

- 문자열 PALINDROME(회문)이 주어지면 알고리즘은 1을 반환해야 합니다.
- 문자열 BUBBASEESABANANA가 주어지면, 여러분의 알고리즘은 70을 반환해야 합니다.
- 동일한 기호 n개로 이루어진 문자열이 주어지면, 여러분의 알고리즘은 2^{n-1}을 반환해야 합니다.

♥(f) ***메타회문***은 문자열을 회문들의 수열로 분해한 것으로, 이때 회문 길이들의 수열 자체도 회문이 되는 것을 말합니다. 예를 들면 다음과 같습니다.

BOB • S • MAM • ASEESA • UKU • L • ELE

는 문자열 BOBSMAMASEESAUKULELE의 메타회문이며, 그 길이 수열은 회문 $(3, 1, 3, 6, 3, 1, 3)$입니다. 주어진 문자열에 대해 가장 짧은 메타회문의 길이를 구하는 효율적인 알고리즘을 설명하고 분석하십시오. 예를 들어, 입력 문자열 BOBSMAMASEESAUKULELE가 주어지면 여러분의 알고리즘은 11을 반환해야 합니다.

10. 양의 정수로 이루어진 배열 $A[1..n]$이 주어졌다고 가정합시다. *증가하는 왕복 부분 수열*이란 다음 성질을 만족하는 인덱스의 수열 $I[1..\ell]$입니다.

- 모든 j에 대해 $1 \le I[j] \le n$입니다.
- 모든 $j < \ell$에 대해 $A[I[j]] < A[I[j+1]]$입니다.
- $I[j]$가 짝수이면 $I[j+1] > I[j]$입니다.
- $I[j]$가 홀수이면 $I[j+1] < I[j]$입니다.

덜 형식적으로 말하면, 각각 양의 정수가 하나씩 들어 있는 n개의 칸으로 이루어진 배열이 주어졌다고 가정합시다. 이 중 한 칸에 토큰을 놓은 다음, 토큰을 왼쪽(인덱스가 홀수인 칸에 있을 때) 또는 오른쪽(인덱스가 짝수인 칸에 있을 때)으로 반복해서 옮기되, 항상 더 작은 수에서 더 큰 수로 이동한다고 가정합시다. 그러면 토큰이 거쳐 간 위치들의 수열은 증가하는 왕복 부분 수열입니다.

n개의 정수로 이루어진 배열이 주어졌을 때, 그 배열의 최장 증가 앞뒤 왕복 부분 수열의 길이를 계산하는 알고리즘을 설명하십시오. 예를 들어, 다음과 같은 입력 배열이 주어지면

1	1	8	7	5	6	3	6	4	4	8	3	9	1	2	2	3	9	4	0
1<	2>	3<	4>	5<	6>	7<	8>	9<	10>	11<	12>	13<	14>	15<	16>	17<	18>	19<	20>

알고리즘은 정수 9를 반환해야 합니다. 이는 다음과 같은 증가하는 왕복 부분 수열의 길이입니다:

0	1	2	3	4	6	7	8	9
20>	1<	15<	18>	10>	6>	4>	3<	13<

11. 텍스트 한 문단을 종이 한 장(굳이 원하신다면 컴퓨터 화면)에 조판하려 한다고 가정합시다. 텍스트는 n개의 단어가 나열된 수열이며, i번째 단어의 길이는 $\ell[i]$입니다. 우리는 이 문단을 여러 줄로 나누되, 각 줄의 전체 길이가 정확히 L이 되도록 하려고 합니다. 예를 들어, 이 노트를 조판하는 데 사용된 프로그램인 TEX에 따르면 *지금 여러분이 읽고 있는 이 문단*의 너비는 약 11.94794 cm ≈ 4.7055인치입니다.

문단을 여러 줄로 어떻게 나누느냐에 따라 단어 사이에 넣어야 하는 공백의 양이 달라집니다. 문단은 완전히 양쪽 정렬되어야 합니다. 즉, 각 줄의 첫 글자는 왼쪽 여백에서 시작하고, *마지막 줄을 제외하면* 각 줄의 마지막 글자는 오른쪽 여백에서 끝나야 합니다. 같은 줄에 있는 두 단어 사이에는 적어도 한 단위의 공백이 있어야 합니다. *지금 읽고 계신 바로 이 문단*이 보이십니까? 바로 그런 식입니다.

문단 배치의 *슬롭(slop)*을 *마지막 줄을 제외한* 모든 줄에 대해 각 줄의 여분 공백량을 세제곱하여 합한 값으로 정의합니다. 이때 인접한 두 단어 사이마다 반드시 필요한 한 칸의 공백은 여분 공백에 포함하지 않습니다. 구체적으로, 한 줄에 단어 i부터 j까지 들어 있다면 그 줄의 슬롭은 $\left(L - j + i - \sum_{k=i}^{j} \ell[k]\right)^3$으로 정의합니다. 슬롭이 최소가 되도록 문단을 출력하는 동적 계획법 알고리즘을 설명하십시오.

12. 당신은 여덟 살 조카 엘모와 간단한 카드 게임을 하기로 합니다. 게임을 시작할 때 카드는 앞면이 보이도록 길게 한 줄로 놓입니다. 카드마다 점수가 서로 다릅니다.

카드를 모두 놓은 뒤에는 카드가 모두 없어질 때까지 당신과 엘모가 번갈아 가며 줄의 맨 왼쪽 카드나 맨 오른쪽 카드 중 하나를 가져갑니다. 매 차례마다 두 카드 중 어느 것을 가져갈지 직접 정할 수 있습니다. 게임이 끝났을 때 가장 많은 점수를 모은 플레이어가 승리합니다.

알고리즘 수업을 한 번도 들어 본 적이 없는 엘모는 뻔한 탐욕 전략을 따릅니다—자기 차례가 되면 엘모는 *항상* 점수가 더 높은 카드를 가져갑니다. 여러분의 과제는 가능할 때마다 엘모를 이길 수 있는 전략을 찾는 것입니다. (이렇게 어린아이를 상대로 이기려 드는 것이 짓궂어 보일 수도 있지만, 엘모는 어른들이 일부러 져 주는 것을 정말로 *싫어합니다.*)

(a) 여러분도 탐욕 전략을 사용해서는 안 된다는 것을 증명하십시오. 즉, 여러분이 이길 수 있는 게임이 존재하지만, 엘모와 같은 탐욕 전략을 따르지 *않을* 때에만 이길 수 있다는 것을 보이십시오.

(b) 카드의 초기 배열이 주어졌을 때, 엘모를 상대로 게임을 하여 얻을 수 있는 최대 점수를 구하는 알고리즘을 기술하고 분석하십시오.

♣(c) 엘모는 네 살 때 이보다 더 단순한 전략을 썼습니다—자기 차례가 되면 다음 카드를 항상 균등한 확률로 무작위로 골랐습니다. 즉, 자기 차례에 카드가 두 장 이상 남아 있으면 확률 1/2로 가장 왼쪽 카드를 가져가고, 확률 1/2로 가장 오른쪽 카드를 가져갔습니다. 처음 카드 순서가 주어졌을 때, 네 살 엘모를 상대로 게임을 하여 얻을 수 있는 점수의 *기댓값*의 최댓값을 구하는 알고리즘을 설명하십시오.

(d) 5년 후, 열세 살이 된 엘모는 *훨씬* 더 강한 플레이어가 되었습니다. 카드의 초기 순서가 주어졌을 때, *완벽한* 상대와 게임을 하여 여러분이 얻을 수 있는 최대 점수를 구하는 알고리즘을 설명하고 분석하십시오.

13. 여러분의 끝내주게 멋진 댄스 실력을 뽐낼 때가 거의 다 되었습니다! 내일은 여러분이 평생을 바쳐 훈련해 온 큰 댄스 대회가 열립니다. 물론 알래스카에서 삼촌과 함께 울버린을 사냥하며 보낸 그 여름만은 빼고 말입니다. 여러분은 심사위원들이 대회 중에 틀어 줄 n곡의 목록을 연주 순서대로 적은 사본을 미리 입수했습니다. 좋았어어어어어어어어어!

여러분은 모든 곡과 모든 심사위원, 그리고 자신의 춤 실력을 매우 잘 알고 있습니다. 각 정수 k에 대해, 일정표의 k번째 곡에 맞춰 춤을 추면 정확히 $Score[k]$점을 받지만, 그 후 다음 $Wait[k]$곡 동안은 체력적으로 춤을 출 수 없다는 것을 알고 있습니다(즉, $k+1$번째 곡부터 $k + Wait[k]$번째 곡까지는 춤을 출 수 없습니다). 밤이 끝났을 때 총점이 가장 높은 댄서가 대회에서 우승하므로, 여러분은 자신의 총점을 최대한 높이고자 합니다.

얻을 수 있는 최대 총점을 계산하는 효율적인 알고리즘을 설명하고 분석하십시오. 여러분의 달콤한 알고리즘에 주어지는 입력은 두 배열 $Score[1..n]$과 $Wait[1..n]$입니다.

14. 새로운 스왑 퍼즐 게임 *Candy Swap Saga XIII*에는 1부터 n까지 번호가 매겨진 귀여운 동물 n마리가 등장합니다. 각 동물은 서커스 피넛, 히스 바, 초콜라테리아 가르디니 초콜릿 트러플의 세 종류 사탕 중 하나를 들고 있습니다. 여러분의 손에도 사탕이 하나 있으며, 게임을 시작할 때 들고 있는 사탕은 서커스 피넛입니다.

    점수를 얻기 위해 여러분은 1번부터 n번까지 각 동물을 차례대로 방문합니다. 동물마다 손에 든 사탕을 그대로 가지고 있거나, 그 동물이 들고 있는 사탕과 바꿀 수 있습니다.

    - 자신의 사탕을 *같은* 종류의 다른 사탕과 바꾸면 1점을 얻습니다.
    - 가지고 있는 사탕을 *다른* 종류의 사탕과 바꾸면 1점을 잃습니다. (그렇습니다, 점수가 음수가 될 수도 있습니다.)
    - 동물을 방문했을 때 사탕을 교환하지 않기로 하면 점수는 변하지 않습니다.

    여러분은 *반드시* 동물들을 순서대로 방문해야 하며, 한 번 방문한 동물은 다시는 방문할 수 없습니다.

    가능한 최대 점수를 계산하는 효율적인 알고리즘을 설명하고 분석하십시오. 입력은 배열 $C[1..n]$이며, 여기서 $C[i]$는 i번째 동물이 들고 있는 사탕의 종류입니다.

15. 새로 설립된 Maksymilian R. Levchin 컴퓨터과학대학의 초대 학장인 Lenny Rutenbar는 Orchard Downs 썰매 언덕의 남쪽 비탈[20]에 눈 점프대를 여러 개 만들게 하고 전기컴퓨터공학과 학과장인 Bill Kudeki에게 썰매 시합을 신청했습니다. Bill과 Lenny는 둘 다 썰매를 타고 언덕을 내려가며, 각자 체공 시간을 최대로 늘리려고 합니다. 승자는 자신의 학과/단과대학을 Siebel Center와 새 ECE 건물 양쪽으로 확장할 수 있고, 패자는 자신의 학과/단과대학 전체를 Loomis Lab 옆의 Boneyard 배수로로 옮겨야 합니다.

    레니나 빌이 *지면 위에 있는 상태에서* 점프대에 도달할 때마다, 그 점프대를 이용해 공중으로 뛰어올라 하나 이상의 점프대를 날아 넘을 수도 있고, 썰매를 탄 채 그 점프대를 지나쳐 지면에 머무를 수도 있습니다. 물론 어떤 점프대를 날아 넘었다면, 그 점프대를 이용해 점프를 연장할 수는 없습니다.

    (a) 두 배열 $Ramp[1..n]$과 $Length[1..n]$이 주어졌다고 합시다. 여기서 $Ramp[i]$는 언덕 꼭대기에서 i번째 점프대까지의 거리이고, $Length[i]$는 i번째 점프대를 타는 썰매꾼이 누구든 공중으로 날아가는 거리입니다. 레니나 빌이 공중에서 보낼 수 있는 최대 총거리를 구하는 알고리즘을 설명하고 분석하십시오.

    (b) 대학의 변호사들은 레니와 빌의 작은 내기 소식을 듣고 즉시 이의를 제기했습니다. 이들은 소송이나 치솟는 보험료로부터 대학을 보호하기 위해 두 썰매꾼 중 누구든 할 수 있는 점프 횟수에 상한을 둡니다. 원래의 배열 $Ramp[1..n]$과 $Length[1..n]$, 그리고 정수 k가 입력으로 주어질 때, 레니나 빌이 *최대*

[20] 북쪽 비탈이 더 빠르지만, 흥미로운 시합을 하기에는 너무 짧습니다.

*k번의 점프*로 공중에서 보낼 수 있는 최대 총 거리를 구하는 알고리즘을 설명하고 분석하십시오.

♥(c) 변호사들은 자신들이 부과한 제한만으로는 대회를 곧바로 중단시킬 수 없다는 사실을 깨닫자, 새로운 제한을 추가했습니다. 어떤 경사로도 두 번 이상 사용할 수 없다는 것입니다! 법적 간섭에 넌더리가 난 레니와 빌은 내기를 포기하고, 관중에게 멋진 쇼를 보여 주기 위해 서로 협력하기로 합니다. 레니와 빌이 각자 최대 k번 점프하고(따라서 합계 최대 $2k$번) 각 경사로를 최대 한 번만 사용할 때, 두 사람이 공중에서 보낼 수 있는 최대 총거리를 구하는 알고리즘을 설명하고 분석하십시오.

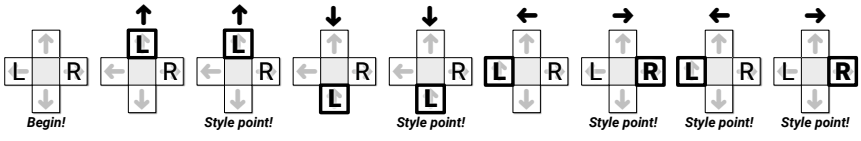
16. 농부 보기스, 번스, 빈이 미스터 폭스를 위해 장애물 코스를 만들었습니다. 이 코스는 길게 한 줄로 늘어선 부스들로 이루어져 있으며, 각 부스의 앞면에는 선명한 빨간 페인트로 숫자가 칠해져 있습니다. 형식적으로 말하면, 미스터 폭스에게 배열 $A[1..n]$이 주어지며, 여기서 $A[i]$는 i번째 부스의 앞면에 칠해진 숫자입니다. 각 숫자 $A[i]$는 양수일 수도, 음수일 수도, 0일 수도 있습니다. 모두가 다음 규칙에 동의합니다:

    - 각 부스에서 폭스 씨는 "Ring!" 또는 "Ding!" 중 하나를 ***반드시*** 말해야 합니다.
    - 폭스 씨가 i번째 부스에서 "링!"이라고 외치면 닭 $A[i]$마리를 보상으로 받습니다. ($A[i] < 0$이면 폭스 씨는 닭 $-A[i]$마리를 벌금으로 냅니다.)
    - 폭스 씨가 i번째 부스에서 "딩!"이라고 말하면 닭 $A[i]$마리를 벌금으로 냅니다. ($A[i] < 0$이면 폭스 씨는 닭 $-A[i]$마리를 보상으로 받습니다.)
    - 폭스 씨는 같은 단어를 네 번 이상 연속으로 말할 수 없습니다. 예를 들어 폭스 씨가 6번, 7번, 8번 부스에서 "Ring!"이라고 말했다면, 9번 부스에서는 반드시 "Ding!"이라고 말해야 합니다.
    - 모든 셈은 폭스 씨가 모든 부스를 방문하고 심판이 "핫 박스!"라고 외친 뒤, 마지막에 한꺼번에 정산됩니다. 폭스 씨가 실제로 닭을 들고 장애물 코스를 통과해야 하는 것은 아닙니다.
    - 마지막으로, 여우 씨가 규칙을 하나라도 어기거나 장애물 코스를 마쳤을 때 농부들에게 갚아야 할 닭이 남아 있다면, 농부들은 여우 씨를 총으로 쏠 것입니다.

    숫자 배열 $A[1..n]$이 입력으로 주어질 때, 폭스 씨가 장애물 코스를 달려서 얻을 수 있는 닭의 최대 마릿수를 계산하는 알고리즘을 설명하고 분석하십시오. *[힌트: 불타는 솔방울을 조심하십시오!]*

17. ***Dance Dance Revolution***은 1998년 코나미가 일본에서 처음 선보인 댄스 비디오 게임입니다. 플레이어는 앞, 뒤, 왼쪽, 오른쪽을 가리키는 네 개의 화살표가 십자 모양으로 배치된 발판 위에 섭니다. 게임이 진행되는 동안 게임은 노래를 틀면서 n개의 화살표(←, ↑, ↓, → 중 하나)로 이루어진 수열을 화면 아래쪽에서 위쪽으로 스크롤합니다. 각 화살표가 화면 맨 위에 도달하는 바로 그 순간, 플레이어는 댄스 발판에서 그에 해당하는 화살표를 밟아야 합니다. (화살표는 노래의 박자에 맞춰

밟게 되도록 타이밍이 맞춰져 있습니다.) 여러분은 이 게임의 변형인 "Vogue Vogue Revolution"을 하고 있으며, 이 게임의 목표는 완벽하게 플레이하되 가능한 한 적게 움직이는 것입니다. 화살표가 화면 맨 위에 도달했을 때 한쪽 발이 이미 올바른 화살표 위에 있다면, 현재 자세를 유지한 대가로 스타일 점수 1점을 받습니다. 어느 발도 올바른 화살표 위에 있지 않다면, 한 발(그리고 *오직* 한 발)을 현재 위치에서 발판 위의 올바른 화살표로 옮겨야 합니다. 잘못된 화살표를 밟거나, 올바른 화살표를 밟지 못하거나, 한 번에 두 발 이상을 움직이거나, 이미 올바른 화살표 위에 서 있는데 어느 한 발이라도 움직이면, 스타일 점수를 모두 잃고 게임에서 집니다. 스타일 점수의 총합을 최대화하려면 발을 어떻게 움직여야 합니까? 이 문제에서는 항상 왼발을 ← 위에, 오른발을 → 위에 두고 시작하며, 화살표 수열 전체를 외우고 있다고 가정합니다. 예를 들어 수열이 ↑↑↓↓←→←→이면, 아래와 같이 발을 움직여 스타일 점수 5점을 얻을 수 있습니다:



(a) n개의 화살표로 이루어진 *임의의* 수열에 대해 스타일 점수를 적어도 $n/4-1$점 얻을 수 있음을 ***증명하십시오***.

(b) 주어진 VVR 루틴에서 얻을 수 있는 스타일 점수의 최댓값을 구하는 효율적인 알고리즘을 설명하십시오. 알고리즘의 입력은 화살표의 순서를 담은 배열 $Arrow[1..n]$입니다.

18. 다음과 같은 1인용 스크래블을 생각해 봅시다. 먼저 고정된 유한 길이의 타일 수열이 주어지며, 각 타일에는 글자와 숫자 값이 하나씩 있습니다. 게임을 시작할 때 수열에서 처음 일곱 개의 타일을 뽑아 손에 듭니다. 매 턴마다 손에 든 타일 중 일부 또는 전부로 영어 단어를 하나 만들어 그 타일들을 테이블에 내려놓고, 그 타일들의 값을 모두 더한 만큼 점수를 얻습니다. (손에 든 타일로 영어 단어를 하나도 만들 수 없으면 게임은 즉시 끝납니다.) 그런 다음 (a) 손에 든 타일이 일곱 개가 되거나 (b) 수열이 빌 때까지 수열의 맨 앞에서 다음 타일을 반복해서 뽑습니다. (아쉽지만 단어/글자 2배·3배 점수, 빙고, 빈 타일, 패스는 없습니다.) 목표는 가능한 한 많은 점수를 얻는 것입니다. 예를 들어, 다음과 같은 20개의 타일로 이루어진 수열을 생각해 봅시다.

I_2	N_2	X_8	A_1	N_2	A_1	D_3	U_5	D_3	I_2	D_3	K_8	U_5	B_4	L_2	A_1	K_8	H_5	A_1	N_2

게임을 시작할 때 타일이 이러한 순서로 주어지면, 다음과 같이 68점을 얻을 수 있습니다.

- 처음에 $\boxed{I_2}\boxed{N_2}\boxed{X_8}\boxed{A_1}\boxed{N_2}\boxed{A_1}\boxed{D_3}$를 뽑습니다.
- $\boxed{N_2}\boxed{A_1}\boxed{I_2}\boxed{A_1}\boxed{D_3}$라는 단어를 놓아 9점을 얻고, 손에는 $\boxed{N_2}\boxed{X_8}$를 남깁니다.

- 다음 타일 다섯 개 $\boxed{U_5}\boxed{D_3}\boxed{I_2}\boxed{D_3}\boxed{K_8}$를 뽑습니다.
- $\boxed{U_5}\boxed{N_2}\boxed{D_3}\boxed{I_2}\boxed{D_3}$ 단어를 내려놓아 15점을 얻고, 손에는 $\boxed{K_8}\boxed{X_8}$를 남깁니다.
- 다음 타일 다섯 개 $\boxed{U_5}\boxed{B_4}\boxed{L_2}\boxed{A_1}\boxed{K_8}$를 뽑습니다.
- 단어 $\boxed{B_4}\boxed{U_5}\boxed{L_2}\boxed{K_8}$를 놓아 19점을 얻고, 손에는 $\boxed{K_8}\boxed{X_8}\boxed{A_1}$를 남깁니다.
- 마지막 세 타일 $\boxed{H_5}\boxed{A_1}\boxed{N_2}$를 뽑습니다.
- 단어 $\boxed{A_1}\boxed{N_2}\boxed{K_8}\boxed{H_5}$를 놓아 16점을 얻고, 손에는 $\boxed{X_8}\boxed{A_1}$를 남깁니다.
- 단어 $\boxed{A_1}\boxed{X_8}$를 놓아 9점을 얻고, 손에 든 타일을 모두 비워 게임을 끝냅니다.

(a) 타일 수열이 두 배열, 즉 A에서 Z 사이의 문자로 이루어진 수열을 담은 $Letter[1..n]$과 $Value[\texttt{A}..\texttt{Z}]$로 표현된다고 가정합시다. 여기서 $Value[\ell]$은 문자 ℓ이 적힌 임의의 타일의 값입니다. 주어진 타일 수열로부터 얻을 수 있는 최대 점수를 계산하는 효율적인 알고리즘을 설계하고 분석하십시오.

(b) 이제 같은 글자가 적힌 두 타일이 서로 다른 값을 가질 수도 있다고 가정합시다. 이 경우 타일 수열은 두 배열 $Letter[1..n]$과 $Value[1..n]$으로 표현되며, 여기서 $Value[i]$는 i번째 타일의 값입니다. 주어진 타일 수열로 얻을 수 있는 최대 점수를 계산하는 효율적인 알고리즘을 설계하고 분석하십시오. 두 문제 모두에서 출력은 가능한 최대 점수라는 하나의 수입니다. 최대 일곱 개의 타일로 이루어진 임의의 집합으로 만들 수 있는 모든 영어 단어와 그 단어들의 점수를 $O(1)$ 시간에 찾을 수 있다고 가정하십시오(실제로 그렇기 때문입니다).

19. 평면 위에 n개의 선분으로 이루어진 집합 L이 주어졌다고 가정합시다. 각 선분의 한 끝점은 직선 $y = 0$ 위에, 다른 끝점은 직선 $y = 1$ 위에 있으며, $2n$개의 끝점은 모두 서로 다릅니다.

(a) 어떤 두 선분도 서로 교차하지 않는 L의 가장 큰 부분 집합을 계산하는 알고리즘을 설명하고 분석하십시오.

(b) **모든** 선분 쌍이 교차하는 L의 가장 큰 부분집합을 계산하는 알고리즘을 기술하고 분석하십시오.

이제 평면 위의 선분 n개로 이루어진 집합 L이 주어졌다고 가정합시다. 여기서 각 선분의 양 끝점은 모두 단위원 $x^2 + y^2 = 1$ 위에 있고, $2n$개의 끝점은 모두 서로 다릅니다.

(c) 어떤 두 선분도 서로 교차하지 않는 L의 부분 집합 중 가장 큰 것을 계산하는 알고리즘을 기술하고 분석하십시오.

(d) **모든** 선분 쌍이 서로 교차하는 L의 부분집합 중 가장 큰 것을 계산하는 알고리즘을 설명하고 분석하십시오.

20. P를 단위원 위에 균등하게 분포된 n개의 점으로 이루어진 집합이라 하고, S를 P에 속한 점을 끝점으로 하는 m개의 선분으로 이루어진 집합이라 합시다. m개 선분의 끝점이 반드시 서로 다를 필요는 *없습니다*. n은 $2m$보다 훨씬 작을 수도 있습니다.

    (a) S에 속한 선분들의 부분집합 가운데 모든 쌍이 서로소인 가장 큰 부분집합의 크기를 구하는 알고리즘을 설명하십시오. 두 선분이 끝점에서조차 교차하지 않으면 두 선분은 서로소라고 합니다.

    (b) S에 속한 선분들의 부분집합 중 모든 쌍이 내부 서로소(interior-disjoint)인 가장 큰 부분집합의 크기를 구하는 알고리즘을 설명하십시오. 두 선분의 교집합이 공집합이거나 두 선분 모두의 끝점이면, 두 선분은 내부 서로소입니다.

    (c) S에 속한 선분들의 부분 집합 중에서 모든 쌍이 서로 교차하는 가장 큰 부분집합의 크기를 구하는 알고리즘을 설명하십시오.

    (d) 모든 쌍이 서로 교차하는 S의 선분 부분집합 중 가장 큰 것의 크기를 구하는 알고리즘을 설명하십시오. 두 선분이 만나되 끝점에서 만나는 것이 아니면 두 선분이 교차한다고 합니다.

    만점을 받으려면 네 가지 알고리즘 모두 $O(mn)$ 시간에 실행되어야 합니다.

21. 여러분은 시끄럽고 들뜨고 목마른 학생들과 탄산음료 디스펜서로 가득 찬 버스를 몰고 고속도로를 달리고 있습니다. 학생은 버스에 타고 있는 동안 1분마다 탄산음료를 1온스씩 마십니다. 여러분의 목표는 학생들을 빨리 내려 주어 모든 학생이 마시는 탄산음료의 총량을 가능한 한 작게 만드는 것입니다. 여러분은 각 출구에서 몇 명의 학생이 버스에서 내릴지 알고 있습니다. 버스는 고속도로 위의 어딘가(아마도 양 끝은 아닌 곳)에서 출발하며 시속 37.4마일의 상수 속도로 이동합니다. 버스는 고속도로를 따라서만 운전해야 합니다. 하지만 한 출구까지 앞으로 갔다가 반대 방향에 있는 출구까지 뒤로 갈 수 있으며, 방향은 원하는 만큼 자주 바꿀 수 있습니다. (버스를 세우고, 학생들을 내려 주고, 방향을 돌리는 데에는 시간이 전혀 걸리지 않습니다.) 학생들이 탄산음료를 가능한 한 적게 마시도록 학생들을 내려 주는 효율적인 알고리즘을 설명하십시오. 입력은 버스 노선(출구 목록과 연속한 출구 사이의 이동 시간), 각 출구에서 내려 줄 학생 수, 그리고 버스의 현재 위치(출구라고 가정해도 됩니다)로 이루어집니다.

22. 두 문자열 A와 B의 *요약*을 다음과 같은 형태의 부분 문자열들을 이어 붙인 것으로 정의합시다:

    - ▲SNA는 첫 번째 문자열 A에만 있는 부분 문자열 SNA를 나타냅니다.
    - ◆FOO는 두 문자열의 공통 부분 문자열 FOO를 나타냅니다.
    - ▼BAR는 두 번째 문자열 B에만 있는 부분 문자열 BAR를 나타냅니다.

    요약의 적절한 부분 문자열들을 순서대로 이어 붙이고 구분자 ▲, ◆, ▼를 버려서 원래 문자열 A와 B를 복원할 수 있으면, 그 요약은 *유효*합니다. 각 일반 문자의

길이는 1이고, 각 구분자 ▲, ◆, ▼는 음이 아닌 어떤 고정된 길이 Δ를 가집니다. 요약의 *길이*는 그 요약을 이루는 기호들의 길이의 합입니다. 예를 들어, 다음 문자열들은 각각 문자열 KITTEN과 KNITTING의 유효한 요약입니다:

- ◆K▼N◆ITT▲E▼I◆N▼G의 길이는 9 + 7Δ입니다.
- ◆K▼N◆ITT▲EN▼ING의 길이는 10 + 5Δ입니다.
- ◆K▲ITTEN▼NITTING의 길이는 13 + 3Δ입니다.
- ▲KITTEN▼KNITTING의 길이는 14 + 2Δ입니다.

주어진 두 문자열 $A[1..m]$과 $B[1..n]$의 가장 짧은 요약의 길이를 계산하는 알고리즘을 설명하고 분석하십시오. 구분자 길이 Δ도 알고리즘의 입력에 포함됩니다. 예를 들면 다음과 같습니다.

- 문자열 KITTEN과 KNITTING, 그리고 $\Delta = 0$이 주어지면 여러분의 알고리즘은 9를 반환해야 합니다.
- 문자열 KITTEN과 KNITTING, 그리고 $\Delta = 1$이 주어지면 여러분의 알고리즘은 15를 반환해야 합니다.
- 문자열 KITTEN과 KNITTING, 그리고 $\Delta = 2$가 주어지면 여러분의 알고리즘은 18을 반환해야 합니다.

23. *Vankin's Mile*은 $n \times n$ 정사각형 격자 위에서 하는 미국의 1인용 게임입니다. 플레이어는 먼저 격자의 아무 칸에나 말을 하나 놓습니다. 그다음 매 차례마다 말을 오른쪽으로 한 칸 또는 아래로 한 칸 옮깁니다. 플레이어가 말을 판의 경계 간선 밖으로 옮기면 게임이 끝납니다. 격자의 각 칸에는 수치가 하나씩 있으며, 이 값은 양수, 음수, 0 중 어느 것이든 될 수 있습니다. 플레이어는 0점에서 시작하며, 말이 어떤 칸에 놓일 때마다 그 칸의 값을 자신의 점수에 더합니다. 게임의 목표는 가능한 한 많은 점수를 얻는 것입니다. 예를 들어 아래 격자에서는 처음 말을 둘째 행의 8에 놓고 아래, 아래, 오른쪽, 아래, 아래 순으로 옮기면 $8-6+7-3+4 = 10$점을 얻을 수 있습니다. (이 점수는 이 숫자 격자에서 얻을 수 있는 최고 점수가 *아닙니다*.)

−1	7	−8	10	−5
−4	−9	**8** ⇓	−6	0
5	−2	**−6** ⇓	−6	7
−7	4	**7** ⇒	**−3** ⇓	−3
7	1	−6	**4** ⇓	−9

(a) $n \times n$ 크기의 값 배열이 입력으로 주어질 때, Vankin's Mile 게임에서 얻을 수 있는 최대 점수를 계산하는 효율적인 알고리즘을 기술하고 분석하십시오.

(b) 이 게임의 유럽판은 적절하게도 *반킨의 킬로미터*라고 불리며, 플레이어는 매 차례 토큰을 아래로 한 칸, 오른쪽으로 한 칸, *또는 왼쪽으로 한 칸* 움직일

수 있습니다. 다만 점수가 무한히 커지는 것을 막기 위해, 토큰은 같은 칸에 두 번 이상 놓일 수 없습니다. $n \times n$ 크기의 값 배열이 입력으로 주어질 때, 반킨의 킬로미터 게임에서 얻을 수 있는 최대 점수를 계산하는 효율적인 알고리즘을 설명하고 분석하십시오.[21]

24. 0과 1로 이루어진 배열 $M[1..n, 1..n]$ 형태로 $m \times n$ 비트맵이 주어졌다고 합시다. M의 *단색* 블록은 모든 비트가 같은 $M[i..i', j..j']$ 형태의 부분 배열입니다. 단색 블록은 행의 수와 열의 수가 같으면 정사각형이라고 합니다.

    (a) M에서 속이 꽉 찬 *정사각형* 블록의 최대 넓이를 $O(n^2)$ 시간에 구하는 알고리즘을 설명하십시오.

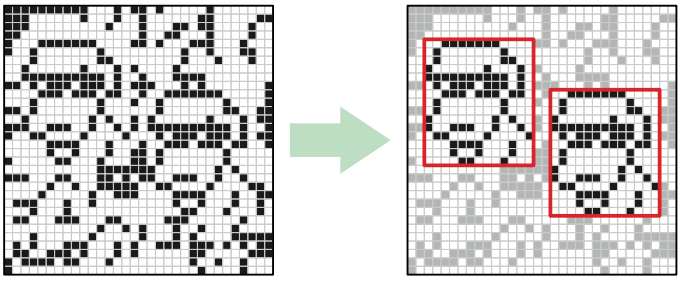
    (b) M에서 속이 꽉 찬 블록의 최대 넓이를 $O(n^3)$ 시간에 구하는 알고리즘을 설명하십시오.

    (c) M에서 속이 찬 블록의 최대 넓이를 $O(n^2 \log n)$ 시간에 찾는 알고리즘을 설명하십시오. *[힌트: 분할 정복을 사용하십시오.]*

    ♥(d) M에서 꽉 찬 블록의 최대 넓이를 $O(n^2)$ 시간에 구하는 알고리즘을 설명하십시오.

25. 양수, 음수 또는 0일 수 있고 반드시 정수일 *필요는 없는* 수들로 이루어진 배열 $M[1..n, 1..n]$이 주어졌다고 가정합시다. $M[i..i', j..j']$ 형태의 직사각형 부분 배열 가운데 원소의 합이 가장 큰 값을 찾는 알고리즘을 설명하십시오. 만점을 받으려면 알고리즘이 $O(n^3)$ 시간에 실행되어야 합니다. *[힌트: 문제 3을 참고하십시오.]*

26. 주어진 비트맵에서 두 번 이상 나타나는 직사각형 패턴 중 넓이가 최대인 것을 찾는 알고리즘을 설계하고 분석하십시오. 구체적으로, 비트로 이루어진 2차원 배열 $M[1..n, 1..n]$이 입력으로 주어질 때, 알고리즘은 M에서 반복되는 직사각형 패턴 중 가장 큰 것의 넓이를 출력해야 합니다. 예를 들어 아래 그림의 왼쪽에 있는 비트맵이 주어지면, 알고리즘은 15×13 크기의 멍멍이의 넓이인 정수 195를 반환해야 합니다. (이 예에서는 그렇지 않지만, 반복되는 패턴의 두 사본은 서로 겹칠 수도 있습니다.)



[21] *위쪽* 이동까지 허용한다면, 그 결과로 얻어지는 게임(반킨의 패덤?)은 NP-난해가 됩니다.

(a) 만점을 받으려면 $O(n^5)$ 시간에 실행되는 알고리즘을 설명하십시오.

♥(b) 추가 점수를 위해, $O(n^4)$ 시간에 실행되는 알고리즘을 설명하십시오.

♣♥(c) 추가 점수를 더 받고 싶다면 $O(n^3 \operatorname{polylog} n)$ 시간에 실행되는 알고리즘을 설명하십시오.

27. P를 평면 위에서 *볼록 위치*에 있는 점들의 집합이라고 합시다. 직관적으로 말하면, 이 점들을 고무줄로 두르면 모든 점이 고무줄에 닿습니다. 더 엄밀하게 말하면, P의 임의의 점 p에 대해 p를 P의 다른 점들과 분리하는 직선이 존재합니다. 또한 가장 왼쪽 점 $P[1]$부터 시작하여 "고무줄"을 따라 반시계 방향 순서로 점들에 $P[1], P[2], \ldots, P[n]$과 같이 번호가 매겨져 있다고 가정합시다. 이 문제에서는 외판원 문제의 한 특수한 경우를 풀어야 합니다. 여기서 외판원은 P의 모든 점을 방문해야 하며, 한 점 $p \in P$에서 다른 점 $q \in P$로 이동하는 비용은 유클리드 거리 $|pq|$입니다.

(a) P의 최단 *순환* 투어를 계산하는 간단한 알고리즘을 설명하십시오.

(b) *단순* 투어란 자기 자신과 한 번도 교차하지 않는 투어를 말합니다. P의 최단 투어는 반드시 단순함을 증명하십시오.

(c) 가장 왼쪽 점 $P[1]$에서 출발하여 가장 오른쪽 점 $P[r]$에서 끝나는 P의 최단 투어를 계산하는 효율적인 알고리즘을 설명하고 분석하십시오.

(d) 끝점에 아무런 제약이 없을 때 P의 최단 투어를 계산하는 효율적인 알고리즘을 설명하고 분석하십시오.

♥28. 외판원 문제를 $O(2^n \operatorname{poly}(n))$ 시간에 해결하는 알고리즘을 설명하고 분석하십시오. 간선에 가중치가 있고 정점이 n개인 무향 그래프 G가 주어질 때, 여러분의 알고리즘은 모든 정점을 정확히 한 번씩 방문하는 G의 사이클 중 가장 가벼운 사이클의 가중치를 반환해야 하며, G에 그러한 사이클이 없으면 ∞를 반환해야 합니다. *[힌트: 뻔한 재귀 백트래킹 알고리즘은 $O(n!)$ 시간이 걸립니다.]*

29. 어떤 고정된 알파벳 Σ 위의 문자열들로 이루어진 유한 집합을 $W = \{w_1, w_2, \ldots, w_n\}$이라고 합시다. W의 *편집 중심*은 C에서 W에 속한 임의의 문자열까지의 최대 편집 거리가 가능한 한 작아지는 문자열 $C \in \Sigma^*$입니다. W의 *편집 반지름*은 편집 중심에서 W에 속한 문자열까지의 최대 편집 거리입니다. 문자열 집합에는 편집 중심이 여러 개 있을 수 있지만, 편집 반지름은 유일합니다.

$$\begin{aligned} \mathit{EditRadius}(W) &:= \min_{C \in \Sigma^*} \max_{w \in W} \mathit{Edit}(w, C) \\ \mathit{EditCenter}(W) &:= \arg\min_{C \in \Sigma^*} \max_{w \in W} \mathit{Edit}(w, C) \end{aligned}$$

(a) 주어진 세 문자열의 편집 반경을 계산하는 효율적인 알고리즘을 설명하고 분석하십시오.

♣♥(b) 임의의 문자열 집합의 편집 반경을 2배 이내로 근사하는 효율적인 알고리즘을 설명하고 분석하십시오. (문자열의 개수가 고정되어 있지 않으면 *정확한* 편집 반경을 계산하는 것은 NP-난해입니다.)

♥30. $D[1..n]$을 각 원소가 0 이상 9 이하의 정수인 숫자 배열이라고 합시다. D의 ***숫자 부분 수열***은 D의 서로 겹치지 않는 부분 문자열들로부터 통상적인 방식으로 만들어지는 양의 정수들의 수열입니다. 예를 들어, 수열 $3, 4, 5, 6, 8, 9, 32, 38, 46, 64, 83, 279$는 π의 처음 몇 자리 숫자의 숫자 부분 수열입니다:

$$3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 6, 4, 3, 3, 8, 3, 2, 7, 9$$

디지털 부분 수열의 *길이*는 그 부분 수열에 포함된 정수의 개수이며, 자릿수가 *아닙니다*. 앞의 예의 길이는 12입니다. 늘 그렇듯이, 디지털 부분 수열의 각 수가 바로 앞의 수보다 크면 그 부분 수열은 ***증가***한다고 합니다. D의 최장 증가 디지털 부분 수열을 계산하는 효율적인 알고리즘을 설명하고 분석하십시오. *[힌트: 계산 모델에 관한 가정에 주의하십시오. k자리 수 두 개를 비교하는 데 시간이 얼마나 걸립니까?]* 만점을 받으려면 알고리즘이 $O(n^4)$ 시간에 실행되어야 하며, 더 빠른 알고리즘에는 추가 점수를 드립니다. 이 문제에 대해 제가 아는 가장 빠른 알고리즘은 $O(n^{3/2}\log n)$ 시간에 실행됩니다. 이 한계를 달성하려면 알고리즘의 설계와 분석 양쪽에서 몇 가지 요령이 필요하지만, 이 수업의 범위를 벗어나는 것은 없습니다.[22]

♥31. 고전적인 하노이의 탑 문제를 다음과 같이 변형한 문제를 생각해 봅시다. 여느 때와 같이 크기가 모두 다른 n개의 원판이 0, 1, 2번으로 번호가 매겨진 세 기둥에 놓여 있습니다. 처음에는 n개의 원판이 모두 기둥 0에 크기순으로 쌓여 있으며, 가장 작은 원판이 맨 위에, 가장 큰 원판이 맨 아래에 있습니다. 우리의 목표는 모든 원판을 기둥 2로 옮기는 것입니다. 한 단계에서는 두 가지 제약 조건을 만족하는 한, 어느 기둥에서든 가장 위에 있는 원판을 다른 기둥으로 옮길 수 있습니다. 첫째, 더 큰 원판 위에 더 작은 원판을 절대 올려놓아서는 안 됩니다. 둘째로, 그리고 이 부분이 표준 문제와 다른 점인데, ***우리는 원판을 기둥 0에서 기둥 2로 직접 옮겨서는 절대 안 됩니다.*** 위의 제약 조건 아래에서 n개의 원판을 모두 기둥 0에서 기둥 2로 옮기는 데 필요한 정확한 이동 횟수를 계산하는 알고리즘을 설명하고 분석하십시오. 만점을 받으려면 알고리즘이 최악의 경우에도 $O(\log n)$번의 산술 연산만 사용해야 합니다. 분석의 편의를 위해 두 k자리 수를 더하거나 곱하는 데 $O(k)$ 시간이 걸린다고 가정하십시오. *[힌트: 행렬을 이용하십시오!]*

수열/배열 분할

32. ***기본 산술 식***은 집합 $\{1, +, \times\}$의 문자와 괄호로 이루어집니다. 거의 모든 정수는 둘 이상의 기본 산술식으로 나타낼 수 있습니다. 예를 들어, 다음 기본 산술식은

[22] 더 고급 기법을 사용하면 실행 시간을 $O(n^{3/2}\log\log n)$까지 줄일 수 있다고 생각하지만, 세부 사항까지 확인해 보지는 않았습니다.

모두 정수 14를 나타냅니다:

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1$$
$$((1+1)\times(1+1+1+1+1))+((1+1)\times(1+1))$$
$$(1+1)\times(1+1+1+1+1+1+1)$$
$$(1+1)\times(((1+1+1)\times(1+1))+1)$$

정수 n이 입력으로 주어졌을 때, 값이 n인 기본 산술식에 들어가는 1의 최소 개수를 계산하는 알고리즘을 설명하고 분석하십시오. 괄호의 개수는 상관없으며, 1의 개수만 중요합니다. 예를 들어 $n = 14$일 때 알고리즘은 위의 마지막 식에 해당하는 8을 반환해야 합니다. 알고리즘의 실행 시간은 n에 대한 작은 다항 함수로 제한되어야 합니다.

33. + 기호와 − 기호로 구분된 정수 수열이 주어졌다고 가정합시다. 예를 들면 다음과 같습니다:

$$1+3-2-5+1-6+7$$

괄호를 다른 위치에 넣으면 이 식의 값을 바꿀 수 있습니다. 예를 들면 다음과 같습니다.

$$1+3-2-5+1-6+7=-1$$
$$(1+3-(2-5))+(1-6)+7=9$$
$$(1+(3-2))-(5+1)-(6+7)=-17$$

+와 − 기호로 구분된 정수 목록이 주어졌을 때, 괄호를 추가하여 이 식이 가질 수 있는 최댓값을 계산하는 알고리즘을 설명하고 분석하십시오. 괄호는 덧셈과 뺄셈을 묶는 데에만 사용해야 합니다. 특히 $1+3(-2)(-5)+1-6+7=33$처럼 괄호를 사용하여 암묵적인 곱셈을 만들어서는 안 됩니다.

34. + 기호와 × 기호로 구분된 정수들의 수열이 주어졌다고 가정합시다. 예를 들면 다음과 같습니다.

$$1+3\times2\times0+1\times6+7$$

괄호를 넣는 위치를 달리하면 이 식의 값을 바꿀 수 있습니다. 예를 들면 다음과 같습니다.

$$(1+(3\times2))\times0+(1\times6)+7=13$$
$$((1+(3\times2\times0)+1)\times6)+7=19$$
$$(1+3)\times2\times(0+1)\times(6+7)=104$$

(a) 입력의 모든 정수가 *양수*라고 가정할 때, 괄호를 추가하여 주어진 식이 가질 수 있는 최댓값을 계산하는 알고리즘을 설명하고 분석하십시오. *[힌트: 이 문제는 쉽습니다.]*

(b) 입력의 모든 정수가 *음이 아니라고* 가정할 때, 주어진 식에 괄호를 추가하여 얻을 수 있는 최댓값을 계산하는 알고리즘을 설명하고 분석하십시오.

(c) 입력 숫자에 아무런 제약이 없을 때, 주어진 식에 괄호를 추가하여 얻을 수 있는 최댓값을 계산하는 알고리즘을 설명하고 분석하십시오.

모든 산술 연산에 $O(1)$ 시간이 걸린다고 가정합니다.

35. Sham-Poobanana 대학교를 졸업한 뒤, 여러분은 월스트리트 은행 ***Long Live Boole***의 한 직책에 면접을 보기로 합니다. 이 은행의 전무이사 Eloob Egroeg는 모든 신입 직원에게 '풀거나 죽거나' 문제를 내며, 직원은 이 문제를 24시간 안에 풀어야 합니다. 문제를 풀지 못한 사람은 즉시 해고됩니다! 처음으로 은행에 들어선 여러분은 직원 사무실들이 일직선으로 늘어서 있고, 각 사무실 문에 커다란 T 또는 F가 적혀 있다는 것을 알아차립니다. 게다가 인접한 두 사무실 사이마다 $\wedge, \vee, \oplus$ 중 하나의 기호가 표시된 판이 있습니다. 이 불가사의한 기호들에 대해 묻자, Eloob는 T와 F가 불리언 값 참(True)과 거짓(False)을 나타내고, 판에 적힌 기호들은 표준 불리언 연산자 And, Or, Xor를 나타낸다고 확인해 줍니다. 그는 또한 이 문자와 기호들이 특정한 직원 조합이 함께 일을 성공적으로 해낼 수 있는지를 나타낸다고 설명합니다. 새 프로젝트를 시작할 때마다 Eloob는 기호 수열에 괄호를 추가하여 모호하지 않은 불리언 식을 얻는 방식으로 직원들을 계층적으로 묶습니다. 이렇게 괄호를 친 불리언 식의 값이 T이면 프로젝트는 성공합니다. 예를 들어, 은행에 직원이 세 명이고 그들의 문 위와 문 사이에 있는 기호 수열이 $T \wedge F \oplus T$라면, 성공하는 괄호 묶음 방식은 $(T \wedge (F \oplus T))$ 정확히 하나뿐입니다. 그러나 문 기호 목록이 $F \wedge T \oplus F$라면, 괄호를 추가하여 프로젝트를 성공시킬 방법은 없습니다. 마침내 Eloob는 여러분에게 풀거나 죽거나 면접 문제를 냅니다: 주어진 기호 수열에 괄호를 쳐서 결과 불리언 식의 값이 T가 되도록 할 수 있는지 판정하는 알고리즘을 설명하십시오. 입력은 배열 $S[0..2n]$이며, i가 짝수일 때 $S[i] \in \{T, F\}$이고 i가 홀수일 때 $S[i] \in \{\vee, \wedge, \oplus\}$입니다.

36. 어퍼 글레이시어빌 남극 달팽이 애호가 협회는 매년 연례 회의의 일환으로 원탁 짝짓기 경주를 개최합니다. 우수한 번식용 달팽이 여러 마리를 원탁의 가장자리에 올려놓습니다. 달팽이들에게는 탁자 둘레를 따라 차례대로 1부터 n까지 번호가 매겨져 있습니다. 경주가 진행되는 동안 각 달팽이는 뒤에 점액 자국을 남기며 탁자 위를 돌아다닙니다. 달팽이들은 탁자 가장자리 밖으로 떨어지거나 점액 자국을 가로지르지 않도록 특별히 훈련받았으며, 자기 자신이 남긴 자국도 가로지르지 않습니다. 두 달팽이가 만나면 번식 쌍으로 선언되어 탁자에서 치워지고, 귀여운 아기 달팽이를 낳을 수 있도록 땅속의 낭만적인 구멍으로 재빨리 옮겨집니다. 경주가 영원히 계속되더라도 끝내 짝을 찾지 못하는 달팽이가 있을 수 있다는 점에 유의하십시오. 남극 SLUG 경주 주최 측은 모든 달팽이 쌍마다 상금을 걸어 두었으며, 이 상금은 해당 달팽이 쌍이 짝짓기 경주 도중 만나면 주인들에게 지급됩니다. 구체적으로, 원탁 뒤 벽에는 2차원 배열 $M[1..n, 1..n]$이 게시되어 있는데, 여기서 $M[i, j] = M[j, i]$는 달팽이 i와 j가 만날 경우 지급되는 상금입니다. 배열 M이 입력으로 주어질 때, 주최 측이 지급해야 할 수도 있는 최대 총상금을 계산하는 알고리즘을 설명하고 분석하십시오.

37. 여러분은 채석장에서 커다란 대리석 판을 캐냈습니다. 간단히 하기 위해, 이 대리석 판이 높이 n인치, 너비 m인치인 직사각형이라고 가정합시다. 여러분은 이 판을 다양한 크기의 더 작은 직사각형들로 자르려고 합니다. 일부는 주방 조리대

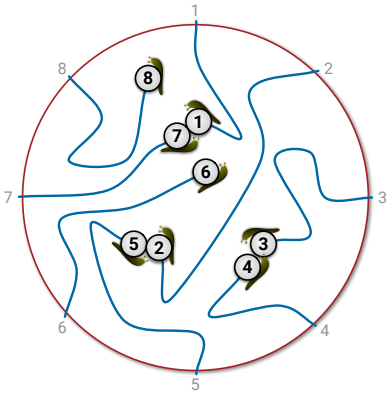


그림 3.6. 전형적인 남극 SLUG 경주의 마지막 모습입니다. 달팽이 6과 8은 끝내 짝을 찾지 못합니다. 주최 측은 $M[3,4] + M[2,5] + M[1,7]$을 지불해야 합니다.

상판용이고, 일부는 대형 조각 작품용이며, 나머지는 추모 비석용입니다. 여러분에게는 어떤 직사각형 판이든 가로 또는 세로로 가로질러 자를 수 있는 대리석 톱이 있습니다. 여러분은 언제든지 임의의 양의 정수 x와 y에 대해 x인치 × y인치 대리석 직사각형의 현물 가격 $P[x, y]$를 조회할 수 있습니다. 이 가격은 고객 수요에 따라 달라지고, 대리석 조리대 상판을 사는 사람들은 별나므로 가격에 대해 어떠한 가정도 하지 마십시오. 특히 더 큰 직사각형의 현물 가격이 훨씬 더 낮을 수도 있습니다. 현물 가격 배열과 정수 m, n이 입력으로 주어질 때, 이익이 최대가 되도록 $n \times m$ 대리석 판을 나누는 방법을 계산하는 알고리즘을 설명하십시오.

38. 이 문제에서는 추가적인 균형 제약 조건을 만족하는 최적 이진 탐색 트리를 구성하는 효율적인 알고리즘을 설계해야 합니다. 입력은 탐색 키의 정렬된 배열 $A[1..n]$과 빈도 횟수의 배열 $f[1..n]$으로 이루어지며, 여기서 $f[i]$는 $A[i]$에 대한 탐색 횟수입니다. 이는 3.9절에서 설명한 비용 함수와 정확히 같습니다. 그러나 이제 여러분의 과제는 몇 가지 추가 제약 조건을 만족하는 최적 트리를 계산하는 것입니다.

   (a) ***AVL 트리***는 최초의 자가 균형 이진 탐색 트리로, 1962년 게오르기 아델손 벨스키(Georgy Adelson-Velsky)와 예브게니 란디스(Evgenii Landis)가 처음 기술하였습니다. AVL 트리는 모든 노드 v에 대해 v의 왼쪽 부분 트리의 높이와 v의 오른쪽 부분 트리의 높이가 최대 1만큼 차이 나는 이진 탐색 트리입니다. 주어진 탐색 키 집합과 빈도에 대해 최적의 AVL 트리를 구성하는 알고리즘을 기술하고 분석하십시오.

   (b) ***대칭 이진 B-트리(symmetric binary B-tree)***는 또 다른 자가 균형 이진 트리로, 1972년에 루돌프 바이어(Rudolf Bayer)가 처음 기술했습니다. 이 트리는 1978년에 레오 기바스(Leo Guibas)와 밥 세지윅(Bob Sedgwick)이 좀 더 단순하게 재정식화한 이후 ***레드-블랙 트리(red-black tree)***라는 이름

으로 더 잘 알려져 있습니다. 레드-블랙 트리는 다음과 같은 추가 제약 조건을 만족하는 이진 탐색 트리입니다:

- 모든 노드는 빨간색이거나 검은색입니다.
- 모든 빨간 노드의 부모는 검은색입니다.
- 루트에서 리프까지의 모든 경로에는 같은 수의 검은색 노드가 있습니다.

주어진 탐색 키와 빈도의 집합에 대해 최적 레드-블랙 트리를 구성하는 재귀 백트래킹 알고리즘을 설명하십시오.

(c) ***AA 트리***는 1993년 아르네 안데르손(Arne Andersson)이 제안했으며, 2000년 마크 앨런 와이스(Mark Allen Weiss)가 약간 단순화하고 이름을 붙였습니다. 2006년 밥 세지윅(Bob Sedgewick)이 (다른 재균형 알고리즘을 사용하여) 대칭적으로 재정식화한 이후, AA 트리는 *왼쪽으로 기운 레드-블랙 트리*로도 알려져 있습니다. AA 트리는 다음과 같은 제약 조건이 하나 더 추가된 레드-블랙 트리입니다:

- 어떤 왼쪽 자식도 빨간색이 아닙니다.[23]

주어진 탐색 키와 빈도의 집합에 대해 최적의 AA 트리를 구성하는 알고리즘을 설명하고 분석하십시오.

39. $m \times n$ 비트맵이 0과 1로 이루어진 배열 $M[1..m, 1..n]$로 주어졌다고 가정합시다. M의 *균일 블록*은 모든 비트가 같은 $M[i..i', j..i']$ 형태의 부분 배열입니다. M을 가능한 한 적은 수의 서로소인 블록으로 분해하려 한다고 가정합시다. 자연스러운 재귀적 분할 전략 중 하나를 *길로틴 분할*이라고 부릅니다. 비트맵 M 전체가 균일 블록이면 더 할 일이 없습니다. 그렇지 않으면 M을 수평선이나 수직선을 따라 더 작은 두 비트맵으로 자른 다음, 그 두 비트맵을 재귀적으로 균일 블록으로 분해합니다. 모든 길로틴 분할은 이진 트리로 나타낼 수 있습니다. 이 트리에서 각 내부 노드는 절단의 위치와 방향을 저장하고, 각 리프는 해당 블록의 내용을 나타내는 비트 0 또는 1 하나를 저장합니다. 길로틴 분할의 *크기*는 대응하는 이진 트리의 리프 수(즉, 최종 균일 블록의 수)이고, 길로틴 분할의 *깊이*는 대응하는 이진 트리의 깊이입니다.

(a) 크기가 가능한 한 가장 작은 M의 길로틴 분할을 계산하는 알고리즘을 설명하고 분석하십시오.

(b) 길로틴 분할이 항상 가장 적은 수의 단색 블록으로 이루어진 분할을 만들어 내는 것은 *아님*을 보이십시오.

(c) M에 대해 깊이가 가능한 한 가장 작은 길로틴 분할을 계산하는 알고리즘을 서술하고 분석하십시오.

(d) M의 길로틴 분해를 나타내는 트리와 두 인덱스 i, j가 주어졌을 때, $M[i, j]$를 구하는 알고리즘을 설명하고 분석하십시오.

[23] Sedgwick의 재정식화에서는 어떤 *오른쪽* 자식도 빨간색이 아니어야 합니다. 뭐, 아무래도 좋습니다. Andersson과 Sedgewick은 각도를 시계 방향으로 재야 하는지 반시계 방향으로 재야 하는지, 명왕성이

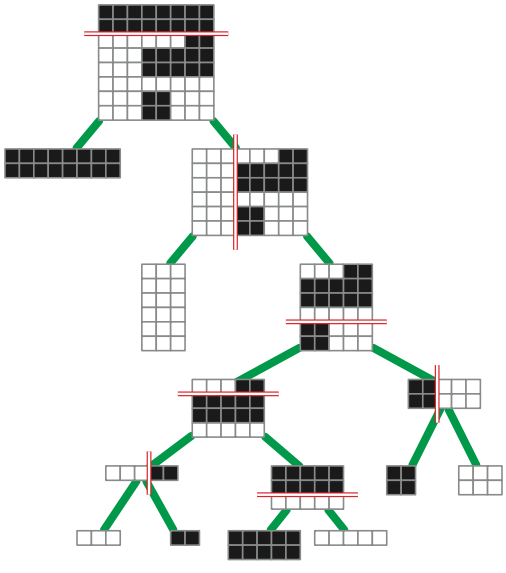


그림 3.7. 크기가 8이고 깊이가 5인 길로틴 분할입니다.

(e) 길로틴 분할에서 픽셀 $M[i,j]$의 *깊이*를 그 픽셀을 포함하는 리프의 깊이로 정의합니다. 픽셀들의 깊이의 합이 가능한 한 작아지도록 M에 대한 길로틴 분할을 계산하는 알고리즘을 설명하고 분석하십시오.

(f) *검은색* 픽셀들의 깊이 합이 가능한 한 작아지도록 M에 대한 길로틴 분할을 계산하는 알고리즘을 설명하고 분석하십시오.

♠40. 축하합니다! 여러분은 거대 온라인 서점 DeNile("이집트에 있는 강만이 아닙니다!")에 창고 로봇을 최적화하는 일로 채용되었습니다. DeNile이 판매하는 모든 책에는 고유한 ISBN(국제 표준 도서 번호)이 있으며, 이는 단순한 수치 값입니다. DeNile의 각 창고에는 상자들이 길게 한 줄로 늘어서 있고, 각 상자에는 한 가지 책이 여러 권 들어 있습니다. 이 상자들은 ISBN 순서대로 정렬되어 있으며, 각 상자의 ISBN은 상자 앞면에 기계가 읽을 수 있는 형태로 인쇄되어 있습니다. 책은 상자 줄과 평행하게 놓인 레일을 따라 움직이는 로봇이 이 상자들에서 꺼내옵니다. DeNile은 어느 상자에 어떤 ISBN 번호가 들어 있는지에 대한 목록을 관리하지 않습니다. 그렇게 하면 너무 간단하기 때문입니다! 대신 원하는 책을 꺼내려면 로봇은 먼저 이진 탐색으로 그 책의 상자를 찾아야 합니다. 탐색에는 로봇의 물리적 이동이 필요하므로, 더 이상 이진 탐색의 각 단계에 $O(1)$ 시간이 걸린다고 가정할 수 없습니다. 구체적으로는 다음과 같습니다.

- 로봇은 항상 "0번째 칸"(고객에게 배송할 책을 상자에 담는 곳)에서 출발합니다.

행성인지, "낮은 순위"가 "더 좋다"는 뜻인지 "더 나쁘다"는 뜻인지, 그리고 오리만 한 말 백 마리와 싸우는 편이 나은지 말만 한 오리 한 마리와 싸우는 편이 나은지에 대해서는 이상하리만치 아무 말도 하지 않습니다.

- 로봇을 i번째 통에서 j번째 통으로 옮기는 데에는 어떤 상수 α에 대해 $\alpha|i-j|$초가 걸립니다.
- 로봇은 어떤 통의 ISBN을 읽으려면 그 통 바로 앞에 있어야 합니다. ISBN 하나를 읽는 데에는 어떤 상수 β에 대해 β초가 걸립니다.
- 로봇의 이동 방향을 반대로 바꾸려면(증가에서 감소로, 또는 그 반대로) 어떤 상수 γ에 대해 γ초가 추가로 필요합니다.
- 로봇은 대상 상자를 찾으면 그 상자에서 책 한 권을 꺼낸 뒤 "0번째 상자"로 돌아갑니다.

로봇이 책을 찾는 데 쓰는 총 시간을 최소화하도록 칸들에 대한 이진 탐색 트리를 계산하는 알고리즘을 설계하고 분석하십시오. 입력으로는 정수 배열 $f[1..n]$과 시간 매개변수 α, β, γ가 주어집니다. 여기서 $f[i]$는 로봇이 i번째 칸에서 책을 꺼내 오도록 요청받는 횟수입니다.

♠41. 탐색 트리의 캐시 성능을 개선하는 표준적인 방법은 각 노드에 더 많은 탐색 키와 부분 트리를 담는 것입니다. ***B*-트리**는 각 내부 노드가 최대 B개의 키와 최대 $B+1$개의 자식을 가리키는 포인터를 저장하는 루트 트리이며, 각 자식은 더 작은 B-트리의 루트입니다. 구체적으로, 각 노드 v는 세 개의 필드를 저장합니다:

- 양의 정수 $v.d \le B$,
- *정렬된* 배열 $v.key[1..v.d]$, 그리고
- 자식 포인터들의 배열 $v.child[0..v.d]$입니다.

특히 자식 포인터의 개수는 항상 키의 개수보다 정확히 하나 더 많습니다.[24] 각 포인터 $v.child[i]$는 NULL이거나, 모든 키가 $v.key[i]$보다 크고 $v.key[i+1]$보다 작은 B-트리의 루트를 가리키는 포인터입니다. 특히 가장 왼쪽 부분 트리 $v.child[0]$의 모든 키는 $v.key[1]$보다 작고, 가장 오른쪽 부분 트리 $v.child[v.d]$의 모든 키는 $v.key[v.d]$보다 큽니다. 직관적으로는 다음과 같은 그림을 떠올리면 됩니다.

$$\begin{array}{ccccccccccc} [\bullet & < key[1] < & \bullet & < key[2] < & \bullet & \cdots & \bullet & < key[d] < & \bullet] \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\ T_0 & & T_1 & & T_2 & \cdots & T_{d-1} & & T_d \end{array}$$

여기서 T_i는 $child[i]$가 가리키는 부분 트리입니다. B-트리에서 키 x를 탐색하는 ***비용***은 루트에서 x를 키 중 하나로 포함하는 노드까지의 경로에 있는 노드의 개수입니다. 1-트리는 표준 이진 탐색 트리일 뿐입니다. 임의의 양의 정수 $B > 0$을 고정합시다. ($B = 8$을 권합니다.) 탐색 키의 정렬된 배열 $A[1, ..., n]$과 이에 대응하는 빈도수 배열 $F[1, ..., n]$이 주어졌다고 가정합시다. 여기서 $F[i]$는 $A[i]$를 탐색할 횟수입니다. 주어진 빈도로 주어진 키들을 탐색하는 총비용을 최소화하는 B-트리를 찾는 효율적인 알고리즘을 설명하고 분석하십시오.

[24] 보통 B-트리는 최악의 경우 탐색 비용 $O(\log_B n)$을 보장하는 두 가지 추가 제약 조건을 만족해야 합니다. 모든 리프는 정확히 같은 깊이를 가져야 하고, 루트를 제외한 모든 노드는 적어도 $B/2$개의 키를 포함해야 합니다. 그러나 이 문제에서는 *최악의 경우* 탐색 비용이 아니라 일련의 탐색에 드는 *총* 비용을 최적화하는 데 관심이 있으므로, 이러한 추가 제약 조건은 부과하지 않겠습니다.

(a) $B = 2$인 특수한 경우에 대한 다항 시간 알고리즘을 설명하십시오.

(b) 임의의 B에 대해 어떤 고정된 정수 c에 대하여 $O(n^{B+c})$ 시간에 실행되는 알고리즘을 설명하십시오.

♥(c) 임의의 B에 대해, B에 의존하지 않는 어떤 고정된 정수 c에 대하여 $O(n^c)$ 시간에 실행되는 알고리즘을 설명하십시오.

42. 괄호 `(`, `)`와 대괄호 `[`, `]`로 이루어진 문자열 w가 다음 조건 중 하나를 만족하면 w는 ***균형 잡혀*** 있다고 합니다.

- w는 빈 문자열입니다.
- 어떤 균형 문자열 x에 대해 $w =$ `(`x`)`인 경우
- 어떤 균형 문자열 x에 대해 $w =$ `[`x`]`
- 어떤 균형 잡힌 문자열 x와 y에 대해 $w = xy$인 경우

예를 들어, 다음 문자열은

$$w = \texttt{([()][]())[()()]()}$$

$w = xy$이므로 균형을 이룹니다. 여기서

$$x = \texttt{([()][]())} \text{ and } y = \texttt{[()()]()}.$$

(a) 소괄호와 대괄호로 이루어진 문자열이 주어졌을 때, 그 문자열이 균형을 이루는지 판별하는 알고리즘을 설명하고 분석하십시오.

(b) 소괄호와 대괄호로 이루어진 문자열이 주어질 때, 균형 잡힌 부분 수열 중 가장 긴 것의 길이를 계산하는 알고리즘을 설명하고 분석하십시오.

(c) 괄호와 대괄호로 이루어진 문자열이 주어졌을 때, 그 문자열의 가장 짧은 균형 잡힌 초수열의 길이를 계산하는 알고리즘을 설명하고 분석하십시오.

(d) 주어진 소괄호와 대괄호로 이루어진 문자열에서 균형 잡힌 소괄호와 대괄호 문자열까지의 최소 편집 거리를 계산하는 알고리즘을 설명하고 분석하십시오.

♥(e) 소괄호와 대괄호로 이루어진 두 문자열이 주어질 때, 두 문자열의 최장 공통 균형 부분 수열을 계산하는 알고리즘을 설명하고 분석하십시오.

♥(f) 소괄호와 대괄호로 이루어진 문자열이 주어질 때, 그 문자열에서 회문이면서 균형 잡힌 부분 수열 중 가장 긴 것을 계산하는 알고리즘을 설명하고 분석하십시오.

♥(g) 괄호와 대괄호로 이루어진 두 문자열이 주어질 때, 두 문자열의 최장 공통 회문 균형 부분 수열(휴!)을 계산하는 알고리즘을 설명하고 분석하십시오.

각 문제에서 입력은 배열 $w[1..n]$이며, 모든 인덱스 i에 대해 $w[i] \in \{$`(`, `)`, `[`, `]`$\}$입니다. (괄호와 대괄호 대신 다른 기호, 예를 들어 `L`, `R`, `l`, `r` 또는 ◁, ▷, ◀, ▶를 사용하는 편을 선호할 수도 있습니다. 다만 어떤 기호를 사용하는지 채점자에게 꼭 알려 주십시오!)

♥43. 축하합니다! 여러분의 연구팀이 DARPA, Google, McDonald's가 공동으로 자금을 지원하는 $50M 규모의 다년간 프로젝트를 방금 수주했습니다. 이 프로젝트의 목표는 DWIM, 즉 프로그래머의 마음을 읽는 최초의 컴파일러를 만드는 것입니다! 여러분의 제안서와 수많은 보도 자료는 모두 DWIM이 어떤 코드가 주어지든 그 코드를 가능한 한 적게 수정하면서 오류를 자동으로 고쳐 줄 것이라고 약속합니다. 안타깝게도 이제는 이 망할 물건이 실제로 작동하도록 만들기 시작해야 할 때입니다. 준비 운동 삼아 여러분은 다음의 필수적인 부분 문제부터 다루기로 합니다. 두 문자열 사이의 *편집 거리*는 한 문자열을 다른 문자열로 바꾸는 데 필요한 한 문자 삽입, 삭제, 치환의 최소 횟수라는 것을 떠올려 보십시오. *산술식*이란 다음 조건을 만족하는 문자열 w입니다.

- w는 하나 이상의 십진 숫자로 이루어진 문자열이며,
- 어떤 산술식 x에 대해 $w = (x)$이거나,
- 어떤 산술식 x와 y, 그리고 어떤 이항 연산자 $\diamond$에 대해 $w = x \diamond y$입니다.

알파벳 {#, ◇, (,)}의 토큰으로 이루어진 문자열이 주어졌다고 가정합시다. 여기서 #은 십진수 숫자 하나를 나타내고 ◇는 이항 연산자를 나타냅니다. 주어진 문자열에서 산술식까지의 최소 편집 거리를 계산하는 알고리즘을 설명하고 분석하십시오.

44. 리보핵산(RNA) 분자는 아데닌(A), 사이토신(C), 구아닌(G), 우라실(U)의 네 가지 유형으로 이루어진 수백만 개의 뉴클레오타이드, 즉 *염기*의 긴 사슬입니다. RNA 분자의 *서열*은 문자열 $b[1..n]$이며, 여기서 각 문자 $b[i] \in \{A, C, G, U\}$는 하나의 염기에 대응합니다. 서열에서 인접한 염기 사이의 화학 결합 외에도, 특정한 염기 쌍 사이에는 수소 결합이 형성될 수 있습니다. 결합된 염기쌍의 집합을 RNA 분자의 *이차 구조*라고 합니다. $i < j$이고 $i' < j'$인 두 염기쌍 (i, j)와 (i', j')에 대해 $i < i' < j < j'$ 또는 $i' < i < j' < j$이면 두 염기쌍이 ***겹친다***고 합니다. 실제로는 대부분의 염기쌍이 겹치지 않습니다. 겹치는 염기쌍은 이차 구조에 이른바 *슈도노트*를 만들어 내는데, 이는 일부 RNA 함수에 필수적이지만 예측하기가 더 어렵습니다. 주어진 RNA 서열에 대해 가능한 최선의 이차 구조를 예측하고 싶다고 가정합시다. 우리는 이차 구조에 대해 대폭 단순화된 다음 모델을 채택하겠습니다.

- 각 염기는 다른 염기 최대 하나와만 결합할 수 있습니다.
- A–U 쌍과 C–G 쌍만 결합할 수 있습니다.
- $(i, i+1)$ 및 $(i, i+2)$ 형태의 쌍은 결합할 수 없습니다.
- 결합된 염기쌍은 서로 겹칠 수 없습니다.

마지막(그리고 가장 비현실적인) 제약 조건 덕분에 아래 그림과 같이 RNA 이차 구조를 일종의 뚱뚱한 트리로 시각화할 수 있습니다.

(a) 주어진 RNA 서열에 대해, 이차 구조에서 결합된 염기쌍의 가능한 최대 *개수*를 계산하는 알고리즘을 설명하고 분석하십시오.

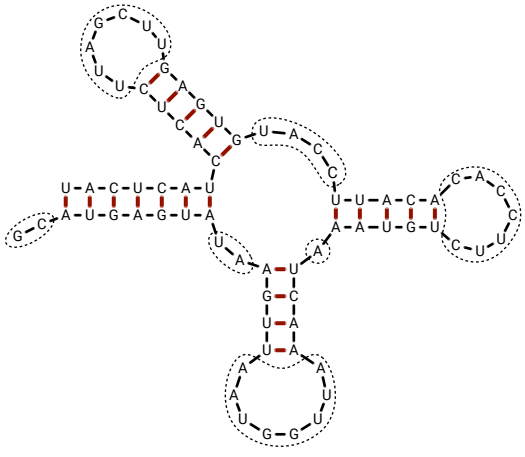


그림 3.8. 굵은 빨간 선으로 표시된 21개의 결합 염기쌍을 가진 RNA 이차 구조의 예시입니다. 간격은 점선 곡선으로 표시되어 있습니다. 이 구조의 점수는 $2^2 + 2^2 + 8^2 + 1^2 + 7^2 + 4^2 + 7^2 = 187$입니다.

(b) 이차 구조에서 *간극*은 짝을 이루지 않은 염기들로 이루어진 극대 부분 문자열입니다. 간극이 크면 화학적으로 불안정해지므로, 간극이 작은 이차 구조일수록 나타날 가능성이 더 높습니다. 이러한 경향을 반영하기 위해, 이차 구조의 *점수*를 간극 길이의 *제곱*의 합으로 정의합시다. 그림 3.8을 참고하십시오. (이 점수 함수는 완전히 지어낸 것입니다. 실제 RNA 구조 예측에는 *훨씬* 더 복잡한 점수 함수가 필요합니다.) 주어진 RNA 서열에 대해 이차 구조가 가질 수 있는 최소 점수를 계산하는 알고리즘을 설명하고 분석하십시오.

♣45.

(a) 문자열 w와 정규 표현식 R이 주어졌을 때 $w \in L(R)$인지 판정하는 효율적인 알고리즘을 설명하고 분석하십시오.

(b) *일반화된* 정규 표현식에서는 통상적인 • (연결), + (또는), * (클레이니 닫힘) 연산자 외에 이항 연산자 ∩ (교집합)과 단항 연산자 ¬ (여집합)도 사용할 수 있습니다. NFA 구성과 클레이니 정리에 따르면 모든 일반화된 정규 표현식 E는 정규 언어 $L(E)$를 나타냅니다. 문자열 w와 일반화된 정규 표현식 E가 주어졌을 때 $w \in L(E)$인지 판정하는 효율적인 알고리즘을 설명하고 분석하십시오.

두 문제 모두, 단순한 문자열이 아니라 (일반화된) 정규 표현식의 파스 트리가 실제로 주어진다고 가정하십시오.

트리와 부분 트리

46. 여러분은 방금 Giggle(Abugida의 자회사)에서 처음 열리는 연례 의무 참석 연말 파티의 새 기획자로 임명되었습니다. Giggle의 직원들은 엄격한 위계 구조, 즉 회사 사장을 루트로 하는 트리로 조직되어 있습니다. 인사부의 전지전능한 오라클

들은 각 직원이 얼마나 "재미있는지"를 나타내는 실수를 직원마다 하나씩 부여해 두었습니다. 사교적인 분위기를 유지하기 위해 초대 명단에는 한 가지 제약이 있습니다. 직원의 직속 상사가 파티에 참석하면 그 직원은 참석할 수 없습니다. 한편 회사 사장은 재미 점수가 음수이더라도 파티에 *반드시* 참석해야 합니다. 어쨌든 사장의 회사이기 때문입니다. 참석자들의 "재미" 점수 합이 최대가 되도록 파티 초대 명단을 만드는 알고리즘을 제시하십시오.

47. 작년 연말 파티에 온 사람이 너무 적었기 때문에, Giggle의 사장은 올해는 파티 대신 직원들에게 각각 선물을 주기로 합니다. 구체적으로, 각 직원은 다음 세 가지 선물 중 하나를 받아야 합니다: (1) 전 세계 어디든 갈 수 있는 전액 지원 6주 휴가, (2) Jumping Jack Flash's Flapjack Stack Shack에서 정렬할 수 있는 만큼 팬케이크를 먹을 수 있는 2인 아침 식사, 또는 (3) 개똥이 가득 든 불타는 종이봉투. 회사 규정상 어떤 직원도 자신의 직속 상사와 정확히 같은 선물을 받을 수 없습니다. 직속 상사보다 더 좋은 선물을 받은 직원은 질투에 사로잡힌 상사에 의해 거의 확실히 해고됩니다. Giggle의 공식 파티 총책임자로서, 각 직원이 어떤 선물을 받을지 결정하는 것은 *여러분*의 일입니다. 해고되는 사람의 수가 최소가 되도록 선물을 나누어 주는 알고리즘을 설명하십시오. 그렇습니다, 사장에게 불타는 개똥 봉투를 보내도 됩니다. 좀 더 형식적으로 말하면, 회사의 위계 구조를 나타내는 루트 트리 T가 주어지며, 모든 노드가 부모와 다른 레이블을 갖도록 T의 노드에 정수 1, 2, 3 중 하나로 레이블을 붙이려고 합니다. 레이블링의 *비용*은 부모보다 작은 레이블을 가진 노드의 수입니다. 예시는 그림 3.9를 참고하십시오. T의 최소 비용 레이블링을 계산하는 알고리즘을 설명하고 분석하십시오.

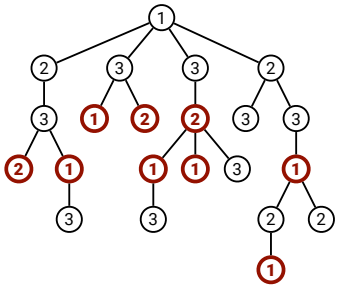


그림 3.9. 비용이 9인 트리 레이블링입니다. 굵게 표시된 아홉 개의 노드는 부모보다 작은 레이블을 가집니다. 이 레이블링은 이 트리의 최적 레이블링이 아닙니다.

48. 불타는 개똥 휴일 대참사 이후 여러분은 다른 일자리를 알아보라는 강한 권유를 받았고, 그래서 Giggle을 떠나 경쟁사인 Twitbook으로 옮겼습니다. 안타깝게도 Twitbook의 신임 사장은 Giggle을 따라 자신만의 휴일 파티를 열기로 막 결정했고, 여러분의 과거 경험을 감안하여 여러분을 공식 파티 기획자로 임명했습니다. 사장은 사장 자신을 포함하여 정확히 k명의 직원을 초대하라고 요구하며, 초대받은 사람은 모두 반드시 참석해야 합니다. 네, 참 재미있겠습니다. Giggle과 마찬가지로 Twitbook의 직원들도 엄격한 위계로 조직되어 있습니다. 즉, 회사 사

장을 루트로 하는 트리입니다. 인사부의 전지전능한 신탁들은 각 직원에게 실수를 하나씩 부여했는데, 이 값은 그 직원과 그 직속 상사를 함께 초대할 때의 *어색함*을 나타냅니다. 음수 값은 그 직원과 상사가 실제로는 서로를 좋아한다는 뜻입니다. 여러분의 목표는 초대할 직원을 정확히 k명 골라, 그 결과로 열리는 파티의 총 어색함을 최대한 작게 만드는 것입니다. 예를 들어 초대 명단에 직원과 그 직속 상사가 함께 들어 있는 경우가 하나도 없으면 총 어색함은 0입니다. 알고리즘의 입력은 트리 T, 정수 k, 그리고 T의 각 노드의 어색함입니다.

(a) 회사의 계층 구조가 *이진* 트리로 표현된다고 가정할 때, k명의 직원으로 이루어진 부분집합 중 어색함이 가장 적은 부분집합의 총 어색함을 계산하는 알고리즘을 설명하십시오. 즉, 각 직원이 최대 두 명의 다른 직원을 직접 감독한다고 가정하십시오.

♥(b) 회사 계층 구조에 아무런 제약이 없을 때, 직원 k명으로 이루어진 부분집합 가운데 가장 덜 어색한 부분집합의 총 어색함을 계산하는 알고리즘을 설명하십시오.

49. 루트 트리의 모든 노드에 메시지를 방송해야 한다고 가정합시다. 처음에는 루트 노드만 메시지를 알고 있습니다. 한 라운드에서 메시지를 알고 있는 노드는 자식 중 최대 하나에게 메시지를 전달할 수 있습니다. 예시는 그림 3.10을 참고하십시오.

(a) *이진* 트리의 모든 노드에 메시지를 전파하는 데 필요한 최소 라운드 수를 계산하는 알고리즘을 설계하십시오.

♦(b) *임의의* 루트 트리에서 모든 노드에 메시지를 전파하는 데 필요한 최소 라운드 수를 계산하는 알고리즘을 설계하십시오. *[힌트: 여러분의 알고리즘은 탐욕 알고리즘이 아니지만, 그 알고리즘이 올바름을 증명하는 데에는 다음 장의 기법이 유용할 수 있습니다.]*

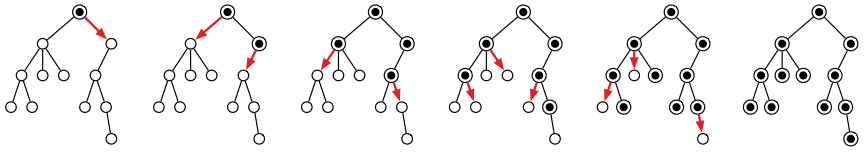


그림 3.10. 다섯 라운드에 걸쳐 트리를 통해 메시지가 전달되는 모습입니다.

50. 어느 날, 알렉스는 실내 암벽장에서 등반하는 데 싫증이 나서 아주 많은 클라이머 친구들을 데리고 야외로 등반하러 가기로 했습니다. 그들이 간 등반 지역에는 높지는 않지만 폭이 매우 넓은 거대한 바위가 있었고, 손과 발을 디딜 여러 홀드가 표시되어 있었습니다. 알렉스는 친구들이 한 홀드에서 다른 홀드로 이동할 때 수행할 수 있는 "허용된" 동작의 집합을 재빨리 정했습니다. 전체 홀드 체계는 n개의 정점을 가진 루트 트리 T로 나타낼 수 있으며, 각 정점은 홀드에, 각 간선은 홀드 사이의 허용된 동작에 대응합니다. 등반 경로는 바위를 올라갈수록 한데 모여

정상에 있는 유일한 홀드에 이르며, 이 홀드는 T의 루트로 표현됩니다.[25] 알렉스와 친구들(모두 뛰어난 클라이머입니다)은 가능한 한 많은 클라이머가 동시에 바위 위에 있고 각 클라이머가 *정확히* k번의 동작을 연속으로 수행해야 하는 게임을 하기로 했습니다. 각 클라이머는 임의의 홀드에서 출발할 수 있으며, 모든 동작은 지면에서 멀어지는 방향이어야 합니다. 따라서 각 클라이머는 트리 T에서 모두 루트를 향하는 k개의 간선으로 이루어진 경로를 따라갑니다. 하지만 두 클라이머가 같은 홀드를 건드리는 것은 허용되지 않습니다. 즉, 서로 다른 클라이머가 따라가는 경로는 전혀 교차할 수 없습니다. 이 게임에 참여할 수 있는 클라이머의 최대 수를 계산하는 효율적인 알고리즘을 설명하고 분석하십시오. 좀 더 형식적으로 말하면, 루트 트리 T와 정수 k가 주어졌을 때, 각 경로의 길이가 k인 T의 서로소 경로를 가능한 한 많이 찾고자 합니다. T가 이진 트리라고 가정하지 ***마십시오***. 예를 들어, 아래의 트리 T와 $k = 3$이 입력으로 주어지면 알고리즘은 정수 8을 반환해야 합니다.

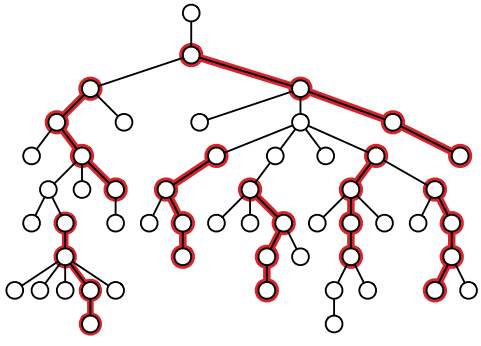


그림 3.11. 길이가 $k = 3$인 서로소 경로 일곱 개입니다. 이것은 이 트리에서 그러한 경로 집합 중 가장 큰 것이 *아닙니다*.

51. T를 정점이 n개인 루트가 있는 이진 트리라고 하고, $k \leq n$을 양의 정수라고 합시다. 모든 정점이 가까운 곳에 표시된 조상을 갖도록 T에서 정점 k개를 표시하고자 합니다. 더 형식적으로, 정점들의 임의의 부분집합 K의 *군집화 비용*을 다음과 같이 정의합니다

$$cost(K) = \max_{v} cost(v, K),$$

여기서 최댓값은 트리의 모든 정점 v에 대해 취하며, $cost(v, K)$는 v에서 K에 속한 가장 가까운 조상까지의 거리입니다:

$$cost(v, K) = \begin{cases} 0 & \text{만약} v \in K \\ \infty & v\text{가 } T\text{의 루트이고} v \notin K \\ 1 + cost(parent(v)) & \text{그렇지 않으면} \end{cases}$$

특히 K가 T의 루트를 포함하지 않으면 $cost(K) = \infty$입니다.

[25] Q: 컴퓨터 과학 교수들은 왜 나무의 뿌리가 꼭대기에 있다고 생각합니까? A: 밖에 나가 본 적이 없기 때문입니다!

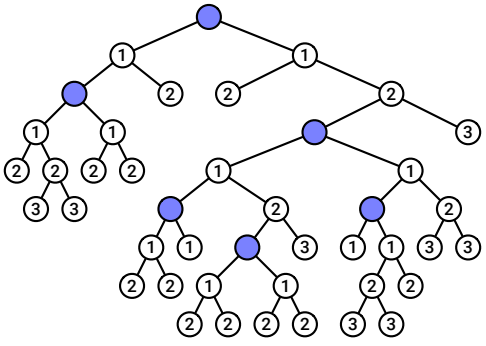


그림 3.12. 이진 트리에서 정점 다섯 개로 이루어진 부분집합이며, 이 부분집합의 클러스터링 비용은 3입니다.

♥(a) 트리 T와 정수 k가 주어졌을 때, T의 정점 k개로 이루어진 부분집합의 군집화 비용 중 최솟값을 계산하는 동적 계획법 알고리즘을 설명하십시오. 만점을 받으려면 알고리즘이 $O(n^2k^2)$ 시간에 실행되어야 합니다.

(b) 트리 T와 정수 r이 주어졌을 때, 클러스터링 비용이 r 이하인 정점 부분집합 중 가장 작은 것의 크기를 계산하는 동적 계획법 알고리즘을 설명하십시오. 만점을 받으려면 알고리즘이 $O(nr)$ 시간에 실행되어야 합니다.

(c) (b)에 대한 여러분의 풀이로부터 (a)를 $O(n^2 \log n)$ 시간에 해결하는 알고리즘을 얻을 수 있음을 보이십시오.

52. 이 문제에서는 주어진 두 루트 트리의 ***최대 공통 루트 부분 트리***를 계산하는 효율적인 알고리즘을 찾아야 합니다. *루트 트리*란 루트라고 부르는 노드 하나가 지정된 연결 비순환 그래프라는 점을 떠올리십시오. 루트 트리의 루트 부분 트리는 임의의 노드 하나와 그 노드의 모든 자손으로 이루어집니다. "공통"의 정확한 정의는 어떤 루트 트리 쌍을 동형으로 간주하느냐에 따라 달라집니다.

(a) *이진* 트리는 모든 노드가 (비어 있을 수도 있는) *왼쪽* 부분 트리와 (비어 있을 수도 있는) 오른쪽 부분 트리를 가지는 루트 트리임을 상기하십시오. 두 이진 트리는 둘 다 비어 있거나, 왼쪽 부분 트리끼리 동형이고 오른쪽 부분 트리끼리 동형일 때, 그리고 그때에만 동형입니다. 주어진 두 *이진* 트리의 가장 큰 공통 *이진* 부분 트리를 찾는 알고리즘을 설명하십시오.

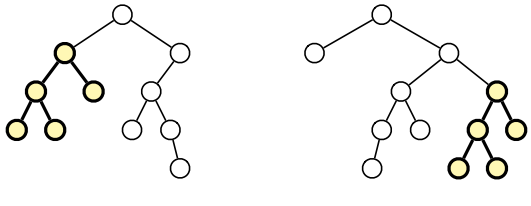


그림 3.13. 가장 큰 공통 (루트가 있는) 부분 트리를 강조하여 표시한 두 이진 트리입니다.

(b) *순서 있는* 루트 트리에서 각 노드는 자식들의 *수열*을 가지며, 각 자식은 순서 있는 루트 부분 트리의 루트입니다. 두 순서 있는 루트 트리는 둘 다 비어 있거나, 모든 인덱스 i에 대해 두 트리의 i번째 부분 트리가 서로 동형이면 동형입니다. 두 순서 있는 트리 T_1과 T_2의 가장 큰 공통 순서 부분 트리를 찾는 알고리즘을 설명하십시오.

♦♥(c) *순서 없는* 루트 트리에서 각 노드는 순서가 없는 자식들의 *집합*을 가지며, 각 자식은 순서 없는 루트 부분 트리의 루트입니다. 두 순서 없는 루트 트리가 모두 비어 있거나, 각 루트의 부분 트리들을 *적절히 순서를 매겨* 모든 인덱스 i에 대해 i번째 부분 트리들이 서로 동형이 되게 할 수 있으면, 두 트리는 동형입니다. 두 순서 없는 트리 T_1과 T_2의 가장 큰 공통 순서 없는 부분 트리를 찾는 알고리즘을 설명하십시오.

53. 이 문제는 *루트 없는* 트리, 즉 연결된 비순환 무방향 그래프에서 최적 부분 트리를 계산하는 효율적인 알고리즘을 찾는 문제입니다. 루트 없는 트리의 *부분 트리*는 연결된 임의의 부분 그래프입니다.

(a) 루트 없는 트리 T가 주어지고, 그 *간선*에는 양수, 음수 또는 0일 수 있는 가중치가 붙어 있다고 가정합시다. T에서 총 가중치가 최대인 *경로*를 찾는 알고리즘을 설명하십시오.

(b) *정점*에 가중치가 있는 루트 없는 트리 T가 주어졌다고 가정합시다. 각 가중치는 양수, 음수, 0 중 어느 것이든 될 수 있습니다. 총 가중치가 최대인 T의 *부분 트리*를 찾는 알고리즘을 설명하십시오. *[이 문제는 2016년 Google 면접 문제였습니다.]*

(c) T_1과 T_2가 임의의 *순서 있는* 루트 없는 트리, 즉 모든 노드의 이웃들이 잘 정의된 순환 순서를 갖는 트리라고 합시다. T_1과 T_2의 가장 큰 공통 *순서 있는* 부분 트리를 찾는 알고리즘을 설명하십시오.

♦♥(d) T_1과 T_2가 임의의 *순서 없는* 루트 없는 트리라고 합시다. T_1과 T_2의 가장 큰 공통 *순서 없는* 부분 트리를 찾는 알고리즘을 설명하십시오.

54. 루트 트리의 ***루트 마이너(rooted minor)***는 부분 수열을 자연스럽게 일반화한 개념입니다. 루트 트리 T의 루트 마이너란 하나 이상의 간선을 *축약*하여 얻을 수 있는 모든 트리를 말합니다. u가 v의 부모일 때 간선 $u \to v$를 축약하면 v의 자식들이 u의 새로운 자식이 되고, 그다음 v가 삭제됩니다. 특히 T의 루트는 T의 모든 루트 마이너의 루트이기도 합니다.

(a) T를 노드에 레이블이 붙은 루트 트리라고 합시다. 각 노드 x에 대해 x의 모든 자식이 같은 레이블을 가지면 T를 *지루하다*고 합니다. 서로 다른 노드의 자식들은 서로 다른 레이블을 가질 수 있습니다. 레이블이 붙은 루트 트리가 주어졌을 때, 가장 큰 지루한 루트 마이너를 찾는 알고리즘을 설명하십시오.

(b) 노드에 숫자 레이블이 붙은 루트 트리 T가 주어졌다고 가정합시다. T의 가장 큰 *힙 순서 루트 마이너*를 찾는 알고리즘을 설명하십시오. 즉, 여러분의

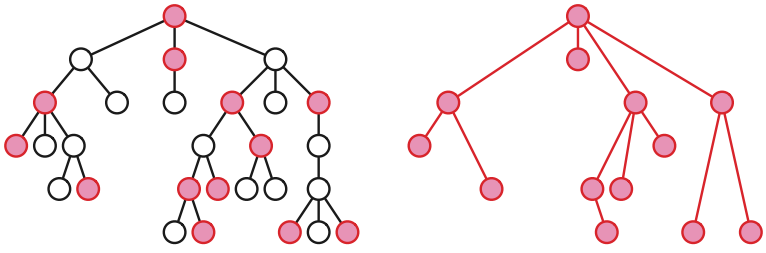


그림 3.14. 루트 트리와 그 루트 마이너 중 하나입니다.

알고리즘은 M의 모든 노드가 M에서 자신의 자식들보다 작은 레이블을 갖는 가장 큰 루트 마이너 M을 반환해야 합니다.

(c) 노드에 수가 레이블로 붙어 있는 *이진* 트리 T가 주어졌다고 가정합시다. T의 가장 큰 *이진 탐색 순서 루트 마이너*를 찾는 알고리즘을 설명하십시오. 즉, 여러분의 알고리즘은 M의 모든 노드가 최대 두 개의 자식을 가지고 M의 중위 순회가 T의 중위 순회의 증가 부분 수열이 되는 루트 마이너 M을 반환해야 합니다.

(d) 루트 트리에서 각 노드의 자식들 사이에 왼쪽에서 오른쪽으로 잘 정의된 순서가 있으면, 그 트리를 *순서* 트리라고 한다는 것을 떠올리십시오. 노드에 수가 레이블로 붙어 있는 *임의의* 순서 트리 T가 주어질 때, 이진 탐색 순서를 따르는 가장 큰 마이너를 찾는 알고리즘을 설명하십시오. 이번에도 M의 노드들이 왼쪽에서 오른쪽으로 놓이는 순서는 T에서의 순서와 일치해야 합니다.

♥(e) 레이블이 붙은 두 *순서* 루트 트리의 가장 큰 공통 *순서* 루트 마이너를 찾는 알고리즘을 설명하십시오.

♦♥(f) *순서 없는* 레이블 루트 트리 두 개의 가장 큰 공통 *순서 없는* 루트 마이너를 찾는 알고리즘을 설명하십시오. *[힌트: 동적 계획법과 최대 흐름을 결합하십시오.]*

요점은 말입니다, 신사 숙녀 여러분, 탐욕은 좋은 것이라는 겁니다. 탐욕은 효과가 있고, 탐욕은 옳습니다. 탐욕은 사물을 명료하게 하고, 핵심을 꿰뚫으며, 진화 정신의 본질을 포착합니다. 모든 형태의 탐욕, 즉 삶과 돈과 사랑과 지식에 대한 탐욕이 인류의 비약적인 발전을 이끌어 왔습니다. 그리고 탐욕은—제 말을 명심하십시오—텔다 제지뿐 아니라 미국이라는 이름의, 제대로 돌아가지 않는 또 하나의 기업도 구해 낼 것입니다.

— 고든 게코 [Michael 더글러스], *월 스트리트* (1987)

인간의 모든 문제에는 언제나 쉬운 해결책이 있습니다—깔끔하고, 그럴듯하며, 틀린 해결책입니다.

— H. L. 멩켄, "신성한 영감(The Divine Afflatus)",
New York Evening Mail (1917년 11월 16일)

4

탐욕 알고리즘

4.1 테이프에 파일 저장하기

자기 테이프에 저장하려는 n개의 파일로 이루어진 집합이 있다고 가정해 봅시다.[1] 앞으로 사용자들은 테이프에서 이 파일들을 읽으려 할 것입니다. 테이프에서 파일을 읽는 것은 디스크에서 파일을 읽는 것과 다릅니다. 먼저 다른 파일들을 모두 빨리 감기로 지나가야 하는데, 여기에는 상당한 시간이 걸립니다. $L[1..n]$을 각 파일의 길이를 나열한 배열이라고 합시다. 구체적으로 말하면, 파일 i의 길이는 $L[i]$입니다. 파일들이 1부터 n까지 순서대로 저장되어 있다면, k번째 파일에 접근하는 비용은 다음과 같습니다.

$$cost(k) = \sum_{i=1}^{k} L[i].$$

이 비용은 파일 k를 읽기 전에 먼저 테이프에서 그보다 앞에 있는 모든 파일을 지나가며 훑어야 한다는 사실을 반영합니다. 잠시 각 파일에 접근할 확률이 모두 같다고 가정하면, 임의의 파일을 찾는 데 드는 *기대* 비용은 다음과 같습니다.

$$\mathrm{E}[cost] = \sum_{k=1}^{n} \frac{cost(k)}{n} = \frac{1}{n} \sum_{k=1}^{n} \sum_{i=1}^{k} L[i].$$

[1] 자기 테이프는 이미 수십 년 전에 구식이 되었다고 반박하고 싶은 독자께서는 가까운 슈퍼컴퓨터 시설을 견학해 보시기를 정중히 권해 드립니다. 가서 테이프 로봇을 보여 달라고 요청해 보십시오. 아니면 도서관 서가에 책을 차례로 꽂는 일을 생각해 보십시오. 아시지 않습니까, 죽은 나무와 잉크로 만든 벽돌처럼 생긴 그 이상한 물건 말입니다.

테이프에 저장된 파일의 순서를 바꾸면 파일에 접근하는 비용도 달라집니다. 어떤 파일은 읽는 비용이 더 커지지만, 다른 파일은 더 작아집니다. 파일 순서가 다르면 기대 비용도 달라지기 마련입니다. 구체적으로, 테이프의 위치 i에 저장된 파일의 인덱스를 $\pi(i)$로 나타냅시다. 그러면 순열 π의 기대 비용은 다음과 같습니다.

$$\mathrm{E}[cost(\pi)] = \frac{1}{n}\sum_{k=1}^{n}\sum_{i=1}^{k} L[\pi(i)].$$

이 기대 비용을 가능한 한 작게 만들려면 어떤 순서를 사용해야 하겠습니까? 답은 직관적으로 명백해 *보입니다*. 파일을 길이가 증가하는 순서로 정렬하면 됩니다. 하지만 직관은 다루기 까다로운 짐승입니다. 이 순서가 올바르게 작동한다고 *확신*할 수 있는 유일한 방법은 이륙해서 궤도에서 기지 전체를 핵으로 날려 버리는 것 실제로 이 순서가 작동함을 *증명*하는 것입니다!

보조정리 4.1. *모든 i에 대해 $L[\pi(i)] \le L[\pi(i+1)]$일 때 $\mathrm{E}[cost(\pi)]$가 최소가 됩니다.*

증명: 어떤 인덱스 i에 대해 $L[\pi(i)] > L[\pi(i+1)]$이라고 가정합시다. 표기를 간단히 하기 위해 $a = \pi(i)$, $b = \pi(i+1)$이라고 둡시다. 파일 a와 b를 서로 바꾸면 a에 접근하는 비용은 $L[b]$만큼 증가하고, b에 접근하는 비용은 $L[a]$만큼 감소합니다. 전체적으로 이 교환은 기대 비용을 $(L[b] - L[a])/n$만큼 변화시킵니다. 그런데 $L[b] < L[a]$이므로 이 변화는 개선입니다. 따라서 파일들의 순서가 어긋나 있다면, 순서가 어긋난 파일 쌍 하나를 서로 바꾸어 기대 비용을 줄일 수 있습니다. □

이것은 올바른 *탐욕 알고리즘*의 첫 번째 예입니다. 파일에 접근하는 *총* 기대 비용을 최소화하기 위해, 접근 비용이 가장 낮은 파일을 맨 앞에 두고 나머지 파일은 모두 재귀적으로 기록합니다. 백트래킹도, 동적 계획법도 없이 그저 국소적으로 가장 좋은 선택을 하고 무작정 밀고 나갈 뿐입니다. 효율적인 정렬 알고리즘을 사용하면 실행 시간은 분명히 $O(n \log n)$이며, 여기에 실제로 파일을 기록하는 데 필요한 시간이 더해집니다. 탐욕 알고리즘이 실제로 올바르다는 것을 보이기 위해, 우리는 다른 어떤 알고리즘의 출력이든 일종의 교환을 통해 개선할 수 있음을 증명했습니다

이 아이디어를 더 일반화해 봅시다. 각 파일의 *접근 빈도*를 담은 배열 $F[1..n]$도 주어진다고 가정합시다. 파일 i는 테이프를 사용하는 전체 기간 동안 정확히 $F[i]$번 접근됩니다. 이제 테이프에 있는 모든 파일에 접근하는 *총* 비용은 다음과 같습니다.

$$\Sigma cost(\pi) = \sum_{k=1}^{n}\left(F[\pi(k)] \cdot \sum_{i=1}^{k} L[\pi(i)]\right) = \sum_{k=1}^{n}\sum_{i=1}^{k}\big(F[\pi(k)] \cdot L[\pi(i)]\big).$$

앞에서와 마찬가지로, 파일의 순서를 바꾸면 이 총비용이 달라질 수 있습니다. 그렇다면 총비용을 가능한 한 작게 하려면 어떤 순서를 사용해야 합니까? (이 질문은 취지면에서 최적 이진 탐색 트리 문제와 비슷하지만, 대상 자료구조와 비용 함수가 모두 다르므로 알고리즘도 달라야 합니다.)

모든 빈도가 같다면 파일을 크기가 증가하는 순서로 정렬해야 한다는 것은 이미 증명했습니다. 빈도는 모두 다르지만 파일 길이 $L[i]$가 모두 같다면, 직관적으로 파일을 접근 빈도가 *감소하는* 순서로, 즉 가장 많이 접근되는 파일이 맨 앞에 오도록 정렬해야 합니다. 실제로 이는 보조정리 4.1의 증명을 수정하면 어렵지 않게 증명할 수

있습니다(힌트, 힌트). 그렇다면 크기와 빈도가 모두 제각각이면 어떻게 해야 합니까? 이 경우에는 파일을 *비율* L/F에 따라 정렬해야 합니다.

보조정리 4.2. *모든 i에 대해* $\dfrac{L[\pi(i)]}{F[\pi(i)]} \le \dfrac{L[\pi(i+1)]}{F[\pi(i+1)]}$ *일 때* $\Sigma cost(\pi)$*가 최소가 됩니다.*

증명: 어떤 인덱스 i에 대해 $L[\pi(i)]/F[\pi(i)] > L[\pi(i+1)]/F[\pi(i+i)]$라고 가정합시다. 표기를 간단히 하기 위해 $a = \pi(i)$, $b = \pi(i+1)$이라고 합시다. 파일 a와 b를 서로 맞바꾸면 a에 접근하는 비용은 $L[b]$만큼 증가하고, b에 접근하는 비용은 $L[a]$만큼 감소합니다. 전체적으로 이 교환으로 총비용은 $L[b]F[a] - L[a]F[b]$만큼 달라집니다. 그런데 이 변화는 개선입니다. 왜냐하면

$$\frac{L[a]}{F[a]} > \frac{L[b]}{F[b]} \iff L[b]F[a] - L[a]F[b] < 0.$$

따라서 인접한 두 파일의 순서가 잘못되어 있다면, 두 파일을 서로 맞바꾸어 총비용을 줄일 수 있습니다. □

4.2 강의 일정 짜기

다음 예제는 조금 더 복잡합니다. 여러분이 컴퓨터 과학을 그만두고 전공을 응용 혼돈학(Applied Chaos)으로 바꾸기로 했다고 가정해 봅시다. 응용 혼돈학과는 모든 수업을 매주 같은 요일에 개설하는데, 학생들은 이 요일을 "맨정신의 날(Soberday)"이라고 부릅니다(그런데 흥미롭게도 교수진은 그렇게 부르지 *않습니다*). 모든 수업은 시작 시각과 종료 시각이 저마다 다릅니다. 예를 들어 AC 101("화장지 조경 건축학")은 오후 10시 27분에 시작하여 오후 11시 51분에 끝나고, AC 666("종말의 내재화")은 오후 4시 18분에 시작하여 오후 4시 22분에 끝나는 식입니다. 여러분은 가능한 한 빨리 졸업하기 위해 가능한 한 많은 수업을 수강 신청하려고 합니다. (응용 혼돈학 수업에서는 실제로 *공부*를 할 필요가 전혀 없습니다.) 대학의 수강 신청 컴퓨터는 시간이 겹치는 수업을 신청하지 못하게 하며, 학과의 누구도 이 "기능"을 우회하는 방법을 모릅니다. 어떤 수업들을 들어야 하겠습니까?

좀 더 엄밀하게 말하면, 각 수업의 시작 시각과 종료 시각을 나열한 두 배열 $S[1..n]$과 $F[1..n]$이 주어진다고 가정합시다. 구체적으로, 어떤 값 M(예를 들어 소버데이의 피코초 수)에 대해 각 i마다 $0 \le S[i] < F[i] \le M$이라고 가정할 수 있습니다. 여러분이 할 일은 임의의 쌍 $i, j \in X$에 대해 $S[i] > F[j]$ 또는 $S[j] > F[i]$가 성립하도록 가능한 한 가장 큰 부분집합 $X \in \{1, 2, \ldots, n\}$을 고르는 것입니다. 각 수업을 왼쪽과 오른쪽 x좌표가 각각 시작 시각과 종료 시각을 나타내는 직사각형으로 그리면 이 문제를 그림으로 나타낼 수 있습니다. 목표는 수직 방향으로 서로 겹치지 않는 직사각형들로 이루어진 가장 큰 부분집합을 찾는 것입니다.

이 문제에는 수업 1을 듣거나 듣지 않거나 둘 중 하나라는 관찰에 기반한, 꽤 간단한 재귀적 해법이 있습니다. 수업 1이 시작하기 *전에 끝나는* 수업들의 집합을 B라 하고,

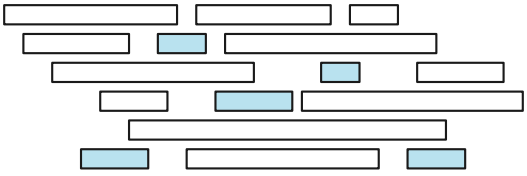


그림 4.1. 수업 집합에 대한 충돌 없는 최대 시간표입니다.

수업 1이 끝난 *후에 시작하는* 수업들의 집합을 A라 합시다:

$$B := \{i \mid 2 \le i \le n \text{그리고} F[i] < S[1]\}$$
$$A := \{i \mid 2 \le i \le n \text{그리고} S[i] > F[1]\}$$

수업 1이 최적 일정에 포함된다면, B와 A에 대한 최적 일정도 함께 포함되며, 이들은 재귀적으로 찾을 수 있습니다. 그렇지 않다면 $\{2, 3, \ldots, n\}$에 대한 최적 일정을 재귀적으로 찾을 수 있습니다. 따라서 두 선택지를 모두 시도해 보고 더 나은 일정을 주는 쪽을 택해야 합니다. 이 재귀 알고리즘을 상향식으로 계산하면 $O(n^3)$ 시간에 실행되는 동적 계획법 알고리즘을 얻습니다. 더 잘할 수 있으므로 굳이 세부 사항까지 살펴보지는 않겠습니다.[2]

직관적으로 생각하면, 첫 번째 수업이 가능한 한 일찍 끝나기를 바랍니다. 그래야 남는 수업의 수가 가장 많아지기 때문입니다. 이 직관에서 다음과 같은 간단한 탐욕 알고리즘이 나옵니다. 수업을 끝나는 시각 순서대로 살펴보면서, 지금까지 선택한 마지막 수업과 겹치지 않는 수업을 만날 때마다 그 수업을 선택하십시오! 이렇게 얻은 탐욕 일정을 시각화한 그림은 그림 4.2를 참고하십시오.

탐욕 알고리즘은 그림 4.3에 보인 것처럼 좀 더 형식적으로 작성할 수 있습니다. (첫 줄은 이해하실 수 있으리라 생각합니다.) 처음에 정렬하고 나면 알고리즘은 단순한 선형 시간 반복문이므로, 전체 알고리즘은 $\boldsymbol{O(n \log n)}$ *시간*에 실행됩니다.

GreedySchedule가 실제로 충돌 없는 가장 큰 일정을 계산한다는 것을 증명하기 위해, 테이프 정렬에서 사용했던 것과 비슷한 교환 논증을 사용합니다. 탐욕 일정이 *유일한* 최대 일정이라고 주장하는 것은 아닙니다. 다른 일정도 있을 수 있습니다. (그림 4.1과 그림 4.2를 비교해 보십시오!) 우리가 주장할 수 있는 것은 최적 일정 중 적어도 하나가 탐욕 알고리즘이 만들어 낸 일정이라는 것뿐입니다.

보조정리 4.3. *극대 무충돌 시간표 중 적어도 하나는 가장 먼저 끝나는 수업을 포함합니다.*

증명: 가장 먼저 끝나는 수업을 f라고 합시다. f를 포함하지 않는 최대 크기의 충돌 없는 시간표 X가 있다고 가정합시다. X에서 가장 먼저 끝나는 수업을 g라고 합시다. f는 g보다 먼저 끝나므로, f는 집합 $X \setminus \{g\}$에 속한 어떤 수업과도 충돌할 수 없습니다. 따라서 시간표 $X' = X \cup \{f\} \setminus \{g\}$ 역시 충돌이 없습니다. X'는 X와 크기가 같으므로, 이 역시 최대 크기입니다. □

[2] 그래도 세부 사항은 여러분이 직접 풀어 보아야 합니다. 동적 계획법 알고리즘은 "최선"을 여러 가지로 정의하더라도 각각에 대해 "최선"의 일정을 찾는 데 사용할 수 있지만, 여기서 설명하는 탐욕 알고리즘은 "최선"이 "가장 큰 것"을 뜻할 때만 작동합니다. 또한 다른 점화식을 사용하면 실행 시간을 $O(n^2)$으로 개선할 수 있습니다.

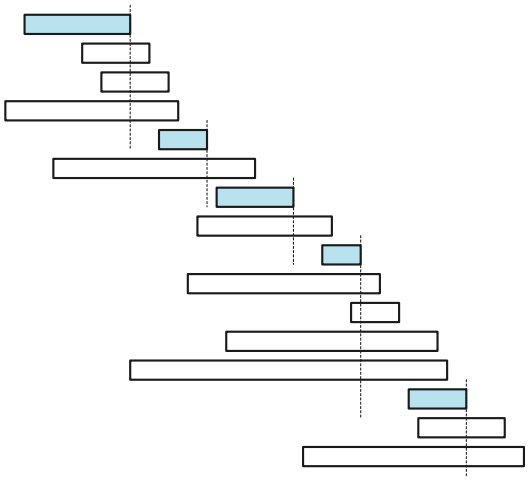


그림 4.2. 종료 시각순으로 정렬한 같은 강의들과 탐욕 일정입니다.

```
GreedySchedule(S[1..n], F[1..n]):
  sort F and permute S to match
  count ← 1
  X[count] ← 1
  for i ← 2 to n
      if S[i] > F[X[count]]
          count ← count + 1
          X[count] ← i
  return X[1..count]
```

그림 4.3. 서로 겹치지 않는 수업들의 최대 집합을 찾는 탐욕 알고리즘

증명을 마무리하기 위해, 우리의 오랜 친구인 귀납법을 불러옵시다.

정리 4.4. *탐욕 일정은 최적 일정입니다.*

증명: f를 가장 먼저 끝나는 수업이라 하고, A를 f가 끝난 뒤에 시작하는 수업들의 부분집합이라 합시다. 앞의 보조정리에 따르면 어떤 최적 시간표는 f를 포함하므로, f를 포함하는 시간표 중 가장 좋은 시간표는 최적 시간표입니다. f를 포함하는 가장 좋은 시간표는 f와 충돌하지 않는 수업들에 대한 최적 시간표, 즉 A에 대한 최적 시간표를 포함해야 합니다. 탐욕 알고리즘은 f를 선택한 다음, 귀납 가정에 의해 A의 수업들로 이루어진 최적 시간표를 계산합니다. □

귀납법을 조금 풀어서 전개하면 증명을 더 쉽게 이해할 수 있을 것입니다.

증명: $\langle g_1, g_2, \ldots, g_k \rangle$를 탐욕 알고리즘이 선택한 수업들을 시작 시각 순으로 정렬한 수열이라고 합시다. 충돌 없는 최대 시간표가 있다고 가정합시다

$$S = \langle g_1, g_2, \ldots, g_{j-1}, \mathbf{c_j}, c_{j+1}, \ldots, c_m \rangle,$$

마찬가지로 시작 시간 순으로 정렬되어 있으며, 여기서 c_j는 탐욕 알고리즘이 선택한 수업 g_j와 다릅니다. ($j = 1$일 수도 있으며, 이 경우 이 일정은 탐욕적이지 않은 선택 c_1

으로 시작합니다.) 구성 방식에 따라 j번째 탐욕적 선택 g_j는 앞선 수업 $g_1, g_2, \ldots, g_{j-1}$ 중 어느 것과도 충돌하지 않으며, 우리의 일정 S에는 충돌이 없으므로 c_j 역시 충돌하지 않습니다. 더욱이 g_j는 앞선 수업들과 충돌하지 않는 모든 수업 중에서 *가장 이른* 종료 시간을 가지며, 특히 g_j는 c_j보다 먼저 끝납니다. 따라서 g_j는 이후의 수업 $c_{j+1}, \ldots, c_m$ 중 어느 것과도 충돌하지 않습니다. 그러므로 수정된 일정

$$S' = \langle g_1, g_2, \ldots, g_{j-1}, \boldsymbol{g_j}, c_{j+1}, \ldots, c_m \rangle,$$

역시 충돌이 없습니다. (이 논증은 $j = 1$인 경우를 다루는 보조정리 4.3을 그대로 일반화한 것입니다.)

이제 귀납법에 의해, 탐욕 알고리즘이 선택한 모든 수업을 포함하는 최적 일정 $\langle g_1, g_2, \ldots, g_k, c_{k+1}, \ldots, c_m \rangle$이 존재함이 따라 나옵니다. 그러나 이는 $k = m$이 아니면 불가능합니다. 어떤 수업 c_{k+1}이 처음 k개의 탐욕 수업 중 어느 것과도 충돌하지 않는다면, 탐욕 알고리즘은 k개보다 많은 수업을 선택했을 것이기 때문입니다! □

4.3 일반적인 패턴

이 정확성 증명의 기본 구조는 테이프 정렬 문제의 경우와 정확히 같습니다. 바로 귀납적 교환 논증입니다.

- 탐욕 알고리즘으로 구한 해와 다른 최적해가 존재한다고 가정합시다.
- 두 해의 "첫 번째" 차이점을 찾으십시오.
- 해를 더 나쁘게 만들지 않고도 최적의 선택을 탐욕적 선택으로 바꿀 수 있음을 논증하십시오(다만 이 교환으로 해가 더 좋아지지는 않을 수도 있습니다).

이 논증에 귀납법을 적용하면 어떤 최적해가 탐욕 해 전체를 *포함*하고, 따라서 탐욕 해와 *같다*는 결론이 나옵니다. 스케줄링 문제에서처럼 어떤 최적해도 탐욕 해를 *엄밀하게* 개선하지 못한다는 것을 보이려면 추가 단계가 필요한 경우도 있습니다.

4.4 허프만 코드

*이진 코드*는 알파벳의 각 문자에 0과 1로 이루어진 문자열을 할당합니다. 어떤 코드도 다른 코드의 접두사가 아니면 그 이진 코드를 *접두사 없는(prefix-free)* 코드라고 합니다. (혼란스럽게도, 접두사 없는 코드는 흔히 *접두사 코드(prefix code)*라고도 불립니다.) 7비트 ASCII와 유니코드의 UTF-8은 모두 접두사 없는 이진 코드입니다. 모스 부호는 • 과 —를 기호로 사용하는 이진 코드이지만, 접두사 없는 코드는 *아닙니다*. E의 코드(•)가 I(••), S(•••), H(••••)의 코드의 접두사이기 때문입니다.[3]

[3]이런 이유로 모스 부호는 •, —, 그리고 쉼(pause)이라는 세 기호를 사용하는 접두사 없는 *삼진* 코드로 설명하는 편이 더 낫다고 볼 수도 있습니다. 또는 모스 부호를 소리/빛/전류/고전압/연기/기체의 한 박자(■)와 침묵/어둠/접지/저전압/공기/액체의 한 박자(□)를 두 기호로 삼는 접두사 없는 이진 코드로 볼 수도 있습니다. 그러면 각 "점(dit)"은 ■□로, 각 "선(dah)"은 ■■■□로, 각 쉼은 □□로 부호화됩니다. 표준 모스 부호에서는 각 글자 뒤에 쉼이 하나 붙고, 각 단어 뒤에는 쉼이 두 개 더 붙습니다. 단, 부호화된 전체 메시지 끝에 오는 □들은 생략합니다. 예를 들어, 문자열 "MORSE CODE"는 다음 비트 문자열로 모호함

모든 접두사 없는 이진 코드는 인코딩된 문자들이 잎에 저장된 이진 트리로 시각화할 수 있습니다. 각 기호의 코드 단어는 루트에서 해당 잎까지의 경로로 주어지며, 왼쪽은 `0`, 오른쪽은 `1`입니다. 따라서 각 기호의 코드 단어 길이는 코드 트리에서 해당 잎의 깊이와 같습니다. 겉보기에는 비슷하지만, 이진 코드 트리는 이진 탐색 트리가 *아닙니다*. 잎에 놓인 기호들의 순서는 전혀 신경 쓰지 않습니다.

인코딩된 메시지가 가능한 한 짧아지도록 n개의 문자로 이루어진 알파벳으로 쓰인 메시지를 인코딩하고 싶다고 가정합시다. 구체적으로, 빈도수 배열 $f[1..n]$이 주어졌을 때, 메시지의 전체 인코딩 길이를 최소화하는 접두사 없는 이진 코드를 계산하고자 합니다:

$$\sum_{i=1}^{n} f[i] \cdot depth(i).$$

이는 이진 탐색 트리를 최적화할 때 살펴보았던 것과 정확히 같은 비용 함수입니다. 하지만 코드 트리는 키를 특정한 순서로 유지할 필요가 없으므로 최적화 문제는 다릅니다.

1951년, MIT의 박사 과정 학생이던 David Huffman은 이러한 최적 코드를 만들어 내는 다음과 같은 탐욕 알고리즘을 개발했습니다:[4]

> HUFFMAN: 빈도가 가장 낮은 두 글자를 병합한 뒤 재귀 호출합니다.

허프만 알고리즘은 예제를 통해 설명하는 것이 가장 좋습니다. Lee Sallows가 발견한, 친절하게도 자기 자신을 기술하는 다음 문장을 부호화하려 한다고 가정합시다:[5]

> 이 문장에는 a가 세 개, c가 세 개, d가 두 개, e가 스물여섯 개, f가 다섯 개, g가 세 개, h가 여덟 개, i가 열세 개, l이 두 개, n이 열여섯 개, o가 아홉 개, r이 여섯 개, s가 스물일곱 개, t가 스물두 개, u가 두 개, v가 다섯 개, w가 여덟 개, x가 네 개, y가 다섯 개, 그리고 z가 단 하나 들어 있습니다.

없이 부호화됩니다.

.

[4]Huffman은 Robert Fano가 가르치는 정보 이론 수업의 학생이었는데, Fano는 정보 이론의 아버지인 Claude Shannon의 가까운 동료였습니다. Fano와 Shannon은 이전에 접두사 코드를 만드는 다른 탐욕 알고리즘을 개발한 적이 있었습니다. 빈도 배열을 가능한 한 균등하게 두 개의 부분 배열로 나눈 다음, 각 부분 배열에 대한 코드를 재귀적으로 구성하는 방식이었습니다. 하지만 이 Fano-Shannon 코드는 최적이 아니라는 사실이 알려져 있었습니다. Fano는 자신의 수업에서 최적 접두사 코드를 찾는 문제를 제시했습니다. Huffman은 기말 시험을 보는 대신 수업 프로젝트로 이 문제를 풀기로 했는데, 이 문제가 미해결 문제라는 것도, Fano와 Shannon이 이미 풀려고 시도했다가 실패했다는 것도 알지 못했습니다. 몇 달 동안 아무 성과 없이 애쓴 끝에 Huffman은 결국 포기하고 그냥 기말 시험을 보기로 했습니다. 그런데 노트를 쓰레기통에 던져 넣던 바로 그때 해법이 떠올랐습니다. Huffman은 훗날 이 깨달음을 "갑작스러운 깨달음의 완벽한 번개"라고 묘사했습니다.

[5]이 문장은 Alexander Dewdney가 *Scientific American* 1984년 10월호의 "Computer Recreations" 칼럼에서 처음 소개했습니다. Sallows 본인은 1985년에 이 문장을 발견하기까지의 놀라운 이야기를 다른 여러 자기 기술 문장과 함께 발표했습니다. Sallows의 논문은 그의 웹사이트에서 찾을 수 있습니다. VAX 11/780에서 실행되는 자신의 코드가 좀처럼 진척이 없자 답답해진 Sallows는 다양한 도입 문구("This pangram has…", "This sentence contains exactly…" 등)를 가진 자기 기술 문장을 무차별 대입으로 탐색하는 전용 하드웨어를 직접 설계하고 제작했습니다. 신중한 이론적 분석을 통해 탐색 공간을 60억 개를 조금 넘는 경우로 줄였고, 그의 1MHz Pangram Machine은 이를 두 시간이 채 걸리지 않아 모두 열거했습니다.

문제를 단순하게 하기 위해 공백 44개, 아포스트로피 19개, 쉼표 19개, 하이픈 3개, 그리고 단 하나뿐인 마침표는 무시하고, 마치 메시지가 *scriptio continua*로 쓰인 것처럼 글자만 인코딩합시다:

```
THISSENTENCECONTAINSTHREEASTHREECSTWODSTWENTYSIXESFIVEFST
HREEGSEIGHTHSTHIRTEENISTWOLSSIXTEENNSNINEOSSIXRSTWENTYSEV
ENSSTWENTYTWOTSTWOUSFIVEVSEIGHTWSFOURXSFIVEYSANDONLYONEZ
```
[6]

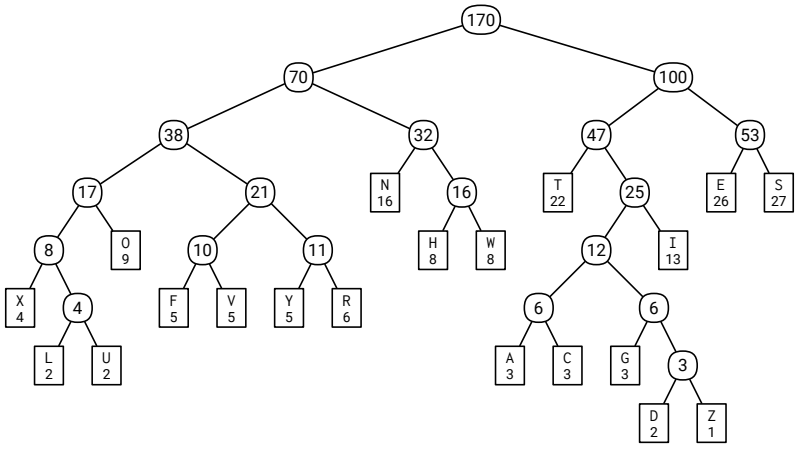
다음은 Sallows의 문장에 대한 빈도표입니다:

A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

허프만 알고리즘은 빈도가 가장 낮은 두 글자를 골라(동점이면 임의로 고릅니다)—이 경우에는 예를 들어 Z와 D라고 합시다—이 둘을 빈도가 3인 하나의 새 문자 Ø로 합칩니다. 이 새 문자는 우리가 구성하고 있는 코드 트리의 내부 노드가 되며, Z와 D를 자식으로 가집니다. 어느 자식이 어느 쪽에 오는지는 상관없습니다. 그런 다음 알고리즘은 다음과 같은 새 빈도표에 대한 허프만 코드를 재귀적으로 구성합니다

A	C	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Ø
3	3	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	3

19번의 병합을 거치면 20개의 글자가 모두 하나로 병합됩니다. 병합 기록이 곧 코드 트리가 됩니다. 이 알고리즘은 여러 차례 임의로 선택을 하므로, 실제로는 서로 다른 허프만 코드가 여러 개 존재합니다. 그러한 허프만 코드 중 하나를 아래에 보였습니다. 리프가 아닌 노드에 적힌 수는 병합된 문자들의 빈도입니다. 예를 들어 A의 코드는 101000이고, S의 코드는 111입니다.



이 특정한 허프만 코드로 Sallows의 문장을 부호화하면 다음과 같이 시작하는 비트 문자열이 만들어집니다:

100	0110	1011	111	111	110	010	100	110	010	101001	110	101001	0001	010	100	…
T	H	I	S	S	E	N	T	E	N	C	E	C	O	N	T	

[6] … 그리고 그 사람은 45분 동안 이야기했는데, 아무도 그가 한 말을 한 마디도 알아듣지 못했습니다. 하지만 우리는 거기 벤치에서 서류를 채우고 연필을 가지고 놀면서 즐거운 시간을 보냈습니다.

다음은 Sallows의 문장에서 각 문자를 인코딩하는 데 드는 비용과, 그 문자가 인코딩된 문장의 전체 길이에 기여하는 양을 함께 나열한 목록입니다:

char	A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
빈도	3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1
깊이	6	6	7	3	5	6	4	4	6	3	4	5	3	3	6	5	4	5	5	7
합계	18	18	14	78	25	18	32	52	12	48	36	30	81	66	12	25	32	20	25	7

종합하면, 인코딩된 메시지의 길이는 649비트입니다. 허프만 코드마다 같은 문자를 다르게 인코딩하며, 코드 단어의 길이가 서로 다를 수도 있습니다. 그러나 인코딩된 메시지의 전체 길이는 *모든* 허프만 코드에서 649비트로 같습니다.

허프만 알고리즘의 단순한 구조를 생각하면, 이 알고리즘이 *최적의* 접두사 없는 이진 코드를 만들어 낸다는 사실은 꽤 놀랍습니다.[7] *어떤* 접두사 없는 코드를 사용하더라도 Sallows의 문장을 인코딩하려면 최소 649비트가 필요합니다! 다행히 재귀적 구조 덕분에, 앞서 살펴본 최적성 증명들과 비슷하게 교환 논증을 사용하여 이 주장을 쉽게 증명할 수 있습니다. 먼저 이 알고리즘의 맨 처음 선택이 올바르다는 것부터 증명합시다.

보조정리 4.5. *x와 y를 빈도가 가장 낮은 두 문자라고 합시다(빈도가 같은 문자들 사이에서는 임의로 고릅니다). x와 y가 서로 형제인 최적 코드 트리가 존재합니다.*

증명: 사실 저는 더 강한 명제를 증명하겠습니다. 즉, x와 y가 형제 노드이면서 *동시에* 모든 리프 중 가장 큰 깊이를 갖는 최적 코드가 존재합니다.

T를 최적 코드 트리라고 하고, 이 트리의 깊이를 d라고 가정합시다. T는 정 이진 트리이므로 깊이 d에 서로 형제인 리프가 적어도 두 개 있습니다. (이 주장을 귀납법으로 확인해 보십시오!) 그 두 리프가 x와 y가 *아니라* 다른 문자 a와 b라고 가정합시다.

x와 a를 맞바꾸어 얻은 코드 트리를 T'이라 하고, $\Delta = d - depth_T(x)$라고 합시다. 이 교환으로 x의 깊이는 Δ만큼 증가하고 a의 깊이는 Δ만큼 감소하므로,

$$cost(T') = cost(T) + \Delta \cdot (f[x] - f[a]).$$

x는 빈도가 가장 낮은 두 문자 중 하나이지만 a는 그렇지 않다는 가정으로부터 $f[x] \le f[a]$가 성립하고, a의 깊이가 최대라는 가정으로부터 $\Delta \ge 0$이 성립합니다. 따라서 $cost(T') \le cost(T)$입니다. 한편 T는 최적 코드 트리이므로 $cost(T') \ge cost(T)$도 성립해야 합니다. 그러므로 T' *역시* 최적 코드 트리라는 결론을 얻습니다.

마찬가지로 y와 b를 맞바꾸어도 또 하나의 최적 코드 트리가 나와야 합니다. 이 최종 최적 코드 트리에서 x와 y는 요구한 대로 최대 깊이에 있는 형제입니다. □

이제 최적성은 우리의 친애하는 친구인 재귀 요정이 보장해 줍니다! 우리의 재귀적 논증은 다음과 같은 비표준적인 재귀적 정의에 기반합니다: *정 이진 트리는 단일 노드이거나, 어떤 리프가 두 개의 리프 자식을 가진 내부 노드로 대체된 정 이진 트리입니다.*

정리 4.6. *모든 허프만 코드는 최적의 접두사 없는 이진 코드입니다.*

증명: 메시지에 서로 다른 문자가 하나 또는 두 개뿐이라면 정리는 자명하므로, 그렇지 않다고 가정합니다.

$f[1..n]$을 원래의 입력 빈도라고 하고, 일반성을 잃지 않고 $f[1]$과 $f[2]$가 가장 작은 두 빈도라고 가정합시다. 재귀적 부분 문제를 구성하기 위해 $f[n+1] = f[1] + f[2]$

[7] 허프만과 파노 두 사람 모두에게도 분명히 놀라운 일이었습니다!

로 정의합니다. 앞서 살펴본 교환 논증에 따르면 1과 2는 $f[1..n]$에 대한 어떤 최적 코드에서 (가장 깊은) 형제입니다.

T'을 $f[3..n+1]$에 대한 허프먼 트리라고 합시다. 귀납 가정에 따라 T'은 더 작은 빈도 집합에 대한 최적 코드 트리입니다. 최종 코드 트리 T를 얻기 위해, $n+1$이라는 레이블이 붙은 리프를 1과 2라는 레이블이 붙은 두 자식을 가진 내부 노드로 교체합니다. 저는 T가 원래의 빈도 배열 $f[1..n]$에 대해 최적이라고 주장합니다.

이 주장을 증명하기 위해 T의 비용을 다음과 같이 T'의 비용으로 나타낼 수 있습니다. (이 식들에서 $depth(i)$는 T 또는 T'에서 i로 레이블된 리프의 깊이를 나타냅니다. T와 T'에 모두 나타나는 각 리프는 두 트리에서 같은 깊이를 가집니다.)

$$\begin{aligned}
cost(T) &= \sum_{i=1}^{n} f[i] \cdot depth(i) \\
&= \sum_{i=3}^{n+1} f[i] \cdot depth(i) \;+\; f[1] \cdot depth(1) + f[2] \cdot depth(2) \\
&\qquad\qquad - \; f[n+1] \cdot depth(n+1) \\
&= cost(T') + (f[1] + f[2]) \cdot depth(T) \;-\; f[n+1] \cdot (depth(T) - 1) \\
&= cost(T') + f[1] + f[2] \;+\; (f[1] + f[2] - f[n+1]) \cdot (depth(T) - 1) \\
&= cost(T') + f[1] + f[2]
\end{aligned}$$

이 등식은 T의 비용을 최소화하는 것이 T'의 비용을 최소화하는 것과 동치임을 뜻합니다. 특히 T'에서 $n+1$로 표시된 잎에 1과 2로 표시된 잎들을 붙이면 원래 빈도에 대한 최적 코드 트리를 얻습니다. □

허프만 코드를 효율적으로 구성하기 위해, 문자 빈도를 우선순위로 삼아 문자들을 우선순위 큐에 보관합니다. 코드 트리는 각 노드의 왼쪽(*L*eft) 자식과 오른쪽(*R*ight) 자식, 그리고 부모(*P*arent)를 나열하는 세 개의 인덱스 배열로 표현할 수 있습니다. 최종 코드 트리의 리프는 인덱스 1부터 n까지의 노드이며, 루트는 인덱스가 $2n-1$인 노드입니다. 이 알고리즘의 의사코드는 그림 4.4에 나와 있습니다. BuildHuffman은 $O(n)$번의 우선순위 큐 연산을 수행하며, 정확히는 $2n-1$번의 Insert와 $2n-2$번의 ExtractMin을 수행합니다. 우선순위 큐를 표준 이진 힙으로 구현하면 이 연산들은 각각 $O(\log n)$ 시간이 걸리므로, 전체 알고리즘은 $\boldsymbol{O(n \log n)}$ *시간*에 실행됩니다.

마지막으로, 고정된 허프만 코드를 사용하여 메시지를 인코딩하고 디코딩하는 간단한 알고리즘을 그림 4.5에 제시합니다. 두 알고리즘 모두 $O(m)$ 시간에 실행되며, 여기서 m은 인코딩된 메시지의 길이입니다.

4.5 안정 매칭

매년 수천 명의 신규 의사들이 미국 전역의 병원에서 인턴 자리를 얻어야 합니다. 20세기 전반기에는 최고의 의사를 확보하려는 병원 간의 경쟁 때문에 인턴 자리 제안이 점점 더 앞당겨졌고, 때로는 의과대학 2학년 때 이미 제안이 오기도 했으며, 수락 기한도 점점 촉박해졌습니다. 1940년대에 의과대학들은 학생들의 4학년 중 공통된 날짜가 되기 전까지는 정보를 공개하지 않기로 합의했습니다. 이에 대응하여 병원들은 더

```
BuildHuffman(f[1..n]):
  for i ← 1 to n
      L[i] ← 0; R[i] ← 0
      Insert(i, f[i])
  for i ← n to 2n − 1
      x ← ExtractMin()        《가장 드문 기호 두 개를 찾습니다》
      y ← ExtractMin()
      f[i] ← f[x] + f[y]      《새 기호로 병합합니다》
      Insert(i, f[i])
      L[i] ← x; P[x] ← i      《트리 포인터 갱신》
      R[i] ← y; P[y] ← i
  P[2n − 1] ← 0
```

그림 4.4. 허프만 코드 구성하기.

```
HuffmanEncode(A[1..k]):
  m ← 1
  for i ← 1 to k
      HuffmanEncodeOne(A[i])

HuffmanEncodeOne(x):
  if x < 2n − 1
      HuffmanEncodeOne(P[x])
      if x = L[P[x]]
          B[m] ← 0
      else
          B[m] ← 1
      m ← m + 1
```

```
HuffmanDecode(B[1..m]):
  k ← 1
  v ← 2n − 1
  for i ← 1 to m
      if B[i] = 0
          v ← L[v]
      else
          v ← R[v]
      if L[v] = 0
          A[k] ← v
          k ← k + 1
          v ← 2n − 1
```

그림 4.5. 허프만 코드의 인코딩 및 디코딩 알고리즘

빠른 결정을 요구하기 시작했습니다. 1950년 무렵에는 병원들이 의사들에게 전화를 걸어 인턴 자리를 제안하고 *즉각적인* 답변을 요구하는 일이 예사가 되었습니다. 세 번째로 원하던 병원에서 먼저 전화가 오면 인턴들은 도박을 할 수밖에 없었습니다—수락하고 나중에 더 좋은 기회를 놓칠 위험을 감수하거나, 거절하고 아예 자리를 얻지 못할 위험을 감수해야 했습니다.[8]

마침내 1950년대 초에 인턴십 배정을 위한 중앙 조정 기관이 설립되었으며, 이 기관은 현재 전국 레지던트 매칭 프로그램(National Resident Matching Program, NRMP)이라고 불립니다. 매년 의사들은 인턴십을 받아들일 의향이 있는 모든 병원의 순위 목록을 제출하고, 각 병원은 인턴으로 받아들일 의향이 있는 의사들의 순위 목록을 제출합니다. 그러면 NRMP는 다음의 *안정성* 요건을 만족하는 의사와 병원 사이의 매칭을 계산합니다. 어떤 의사 α와 병원 B가 둘 다 각자의 현재 매칭 상대보다 서로를

[8] 미국의 학계 채용 시장에도, 적어도 컴퓨터 과학 분야에서는 비슷한 도박이 따릅니다. 어떤 학과는 2주의 결정 기한을 두고 2월부터 제안을 하기 시작하고, 다른 학과는 3월이 되어서야 면접을 시작하기도 합니다. MIT는 모든 면접이 끝나는 5월까지 기다린 뒤에야 비로소 교수직 제안을 *하나라도* 내놓는 것으로 악명이 높습니다. 말할 필요도 없이, 제각각인 제안 날짜와 결정 기한은 지원자와 학과 모두에게 엄청난 스트레스를 안겨 줍니다. 비슷한 이유로 1965년부터 대부분의 미국 대학들은 대학원 진학 예정자들이 재정 지원 제안(나아가 입학 제안)을 수락하는 공통 기한을 4월 15일로 정하는 데 합의했습니다.

더 선호한다면, 그 매칭은 ***불안정***하다고 합니다. 즉,

- α는 B를 더 선호함에도 불구하고 다른 병원 A와 매칭되어 있습니다.
- B는 α를 더 선호하는데도 다른 의사 β와 매칭됩니다.

이 경우 (α, B)를 해당 매칭의 *불안정 쌍*이라고 부릅니다. Resident Match의 목표는 ***안정 매칭***, 즉 불안정 쌍이 없는 매칭입니다.

편의상 지금부터는 의사와 병원의 수가 정확히 같고, 각 병원은 정확히 한 개의 인턴 자리를 제공하며, 각 의사는 모든 병원에 순위를 매기고 병원도 마찬가지로 모든 의사에게 순위를 매기며, 마지막으로 의사나 병원이 매긴 순위에는 동순위가 없다고 가정하겠습니다.[9]

몇 가지 나쁜 아이디어

언뜻 보면 안정 매칭이 항상 존재하는지조차 분명하지 않습니다! 의사와 병원 사이의 매칭이 모두 안정적인 것은 결코 아닙니다. 세 명의 의사(Quincy 박사, 로트방 박사, Shephard 박사이며 소문자로 나타냅니다)와 세 개의 병원(아캄 정신병원, 베들렘 왕립병원, 카운티 종합병원이며 대문자로 나타냅니다)이 있고, 이들이 서로를 다음과 같이 순위 매긴다고 가정합시다:

q	r	s
A	C	A
C	A	B
B	B	C

A	B	C
r	s	q
q	q	r
s	r	s

매칭 $\{Aq, Br, Cs\}$는 불안정합니다. 아캄은 퀸시 박사보다 로트방 박사를 고용하기를 원하고, 로트방 박사는 베들럼보다 아캄에서 일하기를 원하기 때문입니다. (A, r)은 이 매칭의 불안정한 쌍입니다.

임의의 매칭에서 시작한 다음 탐욕적으로 교환을 수행하여 불안정성을 해소하는 점진적 알고리즘을 생각해 볼 수 있습니다. 안타깝게도 불안정성 하나를 해소하면 새로운 불안정성이 생길 수 있습니다. 실제로 이러한 점진적 "개선"은 무한 루프로 이어질 수 있습니다. 예를 들어 앞에서 본 불안정한 매칭 $\{Aq, Br, Cs\}$에서 시작하면, 다음의 각 교환은 불안정한 쌍 하나(화살표 위에 표시됨)를 해소하지만, 이 교환들을 차례로 수행하면 원래의 매칭으로 되돌아옵니다.[10]

$$\{Aq, Br, Cs\} \xrightarrow{Ar} \{Ar, Bq, Cs\} \xrightarrow{Cr} \{As, Bq, Cr\} \xrightarrow{Cq} \{As, Br, Cq\} \xrightarrow{Aq} \{Aq, Br, Cs\}$$

또는 다음과 같은 여러 라운드로 이루어진 탐욕 프로토콜을 시도해 볼 수도 있습니다. 각 라운드에서 매칭되지 않은 모든 병원은 매칭되지 않은 의사 중 가장 선호하는 의사에게 제안을 하고, 그런 다음 제안을 받은 매칭되지 않은 모든 의사는 받은 제안 중 가장 선호하는 제안을 수락합니다. 각 라운드에서 적어도 하나의 새로운 의사-병원

[9] 실제로는 대부분의 병원이 여러 개의 인턴 자리를 제공하고, 각 의사는 병원 중 일부에만 순위를 매기며 병원도 마찬가지이고, 일반적으로 인턴 자리가 지원하려는 의사보다 많습니다. 그러면 문제가 복잡해지기 시작합니다.

[10] 이 예는 도널드 커누스(Donald Knuth)가 발견했습니다.

쌍이 매칭된다는 것은 어렵지 않게 증명할 수 있으므로, 이 알고리즘은 항상 매칭으로 끝납니다. 앞의 예시 입력에 대해서는 첫 번째 라운드가 끝났을 때 이미 안정 매칭 $\{Ar, Bs, Cq\}$를 얻습니다! 하지만 대신 다음 입력을 생각해 보십시오:

q	r	s
C	A	A
B	C	B
A	B	C

A	B	C
q	q	s
s	r	r
r	s	q

첫 번째 라운드에서 셰퍼드 박사는 카운티의 제안을 수락하고, 퀸시 박사는 (아캄의 제안을 거절하고) 베들럼의 제안을 수락하므로, 로트방 박사와 아캄만 매칭되지 않은 채로 남습니다. 따라서 이 프로토콜은 두 라운드 후에 매칭 $\{Ar, Bq, Cs\}$로 끝납니다. 안타깝게도 이 매칭은 불안정합니다. 아캄과 셰퍼드 박사는 각자의 매칭 상대보다 서로를 더 선호하기 때문입니다.

보스턴 풀 알고리즘과 Gale-Shapley 알고리즘

1952년에 NRMP는 인턴 배정을 위해 "보스턴 풀(Boston Pool)" 알고리즘을 채택했습니다. 이 알고리즘은 이전에 보스턴 지역의 한 지역 중개 기관에서 사용되었기 때문에 이런 이름이 붙었습니다. 10년 후, 데이비드 게일(David Gale)과 로이드 섀플리(Lloyd Shapley)는 보스턴 풀 알고리즘을 일반화한 알고리즘을 기술하고 형식적으로 분석했으며, 이 알고리즘이 안정 매칭을 계산한다는 것을 증명했습니다. 게일과 섀플리는 대학 입학이라는 비유를 사용했습니다. 본질적으로 같은 알고리즘을 1972년에 엘리엇 페란슨(Elliott Peranson)이 의과대학 입학에 사용하기 위해 독자적으로 개발했습니다. 그 이후로 비슷한 알고리즘이 프랑스의 교수 채용, 미국의 신규 경제학 박사 채용, 독일의 대학 입학, 뉴욕과 보스턴의 공립학교 입학, 미 해군 수병의 보직 배정, 신장 매칭 프로그램을 비롯한 다른 여러 매칭 시장에서 채택되었습니다.

섀플리는 안정 매칭에 관한 연구로 앨빈 로스와 함께 2012년 노벨 경제학상을 받았습니다. 로스는 섀플리의 연구를 크게 확장했으며, 이를 활용해 여러 실제 교환 제도를 개발했습니다. (게일은 2008년에 세상을 떠났기 때문에 상을 함께 받지 못했습니다.)

앞서 실패했던 탐욕 알고리즘과 마찬가지로, Gale-Shapley 알고리즘도 모든 자리가 수락될 때까지 여러 라운드에 걸쳐 진행됩니다. 각 라운드는 두 단계로 이루어집니다:

1. 아직 짝이 정해지지 않은 임의의 병원 A는 자신을 아직 거절하지 않은 의사 중 (A의 선호 목록에 따라) 가장 선호하는 의사 α에게 자리를 제안합니다.

2. α에게 아직 짝이 없으면, 그녀는 A의 제안을 (잠정적으로) 받아들입니다. α에게 이미 짝이 있지만 A를 더 선호한다면, 그녀는 현재의 짝을 거절하고 A의 새 제안을 (잠정적으로) 받아들입니다. 그렇지 않으면 α는 새 제안을 거절합니다.

각 의사는 최종적으로 자신의 선호 목록에 따라 받은 제안 중 가장 좋은 제안을 수락합니다.[11] 요컨대 병원은 탐욕적으로 제안하고, 의사는 탐욕적으로 제안을 수락합니

[11] 1952년의 보스턴 풀 알고리즘은 제안을 특정한 순서로 실행하는 Gale-Shapley 알고리즘의 특수한 경

다. 이 상호 탐욕 전략이 작동하게 하는 핵심은 의사가 더 좋은 제안을 받으면 현재의 매칭을 거절할 수 있다는 점입니다.

예를 들어, 네 명의 의사(퀸시 박사, 로트방 박사, 셰퍼드 박사, 탬 박사)와 네 곳의 병원(아캄 어사일럼, 베들럼 왕립 병원, 카운티 종합병원, 다르마 이니셔티브)이 있고, 이들이 서로의 순위를 다음과 같이 매긴다고 가정합시다:

q	r	s	t
A	A	B	D
B	D	A	B
C	C	C	C
D	B	D	A

A	B	C	D
t	r	t	s
s	t	r	r
r	q	s	q
q	s	q	t

이러한 선호 목록이 입력으로 주어지면, Gale-Shapley 알고리즘은 다음과 같이 진행될 수 있습니다.

1. 아캄이 탐 박사에게 제안을 합니다.
2. 베들럼이 로트방 박사에게 제안을 하나 합니다.
3. 카운티가 탐 박사에게 제안을 하고, 탐 박사는 앞서 아캄에서 받은 제안을 거절합니다.
4. 다르마가 셰퍼드 박사에게 제안을 합니다. (이 시점부터는 매칭되지 않은 병원이 하나뿐이므로, 알고리즘에는 더 이상 선택의 여지가 없습니다.)
5. 아캄은 셰퍼드 박사에게 제안을 하고, 셰퍼드 박사는 앞서 다르마에게서 받은 제안을 거절합니다.
6. 다르마가 로트방 박사에게 제안을 하고, 로트방 박사는 앞서 베들럼에서 받았던 제안을 거절합니다.
7. 베들럼이 탐 박사에게 제안하고, 탐 박사는 앞서 받은 카운티의 제안을 거절합니다.
8. 카운티는 로트방 박사에게 제안을 하지만, 로트방 박사는 이를 거절합니다.
9. 카운티는 셰퍼드 박사에게 제안을 하지만, 셰퍼드 박사는 이를 거절합니다.
10. 카운티가 퀸시 박사에게 제안을 합니다.

열 번째 라운드가 끝나면 대기 중인 제안이 모두 수락되고, 알고리즘이 반환하는 매칭은 $\{As, Bt, Cq, Dr\}$입니다. 어떤 의사도 자신이 가장 선호하는 병원에 고용되지 않았고 어떤 병원도 가장 선호하는 의사를 고용하지 못했지만, 이 매칭이 안정적이라는 것은 전수 조사로 확인할 수 있으며 또 직접 확인해 보아야 합니다. 실제로 County는 결국 *가장 덜* 선호하는 의사를 고용하게 되었습니다. 이 선호 목록에 대한 안정 매칭이 이것 하나만 있는 것은 아닙니다. 매칭 $\{Ar, Bs, Cq, Dt\}$ 역시 안정적입니다.

우입니다. 대략적으로 말하면, 각 제안은 가장 선호하는 의사가 (X를 아직 거절하지 않은 의사 중에서) X를 가장 높은 순위에 두는 병원 X가 합니다. 제안의 순서가 선호 목록 전체에 의존하기 때문에, 이 알고리즘은 *반드시* 중앙 기관이 실행해야 합니다. 반면 Gale-Shapley 알고리즘에서는 각 참여자가 어떤 고정된 순위에 일관되게 행동하기만 하면, 자신의 선호를 미리 알 필요조차 없습니다.

실행 시간

알고리즘이 수행하는 제안 횟수를 분석하는 것은 비교적 간단합니다(그래서 이것부터 다루는 것입니다). 각 병원은 각 의사에게 최대 한 번만 제안하므로, 알고리즘은 최대 n^2번 제안합니다.

그러나 실제 *실행 시간*을 분석하려면 알고리즘을 더 자세히 명시해야 합니다. 선호 목록은 알고리즘에 어떤 형태로 주어집니까? 알고리즘은 매칭되지 않은 병원이 있는지 어떻게 판단하며, 있다면 그러한 병원을 어떻게 찾습니까? 알고리즘은 임시 매칭을 어떻게 저장합니까? 알고리즘은 의사가 현재 매칭보다 새 제안을 더 선호하는지 어떻게 판단합니까? 가장 근본적인 질문은 이것입니다. *알고리즘은 의사와 병원을 실제로 어떻게 표현합니까?*

한 가지 방법은 각 의사와 병원을 1부터 n 사이의 고유한 정수로 나타내고, 선호도를 두 배열 $Dpref[1..n, 1..n]$과 $Hpref[1..n, 1..n]$으로 나타내는 것입니다. 여기서 $Dpref[i, r]$은 의사 i의 선호 목록에서 r번째 병원을 나타내고, $HPref[j, r]$은 병원 j의 선호 목록에서 r번째 의사를 나타냅니다. 입력이 이러한 형태로 주어지면, 보스턴 풀 알고리즘은 약간의 초기 전처리를 거친 후 각 제안을 상수 시간에 실행할 수 있습니다. 따라서 전체 구현은 $O(n^2)$ *시간*에 실행됩니다. 나머지 세부 사항은 간단한 연습 문제로 남겨 둡니다.

이보다 조금 더 어려운 연습 문제로, 알고리즘이 멈추기 전에 $\Omega(n^2)$번의 제안이 이루어질 수밖에 없게 만드는 입력(그리고 누가 언제 제안할지에 대한 선택)이 존재함을 증명하는 문제가 있습니다. 따라서 최악의 경우 실행 시간에 대한 $O(n^2)$ 상한은 *점근적으로 빈틈없는* 상한입니다.

정확성

그런데 이 알고리즘은 도대체 왜 *올바른* 것입니까? 이 알고리즘이 항상 안정 매칭을, 아니 그 이전에 *어떤* 완전 매칭이라도 계산한다는 것을 우리는 어떻게 알 수 있습니까?

의사는 일단 제안을 받으면 그 이후로 계속 적어도 잠정적인 매칭을 유지합니다. 이를 동치인 명제로 바꾸어 말하면, 어떤 의사가 매칭되지 않았다면 어떤 병원도 그 의사에게 일자리를 제안하지 않은 것이며, 이는 병원들이 아직 선호 목록을 모두 소진하지 않았음을 의미합니다. 따라서 알고리즘이 (최대 n^2 라운드 후에) 종료될 때 모든 의사가 매칭되어 있으며, 그러므로 모든 자리가 채워집니다. 다시 말해, 이 알고리즘은 항상 의사와 병원 사이의 완전 매칭을 계산합니다. (휴!) 이제 남은 것은 그 결과로 얻은 매칭이 안정적임을 증명하는 것뿐입니다.

알고리즘이 어떤 의사 α를 어떤 병원 A에 매칭했지만, 그녀는 다른 병원 B를 더 선호한다고 가정합시다. 모든 의사는 자신이 받은 제안 중 가장 좋은 제안을 수락하므로, α는 A보다 더 마음에 드는 제안을 하나도 받지 못했습니다. 특히 B는 α에게 한 번도 제안하지 않았습니다. 반면에 B는 최종 매칭 상대인 β보다 더 선호하는 모든 의사에게 제안을 했습니다. 따라서 B는 α보다 β를 선호하며, 이는 (α, B)가 불안정한 쌍이 *아니라는* 뜻입니다. 결론적으로 불안정한 쌍은 존재하지 않습니다. 즉, 이 매칭은 안정적입니다!

최적성!

놀랍게도 Gale-Shapley 알고리즘의 정확성은 각 라운드에서 어느 병원이 제안하는지에 좌우되지 않습니다. 실제로 각 라운드에서 배정되지 않은 병원 중 어느 병원이 제안하더라도 *알고리즘은 항상 같은 매칭을 계산합니다*! 의사 α를 병원 A에 배정하는 안정 매칭이 존재하면 α를 A에 대해 ***실현 가능한*** 의사라고 합시다.

보조정리 4.7. *Gale-Shapley 알고리즘이 진행되는 동안, 각 병원 A는 A에게 실현 불가능한 의사들에게서만 거절당합니다.*

증명: 라운드 수에 대한 귀납법으로 이 보조정리를 증명합니다. 의사 α가 한 병원 A를 거절하고 다른 병원 B를 택하는 알고리즘의 임의의 라운드를 생각해 봅시다. 이 거절은 α가 A보다 B를 선호한다는 것을 의미합니다. B의 선호 목록에서 α보다 높은 순위에 있는 각 의사는 이전 라운드에서 이미 B를 거절했으므로, 귀납 가정에 의해 B에게 실현 불가능합니다.

이제 (같은 의사들과 병원들에 대해) α를 A에 배정하는 임의의 매칭을 생각해 봅시다. α가 A보다 B를 선호한다는 것은 이미 보였습니다. 만약 B가 자신의 짝보다 α를 선호한다면, 그 매칭은 불안정합니다. 반면에 B가 α보다 자신의 짝을 선호한다면, (앞선 논증에 의해) 그 짝은 실현 불가능하므로 이 경우에도 그 매칭은 불안정합니다. 따라서 α를 A에 배정하는 안정 매칭은 존재하지 않는다는 결론을 내립니다. □

이제 $\boldsymbol{best(A)}$가 A의 선호 목록에서 가장 높은 순위에 있는 *실현 가능한* 의사를 나타낸다고 합시다. 보조정리 4.7에 따르면 A가 최종 매칭 상대보다 선호하는 모든 의사는 A에게 실현 불가능합니다. 한편 최종 매칭은 안정적이므로 A에 배정된 의사는 반드시 A에게 실현 가능해야 합니다. 이로써 다음 결과가 바로 도출됩니다:

따름정리 4.8. *Gale-Shapley 알고리즘은 모든 병원 A에 대해 $best(A)$를 A와 매칭합니다.*

다시 말해, Gale-Shapley 알고리즘은 병원의 관점에서 *가능한 최선의* 안정 매칭을 계산합니다. 그런데 이 매칭은 의사의 관점에서는 *가능한 최악의* 매칭이기도 합니다! $\boldsymbol{worst(\alpha)}$를 의사 α의 선호 목록에서 순위가 가장 낮은 실현 가능한 병원이라고 합시다.

따름정리 4.9. *Gale-Shapley 알고리즘은 모든 의사 α에 대해 α를 $worst(\alpha)$와 매칭합니다.*

증명: 게일과 섀플리가 의사 α를 병원 A에 배정한다고 가정합시다. 이제 $A = worst(\alpha)$임을 보여야 합니다. A가 α가 *아닌* 다른 의사 β와 짝지어진 임의의 안정 매칭을 생각해 봅시다. 앞의 따름정리에 따르면 A는 β보다 $\alpha = best(A)$를 선호합니다. 이 매칭은 안정적이므로, α는 A보다 자신이 배정받은 병원을 선호해야 합니다. 이 논증은 *어떤* 안정 매칭에 대해서도 성립하므로, α는 A보다 다른 *모든* 가능한 짝을 선호합니다. 다시 말해 $A = worst(\alpha)$입니다. □

1981년 Lester Dubins와 David Freedman이 발견한 이 두 따름정리의 미묘한 귀결은, 의사는 자신의 선호를 속여 매칭을 개선할 수도 있지만 병원은 그럴 수 없다는 것입니다. (다만 여러 병원이 담합하면 그 병원들의 매칭 *중 일부*를 개선할 수 있습니다.) 부분적으로 이러한 이유로 전국 레지던트 매칭 프로그램(National Residency Matching Program)은 1998년에 매칭 알고리즘을 뒤집었습니다. 그 결과 레지던트 지원자들이 자신의 선호 순서에 따라 병원에 근무를 제안하고, 각 병원은 받은 제안

중 가장 좋은 것을 수락하게 되었습니다. 따라서 새 알고리즘은 의사에게는 가능한 최선의 안정 매칭을, 병원에게는 가능한 최악의 안정 매칭을 계산합니다. 그러나 실제로 이 전환으로 바뀐 레지던트 매칭은 1% 미만이었습니다. 제가 아는 한, 이 변화가 *환자*에게 미친 정확한 영향은 아직 풀리지 않은 문제입니다.

연습문제

독자 유의 사항: 이 연습 문제 중 일부는 탐욕 알고리즘으로 *풀 수 없습니다*! 탐욕 알고리즘을 설명하고 분석할 때에는 알고리즘이 올바르다는 증명도 ***반드시*** 함께 제시해야 합니다. 이 증명은 보통 교환 논증의 형태를 띱니다. 이러한 증명은 (제 수업처럼) 평소에 정확성 증명을 요구하지 *않는* 수업에서 특히 중요합니다.

1. 수업 스케줄링 문제를 위해 앞에서 설명한 GreedySchedule 알고리즘이 우리가 시도해 볼 수 있었던 유일한 탐욕 전략은 아닙니다. 다음의 각 대안 탐욕 전략에 대해, 그 결과로 얻어지는 알고리즘이 항상 최적의 스케줄을 구성함을 증명하거나, 그 알고리즘이 최적의 스케줄을 만들어 내지 *못하는* 작은 입력 예를 제시하십시오. 모든 알고리즘은 동점인 경우를 임의로(즉, 여러분이 전혀 통제할 수 없는 방식으로) 처리한다고 가정하십시오. *[힌트: 이 알고리즘들 중 세 개는 실제로 올바릅니다.]*

   (a) *가장 늦게 끝나는* 강의 x를 선택하고, x와 겹치는 강의들을 제외한 뒤 재귀합니다.

   (b) *가장 먼저 시작하는* 강의 x를 선택하고, x와 겹치는 모든 강의를 버린 다음, 나머지 강의에 대해 재귀적으로 반복합니다.

   (c) *가장 늦게 시작하는* 강의 x를 선택하고, x와 겹치는 모든 강의를 제외한 다음, 재귀적으로 진행합니다.

   (d) *지속 시간이 가장 짧은* 강의 x를 선택하고, x와 겹치는 모든 강의를 제외한 다음 재귀적으로 반복합니다.

   (e) *다른 강좌와 가장 적게 충돌하는* 강좌 x를 선택하고, x와 충돌하는 모든 강좌를 버린 뒤 재귀합니다.

   (f) 겹치는 수업이 없으면 모두 선택합니다. 그렇지 않으면 *수강 시간이 가장 긴* 수업을 버리고 재귀합니다.

   (g) 서로 충돌하는 수업이 없으면 모두 선택합니다. 그렇지 않으면 *다른 과목들과 가장 많이 충돌하는* 과목 하나를 버리고 재귀합니다.

   (h) x를 *시작 시각이 가장 이른* 수업이라 하고, y를 *시작 시각이 두 번째로 이른* 수업이라 합시다.

   - x와 y가 서로소이면, x를 선택하고 x를 제외한 나머지 전체에 대해 재귀합니다.
   - x가 y를 완전히 포함하면 x를 버리고 재귀합니다.
   - 그렇지 않으면 y를 버리고 재귀 호출합니다.

(i) 어떤 강의 x가 다른 강의를 완전히 포함하면, x를 버리고 재귀합니다. 그렇지 않으면 *가장 늦게 끝나는* 강의 y를 선택하고, y와 겹치는 모든 강의를 버린 다음 재귀합니다.

2. 이제 수업 스케줄링 문제의 가중 버전을 생각해 봅시다. 이 버전에서는 수업마다 제공하는 학점 수가 서로 다릅니다(수업 강의 시간의 길이와는 전혀 관계가 없습니다). 이제 목표는 시작 시각, 종료 시각, 학점 수의 배열이 입력으로 주어졌을 때, 서로 겹치지 않으면서 학점 수의 합이 가능한 한 가장 큰 수업들의 집합을 선택하는 것입니다.

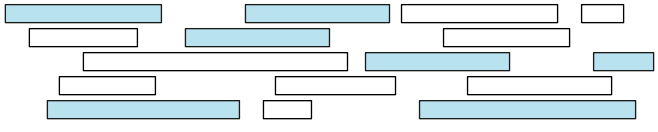
   (a) 이 장의 서두에서 설명한 탐욕 알고리즘, 즉 가장 먼저 끝나는 수업을 선택하고 재귀하는 알고리즘이 항상 최적의 시간표를 반환하지는 *않음*을 증명하십시오.

   (b) 연습문제 1에서 설명한 탐욕 알고리즘 중 ***어느 것도*** 항상 최적의 스케줄을 반환하지는 않음을 증명하십시오. *[힌트: 먼저 연습문제 1을 푸십시오. 거기서 작동하지 않는 알고리즘은 여기서도 작동하지 않습니다.]*

   (c) 항상 최적의 스케줄을 계산하는 알고리즘을 설명하고 분석하십시오. *[힌트: 여러분의 알고리즘은 탐욕 알고리즘이 아닐 것입니다.]*

3. X를 실수 직선 위의 n개 구간으로 이루어진 집합이라고 합시다. 구간들의 부분집합 $Y \subseteq X$에 속한 모든 구간의 합집합이 X에 속한 모든 구간의 합집합과 같으면 Y가 X를 *덮는다*고 합니다. 덮개의 *크기*는 단순히 덮개에 속한 구간의 개수입니다.

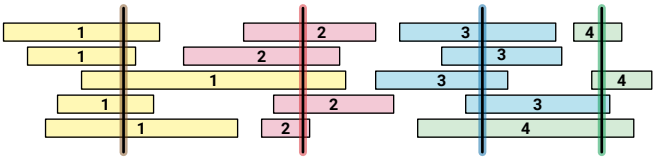
   X의 최소 덮개를 계산하는 효율적인 알고리즘을 설명하고 분석하십시오. 입력은 X에 속한 구간들의 왼쪽 끝점과 오른쪽 끝점을 나타내는 두 배열 $L[1..n]$과 $R[1..n]$으로 이루어진다고 가정하십시오. 탐욕 알고리즘을 사용한다면 그 알고리즘이 올바름을 반드시 증명해야 합니다.



구간들의 집합과 크기가 7인 덮개(음영 표시).

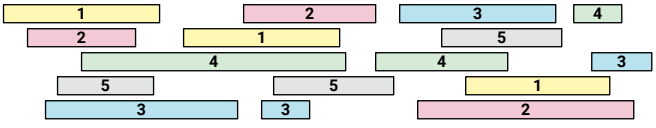
4. X를 실수 직선 위의 구간 n개로 이루어진 집합이라고 합시다. X의 모든 구간이 P의 점을 적어도 하나 포함하면, 점들의 집합 P가 X를 *찌른다*고 말합니다. X를 찌르는 가장 작은 점 집합을 계산하는 효율적인 알고리즘을 기술하고 분석하십시오. 입력은 X에 속한 구간들의 왼쪽 끝점과 오른쪽 끝점을 나타내는 두 배열 $L[1..n]$과 $R[1..n]$으로 이루어져 있다고 가정하십시오. 늘 그렇듯이, 탐욕 알고리즘을 사용한다면 그것이 올바르다는 것을 증명해야 합니다.

5. X를 실수 직선 위의 n개 구간으로 이루어진 집합이라고 합시다. X의 *적절한 색칠*은 서로 겹치는 두 구간에 항상 다른 색이 배정되도록 각 구간에 색을 하나씩



네 점에 찔린 구간들의 집합(여기서는 점을 수직 선분으로 나타냈습니다)

배정하는 것입니다. X를 적절하게 색칠하는 데 필요한 최소 색의 수를 계산하는 효율적인 알고리즘을 설명하고 분석하십시오. 입력은 X에 속한 구간들의 왼쪽 끝점과 오른쪽 끝점을 나타내는 두 배열 $L[1..n]$과 $R[1..n]$으로 이루어져 있다고 가정하십시오. 늘 그렇듯이, 탐욕 알고리즘을 사용한다면 그 알고리즘이 올바르다는 것을 반드시 증명해야 합니다.



다섯 가지 색을 사용한 구간 집합의 올바른 색칠.

6. (a) 모든 정수 n에 대해, 허프만 코드 트리의 깊이가 $n-1$이 되는 빈도 배열 $f[1..n]$ 중에서 가장 큰 빈도가 가능한 한 작은 배열을 찾으십시오.

   (b) *부호화되지 않은* 메시지의 전체 길이 N이 알파벳 크기 n에 대한 다항식으로 상계된다고 가정합시다. 빈도 $f[1..n]$에 대한 임의의 허프만 트리의 깊이가 $O(\log n)$임을 증명하십시오.

♥7. 빈도 배열 $f[1..n]$이 다음 두 조건을 만족하면 이를 ***α*-무거운*** 배열이라고 부릅니다.

- 모든 $i > 1$에 대해 $f[1] > f[i]$입니다. 즉, 1은 *유일한* 최빈 기호입니다.
- $f[1] \geq \alpha \sum_{i=1}^{n} f[i]$입니다. 즉, 기호 중 적어도 α 비율이 1입니다.

모든 α-무거운 빈도 배열에 대한 모든 허프만 코드에서 기호 1이 단일 비트로 표현되도록 하는 가장 큰 실수 α를 구하십시오. *[힌트: 먼저 $1/3 \leq \alpha \leq 1/2$임을 증명하십시오.]*

8. 주어진 빈도 배열 $f[1..n]$에 대해 최적의 *삼진* 접두사 없는 코드를 계산하는 알고리즘을 설명하고 분석하십시오. 여러분의 알고리즘이 모든 n에 대해 올바르다는 것을 증명하는 일을 잊지 마십시오.

9. 이 장의 앞부분에서 주장한 대로 최악의 경우 실행 시간이 $O(n^2)$이 되도록 Gale-Shapley 안정 매칭 알고리즘을 구현하는 방법을 자세히 설명하십시오.

10. (a) Gale-Shapley 알고리즘이 종료되기 전에 $\Omega(n^2)$번의 제안을 수행할 수 있음을 증명하십시오. (적절한 입력과, $\Omega(n^2)$개의 유효한 제안으로 이루어진 순서를 모두 제시해야 합니다.)

    (b) 임의의 정수 n에 대해, *각 라운드에서 어떤 유효한 제안이 이루어지든 관계없이* Gale-Shapley 알고리즘이 $\Omega(n^2)$번의 라운드를 수행하게 만드는 n명의 의사와 n개의 병원에 대한 선호도 집합을 기술하십시오. *[힌트: (b)를 풀면 (a)도 풀립니다.]*

11. 병원과 의사의 선호도 집합이 주어졌을 때, 안정 매칭이 *유일한지* 판별하는 효율적인 알고리즘을 설명하고 분석하십시오.

12. 안정 매칭 문제를 다음과 같이 일반화한 경우를 생각해 봅시다. 일부 의사는 모든 병원의 순위를 매기지는 않고 일부 병원도 모든 의사의 순위를 매기지는 않으며, 의사는 자신과 병원이 서로 상대방의 선호 목록에 나타나는 경우에만 그 병원에 배정될 수 있습니다. 이 경우에는 다음과 같은 세 가지 불안정한 상황이 추가로 존재합니다:

    - 짝이 정해진 병원이 자신에게 배정된 짝보다 짝이 없는 의사를 더 선호합니다.
    - 매칭된 의사가 자신에게 배정된 병원보다 매칭되지 않은 병원을 더 선호합니다.
    - 매칭되지 않은 의사와 매칭되지 않은 병원이 서로의 선호 목록에 나타납니다.

    이 설정에서는 선호 목록이 비어 있지 않더라도 안정 매칭에서 일부 의사나 병원, 또는 둘 다 매칭되지 않은 채로 남을 수 있습니다. 예를 들어 모든 의사가 하버드를 유일하게 받아들일 수 있는 병원으로 적고, 모든 병원이 하우스 박사를 유일하게 받아들일 수 있는 인턴으로 적는다면, 하우스와 하버드만 매칭됩니다.

    이처럼 더 일반적인 상황에서 안정 매칭을 계산하는 효율적인 알고리즘을 설명하고 분석하십시오. *[힌트: 모든 의사가 모든 병원의 순위를 매기고 모든 병원도 모든 의사의 순위를 매기는 인스턴스로 환원한 다음, Gale-Shapley 알고리즘을 호출하십시오.]*

13. 스칸디나비아 가구 회사 Fürni는 델라웨어주 윌밍턴의 서로 다른 주소 n곳에 똑같은 주문품 n개를 배달하려고 운전기사 n명을 고용했습니다. 운전기사마다 윌밍턴을 도는 오래된 자신만의 배달 경로가 있으며, 이 경로는 주소 n곳을 모두 방문합니다. 운전기사들이 늘 하던 대로 각자의 경로를 따른다고 가정하면, 두 운전기사가 같은 시각에 같은 주소를 방문하는 일은 결코 없습니다.

    원칙적으로 n명의 운전기사는 각자 n개의 주소 중 어느 곳으로든 가구를 배달할 수 있지만, 한 가지 복잡한 문제가 있습니다. 운전기사 중 한 명이 (Strinne 초록색 줄무늬 패턴의) Johannshamn 소파에 근접 센서와 폭발물을 몰래 설치해 두었습니다. 두 소파가 같은 시각에 같은 주소에 있게 되면 두 소파 모두 폭발하여 배달 트럭과 그 주소의 건물이 모두 파괴됩니다. 이런 일은 한 운전기사가 어떤 주소에 주문품을 배달한 뒤, 나중에 다른 운전기사가 *아직 트럭에 가구를 실은 채로* 같은 주소를 방문하는 경우에만 일어날 수 있습니다.

Fürni 배차 담당자로서 여러분이 할 일은 각 운전기사를 배달 주소 하나에 배정하는 것입니다. n개의 주소가 모두 각자의 가구 주문품을 받고 폭발이 한 번도 일어나지 않도록 운전기사에게 주소를 배정하는 알고리즘을 설명하십시오.

예를 들어, Jack의 경로는 오후 6시에 Paper Street 537번지를, 오후 8시에 Franklin Street 1888번지를 방문하고, Marla의 경로는 오후 7시에 Paper 537번지를, 오후 9시에 Franklin 1888번지를 방문한다고 가정합시다. 그러면 Jack이 Franklin 1888번지에 배달하고 Marla가 Paper 537번지에 배달해야 합니다. 그렇지 않으면 오후 8시에 Franklin 1888번지에서 폭발이 일어날 것입니다. (여기서 Pixies의 노래가 흘러나옵니다.) *[힌트: Jack과 Marla는 다소 불안정한 사람들입니다.]*

14. 여러분이 서로 다른 n가지 종류의 동전이 있는 나라에 사는 평범한 가게 주인이라고 가정합시다. 동전의 액면가는 $1 = c[1] < c[2] < \cdots < c[n]$입니다. (예를 들어 미국에서는 $n = 6$이고, 액면가는 1, 5, 10, 25, 50, 100센트입니다.) 여러분이 사랑해 마지않는 자애로운 독재자 엘 헤네랄리시모(El Generalissimo)께서는 손님에게 거스름돈을 줄 때마다 가능한 한 가장 적은 수의 동전을 사용해야 한다고 명하셨습니다. 왕립 재무국의 신하들이 각 동전에 정성껏 새겨 넣은 엘 헤네랄리시모의 초상이 닳지 않도록 하기 위함입니다.

    (a) 미국에는 항상 가장 적은 개수의 동전을 내주는 간단한 탐욕 알고리즘이 있습니다. 가장 큰 동전을 빼고 나머지 금액에 대한 거스름돈을 재귀적으로 내주는 것입니다. 엘 헤네랄리시모는 미국 자본주의의 탐욕을 용납하지 않습니다. 탐욕 알고리즘이 가능한 가장 적은 개수의 동전을 항상 내주지는 *않는* 동전 액면가 집합이 존재함을 보이십시오.

    (b) 이제 엘 헤네랄리시모가 어떤 정수 $b \geq 2$의 연속된 거듭제곱 $b^0, b^1, b^2, \ldots, b^k$를 동전 액면가로 삼는 화폐 체계를 도입하기로 했다고 가정합시다. 엘 헤네랄리시모는 못마땅하게 여기겠지만, (a)에서 설명한 탐욕 알고리즘이 이 화폐 체계에서는 실제로 최적의 거스름돈을 만든다는 것을 증명하십시오.

    (c) 대상 금액 T와 동전 액면가의 정렬된 배열 $c[1..n]$이 주어졌을 때, T센트를 거슬러 주는 데 필요한 최소 동전 개수를 구하는 효율적인 알고리즘을 설명하고 분석하십시오. 어떤 금액 T에 대해서도 거스름돈을 만들 수 있도록 $c[1] = 1$이라고 가정하십시오.

15. 각각이 양수, 음수 또는 0일 수 있는 정수들의 배열 $A[1..n]$이 주어졌다고 가정합시다. 연속 부분 배열 $A[i..j]$의 원소들의 합이 0보다 크면 이를 ***양의 구간***이라고 합니다. A의 모든 양수 원소를 덮는 양의 구간의 최소 개수를 계산하는 알고리즘을 설명하고 분석하십시오. 예를 들어, 다음 배열이 입력으로 주어지면 알고리즘은 3을 출력해야 합니다. 입력 배열의 모든 원소가 음수이면 알고리즘은 0을 출력해야 합니다.

$$\overbrace{+3\ -5\ +7\ -4\ +1}^{\text{sum}=2}\ -8\ \overbrace{+3\ -7\ +5}^{\text{sum}=1}\ -9\ \overbrace{+5\ -2\ +4}^{\text{sum}=7}$$

16. 다음 과정을 생각해 봅시다. 여러분은 항상 하나의 양의 정수 x를 가지고 있으며, 그 값은 처음에 1입니다. 각 단계에서 x를 *1 증가*시키거나 x를 *두 배*로 만들 수 있습니다. 목표는 대상 값 n을 만들어 내는 것입니다. 예를 들어, 정수 10은 다음과 같이 네 단계 만에 만들 수 있습니다:

$$1 \xrightarrow{+1} 2 \xrightarrow{\times 2} 4 \xrightarrow{+1} 5 \xrightarrow{\times 2} 10$$

물론 정확히 $n-1$번의 증가 연산으로 임의의 정수 n을 만들 수 있지만, 거의 모든 n 값에 대해 이 방법은 지독하게 비효율적입니다. 임의로 주어진 정수 n을 만드는 데 필요한 *최소* 단계 수를 계산하는 알고리즘을 설명하고 분석하십시오.

17. n명의 스키어가 있고 그들의 키가 배열 $P[1..n]$에 주어지며, n개의 스키가 있고 그 길이가 배열 $S[1..n]$에 주어진다고 가정합시다. 각 스키어에게 스키를 하나씩 배정하되, 스키어의 키와 그 스키어에게 배정된 스키의 길이 사이의 평균 차이가 가능한 한 작아지도록 하는 효율적인 알고리즘을 설명하십시오. 이 알고리즘은 다음 식을 만족하는 순열 σ를 계산해야 합니다.

$$\frac{1}{n}\sum_{i=1}^{n}\left|P[i] - S[\sigma(i)]\right|$$

가능한 한 작습니다.

18. 앨리스는 파티를 열려고 하며, 누구를 초대할지 정하려고 합니다. 앨리스가 초대할 수 있는 사람은 n명이고, 앨리스는 이들 중 어떤 두 사람이 서로 아는 사이인지 알고 있습니다. 앨리스는 다음 두 가지 제약 조건을 지키면서 가능한 한 많은 사람을 초대하려고 합니다:

- 각 손님마다 이미 알고 있는 다른 손님이 적어도 다섯 명은 있어야 합니다.
- 각 손님에게는 이미 아는 사이가 *아닌* 다른 손님이 적어도 다섯 명은 있어야 합니다.

n명의 사람 목록과 서로 아는 사이인 사람들의 쌍 목록이 주어졌을 때, 앨리스가 초대할 수 있는 손님 수의 최댓값을 계산하는 알고리즘을 설명하고 분석하십시오.

19. 양의 정수로 이루어진 두 배열 $C[1..n]$과 $R[1..n]$이 주어졌다고 가정합시다. 0과 1로 이루어진 $n \times n$ 행렬에서 모든 인덱스 i에 대해 i번째 행에 1이 $R[i]$개 있고 i번째 열에 1이 $C[i]$개 있으면, 이 행렬은 *R과 C에 부합한다*고 합니다. R과 C에 부합하는 행렬을 구성하거나, 그러한 행렬이 존재하지 않음을 올바르게 보고하는 알고리즘을 기술하고 분석하십시오.

20. 여러분은 방금 일론 머스크에게서 샌프란시스코에서 뉴욕시까지 부리토를 배달하는 일자리를 얻었습니다. 여러분은 이 두 도시 사이를 일직선으로 잇는 일론의 새 대륙횡단 지하 부리토 배달 튜브를 지나며 부리토 배달 차량을 운전하게 됩니다.[12]

[12] . . . 그리고 이 튜브는 분명히 마치에이 체그워프스키(Maciej Cegłowski)가 지어낸 가상의 "앨러미다-위호컨 부리토 터널"을 본떠 만든 것입니다

여러분의 부리토 배달 차량은 일회용 배터리로 움직이며, 이 배터리는 최대 100마일을 달린 뒤에는 반드시 교체해야 합니다. 연료 자체는 사실상 무료이지만 배터리는 비싸고 깨지기 쉬우므로, 대륙횡단 지하 부리토 배달 차량 배터리 교체 기술자 조합의 공식 조합원만 설치할 수 있습니다.[13] 따라서 배터리를 일찍 교체하더라도 새 배터리를 설치할 때마다 정가를 모두 내야 합니다. 게다가 차량이 너무 작아서 한 번에 배터리를 두 개 이상 실을 수 없습니다.

튜브를 따라 여러 개의 충전소가 있으며, 충전소마다 새 배터리를 장착하는 요금이 서로 다릅니다. 여행을 시작하기 전에 여러분은 튜브를 따라 있는 모든 충전소의 위치와 요금이 나와 있는 위키백과 페이지를 꼼꼼히 인쇄해 둡니다. 이 정보가 주어졌을 때, 연료를 보충하기 위해 멈출 최적의 장소를 어떻게 결정하겠습니까?

좀 더 엄밀하게 말하면, 두 배열 $D[1..n]$과 $C[1..n]$이 주어진다고 가정합시다. 여기서 $D[i]$는 튜브의 시작점에서 i번째 충전소까지의 거리이고, $C[i]$는 i번째 충전소에서 배터리를 교체하는 비용입니다. 여행은 충전소에서 시작하여 충전소에서 끝나고(따라서 $D[1] = 0$이고 $D[n]$은 여행의 전체 길이입니다), 자동차는 빈 배터리로 출발한다고 가정합니다(따라서 충전소 1에서 반드시 새 배터리를 장착해야 합니다).

(a) 여행을 마치는 데 필요한 최소 급유 정차 횟수를 구하는 탐욕 알고리즘을 설명하고 분석하십시오. 여러분의 알고리즘이 올바르다는 것을 증명하는 것도 잊지 마십시오.

(b) 하지만 실제로 최소화하려는 것은 이동에 드는 *총 비용*입니다. (a)의 탐욕 알고리즘을 이 상황으로 확장하면 최적해를 만들어 내지 *못한다*는 것을 보이십시오.

(c) 총 이동 비용을 최소화하려면 어느 주유소들에 들러야 하는지, 그 위치를 계산하는 효율적인 알고리즘을 설명하십시오.

21. 여러분은 도서관 서가에 n권의 책을 순서대로 보관하는 일을 맡았습니다. 책의 순서는 도서 분류 체계에 따라 정해져 있어 바꿀 수 없으며, 각 서가에는 주어진 책 순서에서 연속된 구간을 보관해야 합니다. 두 배열 $H[1..n]$과 $T[1..n]$이 주어지는데, $H[i]$와 $T[i]$는 각각 순서상 i번째 책의 높이와 두께입니다. 이 도서관의 모든 서가는 길이가 L로 같으며, 한 서가에 놓인 모든 책의 두께 합은 L을 넘을 수 없습니다.

(a) 모든 책의 높이가 h로 같고 선반의 높이는 h보다 커서 모든 책이 어느 선반에나 들어간다고 가정합시다. 가능한 한 적은 수의 선반에 책을 꽂는 탐욕 알고리즘을 설명하고 분석하십시오. *[힌트: 알고리즘은 자명하지만, 왜 올바른 것입니까?]*

[13] 또는 이들이 독일어로 스스로를 부르는 이름으로는 Die Transkontinentaluntergrundburritolieferfahrzeugbatteriewechseltechnikervereinigung입니다.

(b) 여기까지는 좋은 준비 운동이었고, 이제 진짜 문제입니다. 실제로는 책마다 높이가 다르지만, 각 선반의 높이를 그 선반에 놓인 가장 높은 책에 맞춰 조절할 수 있습니다. (특히 비어 있는 선반의 높이는 0으로 바꿀 수 있습니다.) 이제 여러분이 할 일은 선반 높이의 합이 가능한 한 작아지도록 책을 꽂는 것입니다. (a)에서 만든 탐욕 알고리즘이 이 문제에 대해 항상 최선의 해를 내놓는 것은 *아님*을 보이십시오.

(c) 문항 (b)에서 설명한 대로 책과 선반 사이의 최적 매칭을 찾는 알고리즘을 설명하고 분석하십시오.

22. 괄호 (와)로 이루어진 문자열 w가 다음 조건 중 하나를 만족하면, w는 ***균형 잡혀*** 있다고 합니다.

- w는 빈 문자열입니다.
- 어떤 균형 문자열 x에 대해 $w =$ (x)인 경우
- 어떤 균형 잡힌 문자열 x와 y에 대해 $w = xy$입니다

예를 들어, 다음 문자열은

$$w = (((())()()) (()()) ()$$

$w = xy$이므로 균형 잡혀 있습니다. 여기서

$$x = (((())\ ()\ ())\qquad \text{그리고} \qquad y = (()\ ())\ ().$$

(a) 주어진 괄호 문자열이 균형을 이루는지 판별하는 알고리즘을 설명하고 분석하십시오.

(b) 주어진 괄호 문자열에서 가장 긴 균형 잡힌 부분 수열의 길이를 계산하는 탐욕 알고리즘을 설명하고 분석하십시오. 늘 그렇듯이, 알고리즘이 올바르다는 것을 증명하는 것도 잊지 마십시오.

두 문제 모두 입력은 배열 $w[1..n]$이며, 각 i에 대해 $w[i] =$ (이거나 $w[i] =$)입니다. 두 알고리즘 모두 $O(n)$ 시간에 실행되어야 합니다.

23. 어느 날 알렉스는 실내 암장에서 등반하는 데 싫증이 나서, 등반가 친구들을 여럿 데리고 야외로 등반하러 가기로 했습니다. 그들은 그다지 높지는 않지만 폭이 아주 넓은 거대한 바위가 있는 등반 지역으로 갔는데, 그 바위에는 손 홀드와 발 홀드가 여러 개 표시되어 있었습니다. 알렉스는 친구들이 한 홀드에서 다른 홀드로 옮겨 갈 때 쓸 수 있는 "허용된" 동작의 집합을 재빨리 정했습니다.

홀드 전체의 체계는 n개의 정점을 가진 루트 트리 T로 나타낼 수 있습니다. 여기서 각 정점은 하나의 홀드에 대응하고, 각 간선은 홀드 사이에서 허용되는 이동에 대응합니다. 등반 경로들은 볼더를 올라갈수록 한데 모여 결국 정상에 있는 유일한 홀드에 이르며, 이 홀드는 T의 루트로 나타냅니다.

알렉스와 친구들(모두 뛰어난 클라이머입니다)은 한 가지 게임을 하기로 했습니다. 이 게임에서는 가능한 한 많은 클라이머가 동시에 볼더 위에 있어야 하고, 각 클라이머는 *정확히* k번의 동작을 연속으로 수행해야 합니다. 각 클라이머는 시작할 홀드를 임의로 고를 수 있으며, 모든 동작은 지면에서 멀어지는 방향이어야 합니다. 따라서 각 클라이머는 트리 T에서 k개의 간선으로 이루어진 경로를 그리게 되며, 이 간선들은 모두 루트를 향합니다. 그러나 어떤 두 클라이머도 같은 홀드를 만질 수 없습니다. 즉, 서로 다른 클라이머가 따라가는 경로는 전혀 교차할 수 없습니다.

(a) 이 게임을 할 수 있는 등반가의 최대 수를 계산하는 탐욕 알고리즘을 설명하고 분석하십시오. 알고리즘은 루트 트리 T와 정수 k를 입력으로 받으며, T에서 각 경로의 길이가 k인 서로소 경로의 가능한 최대 개수를 계산해야 합니다. T가 이진 트리라고 가정하지 ***마십시오***. 예를 들어, 아래 트리가 입력으로 주어지면 알고리즘은 정수 8을 반환해야 합니다.

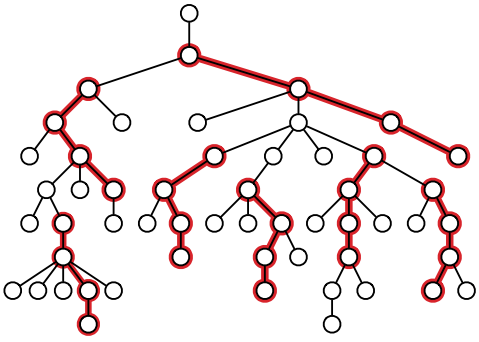


그림 4.6. 길이가 $k = 3$인 서로소 경로 일곱 개입니다. 이는 이 트리에서 이러한 경로들로 이루어진 가장 큰 집합이 *아닙니다*.

(b) 이제 T의 각 정점에 *보상*이 하나씩 주어져 있고, 목표가 경로의 총 개수가 아니라 경로에 포함된 정점들의 보상 총합을 최대화하는 것이라고 가정합시다. 여러분의 탐욕 알고리즘이 항상 최적의 보상을 반환하지는 *않는다*는 것을 보이십시오.

(c) (b)에서 설명한 대로 얻을 수 있는 최대 보상을 계산하는 효율적인 알고리즘을 설명하십시오.

24. 축하합니다! 여러분은 카멜롯을 성공적으로 정복하여, 전쟁으로 상처투성이였던 옛 세습 군주국을 아나코 생디칼리스트 공동체로 탈바꿈시켰습니다. 이 공동체에서는 시민들이 돌아가며 일종의 '이번 주의 집행관' 역할을 맡지만, 그 집행관의 모든 결정은 격주로 열리는 특별 회의에서 비준을 받아야 하는데, 순전히 내부적인 사안이라면 단순 과반수로, 그러나 더 중대한 사안이라면 3분의 2 다수로 비준합니다. . . .

마지막 상징적 행위로, 여러분은 원탁(놀랍게도 실제로 원형인 탁자입니다)을 피자 조각 같은 쐐기 모양으로 잘라 캐멀롯의 시민들에게 기념품으로 나누어 주라

고 명령합니다. 각 시민은 두 각도로 지정한 탁자의 부채꼴 조각을 달라는 요청을 제출했습니다. 예를 들어 용감한 로빈 경은 17.23°에서 42°까지의 조각을 요청할 수 있고, 순결한 랜슬롯 경은 359°에서 1°까지의 2° 조각을 요청할 수 있습니다. 각 시민은 자신이 요청한 조각을 *정확히* 받을 때에만 만족하며, 그 경우에는 반드시 만족합니다. 안타깝게도 이 범위 중 일부는 서로 겹치므로 모든 시민의 요청을 만족시키는 것은 그야말로 불가능합니다. 정치의 세계에 오신 것을 환영합니다.

충족할 수 있는 요청의 최대 개수를 구하는 알고리즘을 설명하고 분석하십시오. *[힌트: 테이블을 회전시켜도 알고리즘의 출력은 바뀌지 않아야 합니다. 각도가 정수라고 가정하지 마십시오.]*

25. 여러분이 큰 풍선 여러 개에 둘러싸인 들판에 서 있다고 가정합시다. 여러분은 현재 위치에서 움직이지 않고, 새로 산 Acme Brand Zap-O-Matic™을 사용하여 모든 풍선을 터뜨리려고 합니다. Zap-O-Matic™은 고출력 레이저 빔을 발사하며, 이 빔은 맞힌 풍선을 모두 터뜨립니다. 한 번 발사할 때마다 작은 나라 하나에 1년 동안 전력을 공급할 만큼의 에너지가 필요하므로, 여러분은 가능한 한 적은 횟수만 발사하려고 합니다.

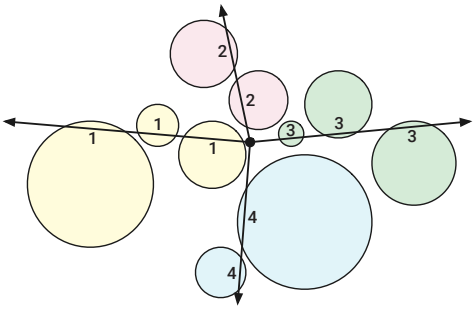


그림 4.7. Zap-O-Matic™을 네 번 쏘아 터뜨린 풍선 아홉 개

최소 잽 문제는 다음과 같이 좀 더 형식적으로 서술할 수 있습니다. 평면 위에 있는 n개의 원으로 이루어진 집합 C가 주어지며, 각 원은 반지름과 중심의 (x, y) 좌표로 지정됩니다. 이때 C에 속한 모든 원과 교차하는, 원점에서 출발하는 반직선의 최소 개수를 계산하십시오. 여러분의 목표는 이 문제를 푸는 효율적인 알고리즘을 찾는 것입니다.

(a) 어떤 풍선과도 교차하지 않는 반직선을 쏠 수 있다고 가정합시다. 이 특수한 경우에 최소 사격 문제를 해결하는 탐욕 알고리즘을 설명하고 분석하십시오. *[힌트: 연습문제 4를 참고하십시오.]*

(b) 출력값이 최적값과 최대 1만큼 차이 나는 탐욕 알고리즘을 설명하고 분석하십시오. 즉, 모든 풍선을 맞히는 데 필요한 광선의 최소 개수가 m이라면, 여러분의 탐욕 알고리즘은 m 또는 $m + 1$을 출력해야 합니다. (물론 이 사실도 증명해야 합니다.)

(c) $O(n^2)$ 시간에 최소 잽 문제를 해결하는 알고리즘을 설명하십시오.

♥(d) 최소 잽 문제를 $O(n \log n)$ 시간에 해결하는 알고리즘을 설명하십시오.

임의의 원 c와 교차하는 반직선들의 각도 범위를 $O(1)$ 시간에 알려 주는 서브루틴이 있다고 가정하십시오. 이 서브루틴은 작성하기 어렵지 않지만, 이 문제에서 흥미로운 부분은 아닙니다.

[지]식의 분류와 구분은 한 각에서 만나 단지 한 점에서만 맞닿는 여러 선과 같은 것이 아니라, 하나의 줄기에서 만나는 나무의 가지들과 같습니다. 그 줄기는 끊어져 큰 가지와 잔가지로 갈라지기 전까지 온전함과 연속성의 크기와 양을 지니고 있습니다.

— 프랜시스 베이컨, *학문의 진보* (1605)

지극히 고귀하신 귀하, 이와 같이 이러한 종류의 풀이는 수학과 거의 관련이 없음을 보실 수 있을 것입니다. 그런데도 어째서 다른 누구도 아닌 수학자가 이를 내놓기를 기대하시는지 저는 이해하지 못하겠습니다.

— 레온하르트 오일러, 카를 레온하르트 고틀리프 엘러에게 보낸 편지에서 쾨니히스베르크 다리 문제를 설명하며 (1736년 4월 3일)

그러니까, 마을 소방서 옆에서 왼쪽으로 꺾어 저수지 옆 옛 우편로를 타시고…아니, 그렇게 가면 안 되겠습니다.
포장도로를 따라 학교 건물이 나올 때까지 곧장 가시다가 베넷 호수로 가는 길에서 왼쪽으로 꺾어서…아니, 그것도 안 되겠습니다.
이스트 밀리노켓이라고 하셨습니까? 생각해 보니, 여기서는 거기로 갈 수가 없습니다.

— 로버트 브라이언, 마셜 도지, *다운 이스트에서 온 버트와 나, 그리고 다른 이야기들* (1961)

5

기본 그래프 알고리즘

5.1 소개 및 역사

그래프는 쌍들의 모음입니다—정수의 쌍, 사람의 쌍, 도시의 쌍, 별의 쌍, 국가의 쌍, 과학 논문의 쌍, 웹 페이지의 쌍, 게임 국면의 쌍, 재귀적 부분 문제의 쌍, 심지어 그래프의 쌍일 수도 있습니다. 그래프를 시각화하는 가장 일반적인 방법을 따라, 쌍을 이루는 기본 대상은 보통 ***정점*** 또는 **노드**라고 하고, 쌍 자체는 ***간선*** 또는 **호**라고 합니다. 하지만 실제로 대상과 쌍은 무엇이든 될 수 있습니다.

그래프의 가장 초기 사례 중 하나는 도로망과 그 지도입니다. 로마 제국의 전성기에 로마의 기술자들은 유럽, 서아시아와 중앙아시아, 북아프리카에 걸쳐 400 000 km가 넘는 공공 도로망을 건설했습니다. 이 도로망을 이용하는 여행자들은 *이티네라리아*(itineraria)를 지니고 다녔습니다. 이것은 여러 도로를 따라 있는 주요 지형지물과 거리를 단순한 목록으로 적거나 좀 더 그림에 가깝게 나타낸 것이었습니다. 로마의 *쿠르수스 푸블리쿠스*(cursus publicus) 전체를 묘사한 13세기 두루마리인 *타불라 페우팅게리아나*(Tabula Peutingeriana)는 아우구스투스 카이사르의 치세에 제작이 의뢰된 1세기 *이티네라리움 픽툼*(itinerarium pictum)을 5세기에 개정한 판본의 중

세 사본이라고 널리 여겨지고 있습니다. 포이팅거 지도는 지리적으로 정확한 지도가 아니라—역사학자들은 이것을 과연 "지도"라고 부를 수 있는지조차 논쟁하고 있습니다!—현대의 지하철 노선도처럼 도로망을 추상적으로 표현한 것입니다. 각 도로를 따라 있는 도시들은 그 도로를 나타내는 곡선의 꺾인 부분으로 표시되며, 이 도시들의 이름과 도시 사이 도로 구간의 길이도 지도에 표시되어 있습니다. 따라서 이 지도에는 5세기 로마 제국의 임의의 두 도시 사이의 최단 경로를 찾기에 충분한 정보가 담겨 있습니다. 그림 5.1을 참조하십시오.



그림 5.1. 콘라트 밀러(Konrad Miller)가 1872년에 복원한 *Tabula Peutingeriana*의 작은 일부입니다. 오늘날의 비르텐(*Veteribus*, 왼쪽 위)에서 쾰른(*Agripina*)과 본(*Bonnae*)을 거쳐 마인츠(*Mogontiaco*, 오른쪽 위)에 이르는 로마 가도와, 트리어(*Avg Tresvirorvm*, 가운데) 및 메스(*Matricorvm*, 가운데 아래)로 갈라지는 지선을 보여 줍니다. (책 말미의 이미지 출처를 참조하십시오.)

그래프—특히 트리—의 가장 오래된 고전적 응용 중 하나는 족보를 표현하는 것입니다. 복잡한 가계 "트리"는 수 세기 동안 결혼, 상속, 왕위 계승에 관한 법적 문제를 해결하는 데 사용되어 왔습니다. 로마 제국의 민법은 훗날 초기 가톨릭 교회에 교회법으로 채택되었으며, 사촌 또는 그보다 가까운 친척 간의 결혼을 금지하였습니다. 9세기 초에 교회는 요구되는 거리와 계산 방법을 모두 바꾸었습니다. 로마의 *computatio legalis*는 가장 가까운 공통 조상까지의 두 거리의 합이 4 이상일 것을 요구하였던 반면, 새로운 *computatio canonica*는 두 거리 중 최댓값이 7 이상일 것을 요구하였습니다. 1215년, 현실적인 고려(그리고 실제 관행)에 굴복하여 교회는 결혼에 필요한 최소 거리를 4로 완화하였습니다.[1] 그림 5.2의 왼쪽 도식은 특히 복잡하게 얽힌 사례를 보여 줍니다. 티리우스와 테부르가가 결혼하여 아들 가이우스를 낳은 후 티리우스가 사망합니다. 이어서 테부르가는 로타르와 결혼하여 그의 아들을 낳고 사망합니다. 마지막으로 로타르와 베르타가 결혼하여 딸 젬마를 낳습니다. 가이우스의 아들은 젬마의 딸과 법적으로 결혼할 수 있습니까?

1600년대 후반, 프랑스의 수학자 피에르 바리뇽(Pierre Varignon)은 그보다 한 세기 앞서 출판된 시몬 스테빈(Simon Stevin)의 연구를 바탕으로, 장력을 받는 밧줄들이 이루는 트리 형태의 네트워크의 평형 위치를 찾는 도식적 방법을 개발했습니다. 바리뇽은 밧줄들이 평형 상태에 있을 때 밧줄과 평행한 선분을 간선으로 하고 각 간선의

[1] 11세기와 12세기에 걸쳐 이 제한은 인척 관계를 통한 최대 네 단계의 연결까지 포함하도록 점차 확대되었는데, 처음에는 혼인을 통한 연결을, 나중에는 혼외 성관계, 약혼, 심지어 대부모 관계를 통한 연결까지 포함하였습니다. 예를 들어, 어떤 남자와 그의 누이의 남편의 누이의 남편의 누이 사이의 결혼은 공식적으로 금지되었으며, 홀아비와 그의 아들의 아내의 과부가 된 어머니 사이의 결혼도 마찬가지였습니다. 이러한 인척 요건은 1215년에 크게 축소되었지만 완전히 폐지되지는 않았습니다. 교회는 1917년에 이르러서야 *ex copula illicita*에 의한 인척 개념을 폐기하였습니다.

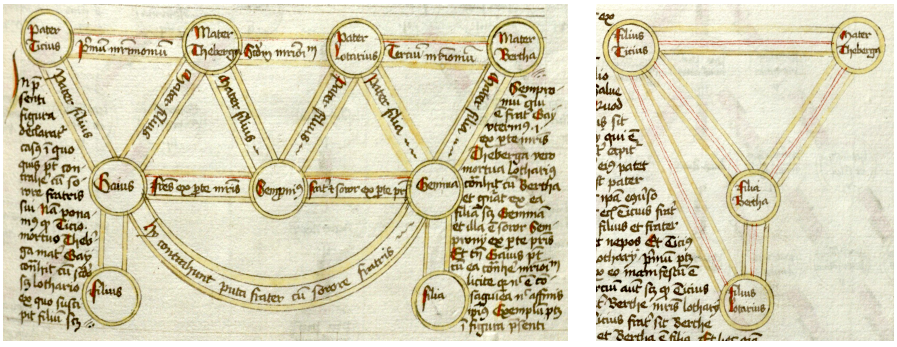


그림 5.2. 복잡한 혼인 사례를 설명하는 두 도표로, 14세기 초의 교회법 논고인 요하네스 안드레아이(Johannes Andreae)의 *Super arboribus consanguinitatis et affinitatis*에 관해 작자 미상의 15세기 논고에서 가져온 것입니다. (책 끝의 이미지 출처를 참조하십시오.)

길이가 해당 밧줄에 작용하는 힘과 같은 그래프를 그릴 수 있으며, 이때 네트워크의 어느 한 점에서 만나는 밧줄들이 그래프에서 하나의 닫힌 사이클을 이룬다는 것을 관찰했습니다. 바리뇽의 "도식 정역학" 방법은 그가 세상을 떠나고 2년이 지난 1725년에 이르러서야 완전한 형태로 출판되었습니다. 오늘날 이러한 그래프는 *상반 힘 다이어그램* 또는 *맥스웰-크레모나 다이어그램*이라고 불립니다. 이는 1800년대 후반에 (카를 쿨만(Carl Culmann) 등과 함께) 상반 다이어그램에 관한 풍부한 이론을 발전시킨 제임스 클러크 맥스웰(James Clerk Maxwell)과 루이지 크레모나(Luigi Cremona)의 이름을 딴 것입니다.

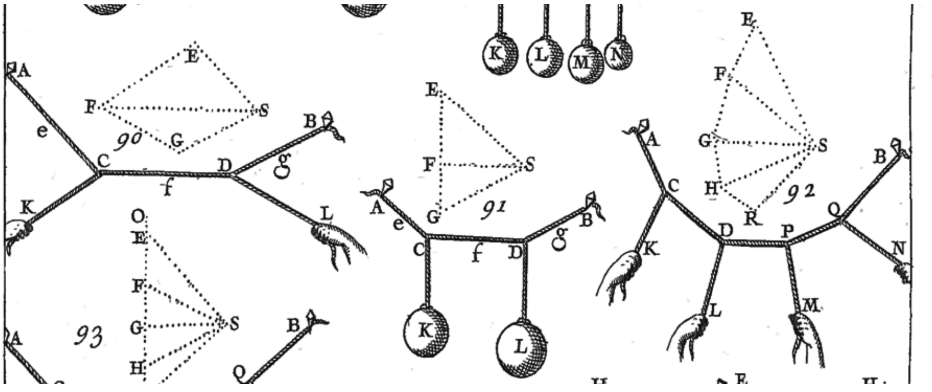


그림 5.3. 바리뇽 사후에 출간된 *Nouvelle mécanique, ou statique, dont le projet fut donné en MDCLXXXVII [1687년에 구상이 제시된 새로운 역학, 또는 정역학]*에 실린 상반 힘 도해(점선)입니다. (책 끝의 이미지 출처를 참조하십시오.)

물론 그래프의 친숙한 예는 이 밖에도 많습니다. 보드게임(고대부터 존재함), 볼록 다면체의 정점과 간선(고대 그리스 철학자들이 정식으로 연구했지만, 그 역사는 훨씬 더 오래됨), 별자리 패턴의 시각화(7세기 무렵 동아시아에서 이미 발전함), 나이트 투어(9세기와 10세기에 알아들리, 루드라타(Rudraṭa), 알술리 등이 기술함), 미로(1420년경 조반니 폰타나가 현대적인 형태로 도입함), 측지 삼각측량(1533년 젬마 프리시우

스가 도입했으며, 1615년 빌레브로르트 스넬이 지구 둘레를 계산하는 데, 1799년에는 미터를 정의하는 데 사용됨), 쾨니히스베르크 다리 퍼즐에 대한 레온하르트 오일러의 잘 알려진 부분적[2] 해법(1735년), 전신을 비롯한 통신망(1753년에 처음 제안되었고, 1800년대 초 로널즈, 실링, 가우스, 베버 등이 개발했으며, 1800년대 말에는 전 세계에 보급됨), 전기 회로(1800년대 초 옴, 맥스웰, 키르히호프 등이 정식화함), 분자 구조식(1857년 아우구스트 케쿨레와 1858년 아치볼드 쿠퍼가 각각 독립적으로 도입함), 소셜 네트워크(1930년대 중반 사회학자 제이컵 모레노가 처음 연구함), 디지털 전자 회로(1886년 찰스 샌더스 퍼스가 제안했고, 1937년 클로드 섀넌이 현대적인 형태로 정립함), 그리고 네, 좋습니다, 굳이 원하신다면 현대 인터넷도 있습니다.

추상적인 수학적 대상을 가리키는 "그래프(graph)"라는 단어는 1878년 제임스 실베스터(James Sylvester)가 만들었습니다. 그는 동료 윌리엄 클리퍼드(William Clifford)의 제안에 따라 케쿨레(Kekulé)의 "케미코그래프(chemicographs)"를 변형하여 특정한 대수적 불변식을 기술하는 데 사용했습니다. "트리(tree)"라는 단어를 연결된 비순환 그래프를 가리키는 데 처음 사용한 사람은 1857년 아서 케일리(Arthur Cayley)입니다. 다만 트리라는 추상적 개념 자체는 그보다 10년 앞서 구스타프 키르히호프(Gustav Kirchhoff)와 카를 폰 슈타우트(Karl von Staudt)가 이미 사용한 바 있습니다. 그래프 이론에 관한 0번째 책은 1926년 앙드레 생트라게(André Sainte-Laguë)가 출간했으며, 그로부터 10년 뒤 데네시 쾨니그(Dénes Kőnig)가 최초의 그래프 이론 책을 출간했습니다.

5.2 기본 정의

형식적으로, (단순) ***그래프***는 집합의 쌍 (V, E)입니다. 여기서 V는 공집합이 아닌 임의의 유한 집합이며, 그 원소를 ***정점***[3] 또는 ***노드***라고 부릅니다. E는 V의 원소들의 쌍으로 이루어진 집합이며, 그 원소를 ***간선***이라고 부릅니다. ***무방향*** 그래프에서 간선은 순서가 없는 쌍, 즉 크기가 2인 집합입니다. 저는 u와 v 사이의 무방향 간선을 나타낼 때 보통 $\{u, v\}$ 대신 $\boldsymbol{uv}$라고 씁니다. ***방향*** 그래프에서 간선은 정점들의 순서쌍입니다. 저는 u에서 v로 가는 방향 간선을 나타낼 때 보통 (u, v) 대신 $\boldsymbol{u{\to}v}$라고 씁니다.

표준적인(하지만 솔직히 혼동을 일으키는) 관례에 따라, V는 그래프의 정점 *개수*를, E는 간선의 *개수*를 나타내는 데에도 사용하겠습니다. 따라서 임의의 무방향 그래프에서는 $0 \le E \le \binom{V}{2}$이고, 임의의 방향 그래프에서는 $0 \le E \le V(V-1)$입니다.

간선 uv 또는 $u{\to}v$의 ***끝점***은 그 간선의 정점 u와 v입니다. 방향 간선 $u{\to}v$의 두 끝점은 u를 ***꼬리***, v를 ***머리***라고 불러 구별합니다.

---

[2] 오일러는 자기 논증의 마지막 단계—모든 정점의 차수가 짝수일 때 그래프의 오일러 투어를 실제로 찾는 과정—를 자명하다고 여기고 넘어갔습니다. 또한 오일러는 오일러 투어를 가지는 그래프가 반드시 연결되어 있어야 한다는 사실도 알아차리지 못했습니다. 그래프가 오일러 투어를 가질 필요충분조건이 그래프가 연결되어 있고 모든 정점의 차수가 짝수라는 것임을 처음으로 완전하게 증명한 결과는 1873년 카를 히어홀처가 발표했습니다.

[3] 영어 단어 "vertices"의 단수형은 ***vertex***입니다. 마찬가지로 "matrices"의 단수형은 ***matrix***이고, "indices"의 단수형은 ***index***입니다. 이탈리아어를 쓰는 것이 아니라면 vertice, matrice, indice, appendice, helice, apice, vortice, radice, simplice, codice, directrice, dominatrice, Unice, Kleenice, Asterice, Obelice, Dogmatice, Getafice, Cacofonice, Vitalstatistice, Geriatrice, Jimi Hendrice 같은 것은 존재하지 않습니다! 이 규칙을 기억하기 어렵다면 그냥 "node"라는 말을 쓰십시오.

그래프를 두 *집합*의 쌍으로 정의하면 양 끝점이 같은 무방향 간선이 여러 개 있거나, 머리와 꼬리가 모두 같은 방향 간선이 여러 개 있는 것이 허용되지 않습니다. (하나의 방향 그래프에 방향 간선 $u \to v$와 그 역방향 간선 $v \to u$가 함께 들어 있을 수는 있습니다.) 마찬가지로 무방향 간선을 정점들의 *집합*으로 정의하면, 한 정점에서 자기 자신으로 가는 무방향 간선도 허용되지 않습니다. 루프와 평행 간선이 없는 그래프를 흔히 ***단순*** 그래프라고 부르며, 단순하지 않은 그래프를 ***다중 그래프***라고 부르기도 합니다. 이처럼 형식적인 정의에는 빈틈이 있지만, 단순 그래프를 위한 알고리즘 대부분은 거의 또는 전혀 수정하지 않고도 다중 그래프로 확장됩니다. 그래서 여기서 형식적인 정의를 내릴 필요는 없다고 봅니다.

무방향 그래프의 임의의 간선 uv에 대해, u를 v의 ***이웃***이라고 부르며 그 역도 마찬가지입니다. 또한 u와 v가 ***인접***한다고 말합니다. 노드의 ***차수***는 그 노드의 이웃의 수입니다. 방향 그래프에서는 두 종류의 이웃을 구별합니다. 임의의 방향 간선 $u \to v$에 대해, u를 v의 ***선행자***라고 부르고, v를 u의 ***후속자***라고 부릅니다. 정점의 ***진입 차수***는 그 정점의 선행자의 수이고, ***진출 차수***는 그 정점의 후속자의 수입니다.

$V' \subseteq V$이고 $E' \subseteq E$이면 그래프 $G' = (V', E')$를 $G = (V, E)$의 ***부분 그래프***라고 합니다. G의 ***진부분 그래프***는 G 자신을 제외한 모든 부분 그래프입니다.

무방향 그래프 G에서 ***보행***은 정점들의 수열로, 수열에서 이웃한 두 정점은 모두 G에서 서로 인접합니다. 비형식적으로는 보행을 간선들의 수열로 생각할 수도 있습니다. 각 정점을 최대 한 번만 방문하는 보행을 ***경로***라고 합니다. 그래프 G의 임의의 두 정점 u와 v에 대해, G에 u와 v 사이의 보행(따라서 경로)이 존재하면 v는 u로부터 ***도달 가능***하다고 합니다. 모든 정점이 다른 모든 정점으로부터 도달 가능한 무방향 그래프를 ***연결*** 그래프라고 합니다. 모든 무방향 그래프는 하나 이상의 ***컴포넌트***로 이루어지며, 컴포넌트는 그 그래프의 극대 연결 부분 그래프입니다. 두 정점 사이에 경로가 존재하는 경우에, 그리고 오직 그 경우에만 두 정점은 같은 컴포넌트에 속합니다.[4]

보행이 같은 정점에서 시작하고 끝나면 그 보행은 ***닫혀*** 있다고 합니다. ***사이클***은 각 정점에 최대 한 번씩만 들어가고 나오는 닫힌 보행입니다. 무방향 그래프의 어떤 부분 그래프도 사이클이 아니면 그 그래프는 ***비순환*** 그래프입니다. 비순환 그래프는 ***숲***이라고도 부릅니다. ***트리***는 연결된 비순환 그래프이며, 동치로 숲의 한 연결 요소입니다. 무방향 그래프 G의 ***신장 트리***는 트리이면서 G의 모든 정점을 포함하는 부분 그래프입니다. 그래프는 연결되어 있을 때, 그리고 오직 그때에만 신장 트리를 가집니다. G의 ***신장 숲***은 G의 각 연결 요소마다 하나씩 있는 신장 트리들의 모음입니다.

방향 그래프에서는 정의가 약간 달라집니다. ***방향 보행***은 모든 인덱스 i에 대해 $v_{i-1} \to v_i$가 방향 간선인 정점들의 수열 $v_0 \to v_1 \to v_2 \to \cdots \to v_\ell$이며, 방향 경로와 방향 사이클도 비슷하게 정의됩니다. 방향 그래프 G에서 정점 v가 정점 u로부터 ***도달 가능***하다는 것은 G가 u에서 v로 가는 방향 보행(따라서 방향 경로)을 포함한다는 것과 필요충분조건입니다. 방향 그래프의 모든 정점이 다른 모든 정점으로부터 도달 가능하면, 그 방향 그래프는 ***강하게 연결되어*** 있다고 합니다. 방향 그래프가 방향 사이클을 포함하지 않으면 그 그래프는 ***비순환***이라고 하며, 방향 비순환 그래프는 흔히 ***dag***라고

[4] 컴포넌트는 흔히 "연결 컴포넌트"라고 불리지만, 이는 중복된 표현입니다. 컴포넌트는 정의상 연결되어 있기 때문입니다.

부릅니다.

5.3 표현과 예시

그래프를 시각적으로 표현하는 가장 흔한 방법은 그래프를 ***그리는*** 것입니다. 그래프의 그림은 각 정점을 평면 위의 한 점(보통 작은 원이나 다른 도형으로 그립니다)에 대응시키고, 각 간선을 두 정점을 잇는 곡선이나 직선 선분에 대응시킵니다. 어떤 두 간선도 교차하지 않는 그림이 존재하면 그 그래프를 ***평면*** 그래프라고 하며, 그러한 그림을 ***임베딩***이라고도 부릅니다.[5] 같은 그래프라도 여러 가지 다른 그림을 가질 수 있으므로, 특정한 그림을 그래프 자체와 혼동하지 않는 것이 중요합니다. 특히, 평면 그래프도 평면적이지 않은 그림을 가질 수 있습니다!

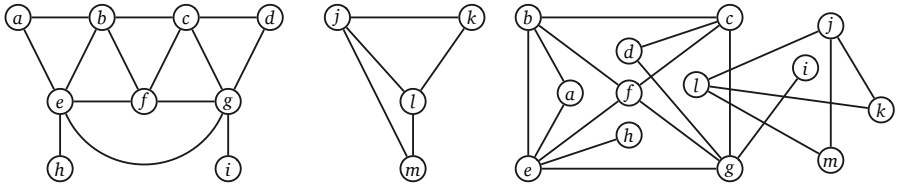


그림 5.4. 정점 13개, 간선 19개, 연결 요소 2개로 이루어진 동일한 비연결 평면 그래프를 두 가지 방식으로 그린 그림입니다. 이 중 왼쪽 그림만 임베딩입니다.

하지만 그림이 그래프를 표현하는 유용한 방법의 전부는 결코 아닙니다. 예를 들어, 기하학적 객체들의 모음에 대한 ***교차 그래프***는 각 객체마다 노드를 하나씩 가지고, 서로 교차하는 객체 쌍마다 간선을 하나씩 가집니다. 특정 그래프를 교차 그래프로 표현할 수 있는지는 정점으로 어떤 종류의 객체를 사용하려는지에 따라 달라집니다. 선분, 직사각형, 원 등 서로 다른 종류의 객체는 서로 다른 그래프 부류를 정의합니다. 특히 유용한 교차 그래프의 한 종류는 *구간 그래프*입니다. 구간 그래프의 정점은 실수 직선 위의 구간이며, 서로 겹치는 두 구간 사이에는 간선이 있습니다.

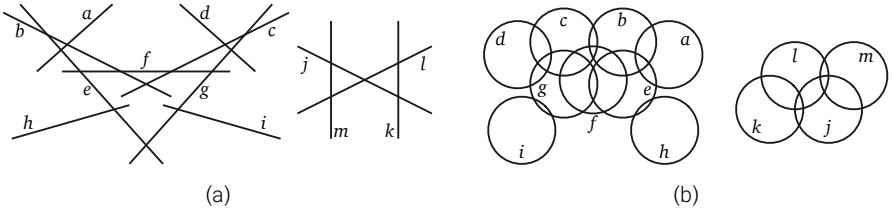


그림 5.5. 그림 5.4의 그래프는 (a) 선분 집합의 교차 그래프이면서 (b) 원 집합의 교차 그래프이기도 합니다.

또 다른 좋은 예는 재귀 알고리즘의 ***의존성 그래프***입니다. 의존성 그래프는 방향 비순환 그래프입니다. 정점은 특정 입력에 대해 알고리즘을 실행할 때 발생하는 서로 다른 모든 재귀적 부분 문제입니다. 두 번째 부분 문제를 평가하는 데 첫 번째 부분

[5] 혼란스럽게도, "임베딩"이라는 단어는 간선이 교차하는 경우에도 "그림"의 동의어로 자주 쓰입니다. 부디 그렇게 쓰지 마십시오.

문제의 재귀적 평가가 필요하면, 첫 번째 부분 문제에서 두 번째 부분 문제로 가는 간선이 있습니다. 예를 들어, 피보나치 점화식

$$F_n = \begin{cases} 0 & n = 0\text{이면,} \\ 1 & n = 1\text{이면,} \\ F_{n-1} + F_{n-2} & \text{그렇지 않으면,} \end{cases}$$

의존성 그래프의 정점은 정수 $0, 1, 2, \ldots, n$이고, 간선은 2 이상 n 이하인 모든 정수 i에 대한 쌍 $(i-1) \rightarrow i$와 $(i-2) \rightarrow i$입니다.

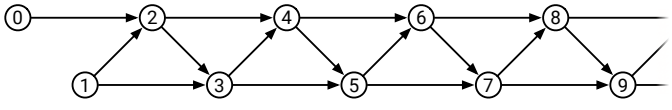


그림 5.6. Piṅgala-피보나치 점화식의 의존성 그래프입니다.

더 복잡한 예로, 3장에서 다룬 *편집 거리* 문제의 점화식을 떠올려 봅시다:

$$Edit(i,j) = \begin{cases} i & \text{만약} j = 0 \\ j & \text{만약} i = 0 \\ \min \left\{ \begin{array}{c} Edit(i-1,j) + 1 \\ Edit(i,j-1) + 1 \\ Edit(i-1,j-1) + [A[i] \neq B[j]] \end{array} \right\} & \text{그 외의 경우} \end{cases}$$

이 점화식의 의존성 그래프는 정점 (i,j)로 이루어진 $m \times n$ 격자이며, 이 정점들은 수직 간선 $(i-1,j) \rightarrow (i,j)$, 수평 간선 $(i,j-1) \rightarrow (i,j)$, 대각선 간선 $(i-1,j-1) \rightarrow (i,j)$으로 연결됩니다. 동적 계획법은 의존성 그래프가 적당히 작은 점화식이라면 어떤 것이든 효율적으로 동작합니다. 적절한 평가 순서를 따르면 각 부분 문제를 반드시 그 선행 부분 문제들을 모두 방문한 *후에* 방문하게 됩니다.

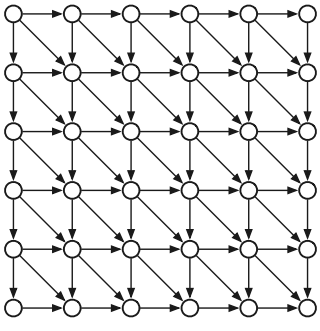


그림 5.7. 편집 거리 점화식의 의존성 그래프입니다.

또 다른 흥미로운 예로는 틱택토, 체커, 루빅스 큐브, 하노이의 탑, 튜링 기계와 같은 게임, 퍼즐 또는 메커니즘의 ***구성 그래프***가 있습니다. 구성 그래프의 정점은 그

퍼즐의 모든 유효한 구성이며, 한 구성을 단 한 번의 간단한 "수"로 다른 구성으로 바꿀 수 있으면 그 구성에서 다른 구성으로 향하는 간선이 있습니다. (물론 정확한 정의는 어떤 수가 허용되는지에 따라 달라집니다.) 꽤 단순한 메커니즘이라 하더라도 구성 그래프는 극도로 복잡할 수 있으며, 우리는 보통 구성 그래프에 대한 국소적인 정보에만 접근할 수 있습니다.

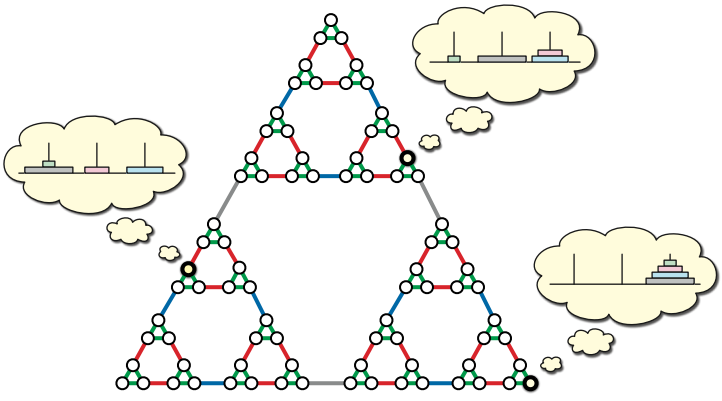


그림 5.8. 원판 4개짜리 하노이의 탑의 구성 그래프입니다.

구성 그래프는 2장에서 살펴본 *게임 트리*와 가까운 친척이지만, 결정적인 차이가 하나 있습니다. 게임의 각 상태는 구성 그래프에는 *정확히* 한 번 나타나지만, 게임 트리에는 여러 번 나타날 수 있습니다. 한마디로 구성 그래프는 *메모이제이션된* 게임 트리입니다!

형식 언어 이론에서 사용하는 ***유한 상태 오토마타***는 레이블이 붙은 방향 그래프로 모델링할 수 있습니다. 결정적 유한 상태 오토마톤은 형식적으로 5-튜플 $M = (\Sigma, Q, s, A, \delta)$로 정의된다는 것을 떠올려 봅시다. 여기서 Σ는 *알파벳*이라고 부르는 유한 집합이고, Q는 *상태*들의 유한 집합이며, $s \in Q$는 *시작 상태*이고, $A \subseteq Q$는 *수락 상태*들의 집합이며, $\delta : Q \times \Sigma \to Q$는 *전이 함수*입니다. 그러나 M을 상태 Q를 정점으로 하고, 모든 상태 $q \in Q$와 기호 $a \in \Sigma$에 대해 $q \to \delta(q, a)$ 형태의 간선을 갖는 방향 그래프 G_M으로 생각하는 편이 더 유용한 경우가 많습니다. 그러면 M이 수락하는 언어 $L(M)$에 관한 기본적인 질문들을 그래프 G_M에 관한 질문으로 표현할 수 있습니다. 예를 들어, 시작 상태/정점 s에서 도달할 수 있는 수락 상태/정점이 하나도 없을 때, 그리고 오직 그때에만 $L(M) = \varnothing$입니다.

마지막으로, 하나의 그래프를 사용하여 더 큰 다른 그래프들을 암묵적으로 표현할 수 있는 경우도 있습니다. 이러한 암묵적 표현의 좋은 예가 *부분집합 구성*입니다. 부분집합 구성은 보통 NFA를 DFA로 변환하는 데 사용되지만, 다음과 같이 *임의의* 방향 그래프에도 적용할 수 있습니다. *어떤* 방향 그래프 $G = (V, E)$가 주어지든, 새로운 방향 그래프 $G' = (2^V, E')$를 정의할 수 있습니다. 이 그래프의 정점은 V에 속한 정점들의 모든 *부분집합*이고, 간선 E'는 다음과 같이 정의됩니다:

$$E' := \{A \to B \mid u \to v \in E \text{어떤} u \in A \text{그리고} v \in B\}$$

이 정의를 기계적으로 옮기면 G로부터 G'을 구성하는 알고리즘을 만들 수 있지만, 엄밀히 말하면 이러한 구성은 필요하지 않습니다. G가 *이미* G'의 *암시적* 표현이기 때문입니다.

이러한 예시나 표현 중 어느 것도 실제 형식적 *정의*와 혼동하지 않는 것이 중요합니다. 그래프는 집합의 쌍 (V, E)이며, 여기서 V는 공집합이 아닌 *임의의* 유한 집합이고, E는 V의 원소들로 이루어진 쌍(순서쌍이든 비순서쌍이든)의 집합입니다. 요컨대 그래프는 사물들의 쌍으로 이루어진 집합입니다.

5.4 자료구조

실제로 그래프는 보통 두 가지 표준 자료구조, 즉 *인접 리스트*와 *인접 행렬* 중 하나로 표현됩니다. 높은 수준에서 보면 두 자료구조 모두 정점으로 인덱싱되는 배열입니다. 따라서 각 정점은 1과 V 사이의 고유한 정수 식별자를 가져야 합니다. 형식적인 의미에서는 이 정수들이 곧 정점입니다.

인접 리스트

그래프를 저장하는 데 단연 가장 흔히 쓰이는 자료구조는 ***인접 리스트***입니다. 인접 리스트는 리스트의 배열로, 각 리스트에는 한 정점의 이웃들(그래프가 방향 그래프라면 나가는 이웃들)이 담겨 있습니다.[6] 무방향 그래프에서는 각 간선 uv가 두 번, 즉 u의 이웃 리스트에 한 번, v의 이웃 리스트에 한 번 저장됩니다. 방향 그래프에서는 각 간선 $u \to v$가 꼬리 u의 이웃 리스트에 한 번만 저장됩니다. 두 종류의 그래프 모두에서 인접 리스트에 필요한 전체 공간은 $O(V + E)$입니다.

이러한 이웃 리스트를 표현하는 방법은 여러 가지가 있지만, 표준 구현은 단순한 단일 연결 리스트를 사용합니다. 이렇게 얻은 자료구조를 사용하면 노드 v의 (나가는) 이웃을 $O(1 + \deg(v))$ 시간에 나열할 수 있습니다. v의 이웃 리스트를 훑기만 하면 됩니다. 마찬가지로, u의 이웃 리스트를 훑어서 $u \to v$가 간선인지 $O(1 + \deg(u))$ 시간에 판별할 수 있습니다. 무방향 그래프의 경우에는 u와 v 양쪽의 이웃 리스트를 동시에 훑다가 간선을 찾거나 어느 한 리스트의 끝에 도달하면 멈추는 방식으로 시간을 $O(1 + \min\{\deg(u), \deg(v)\})$로 개선할 수 있습니다.

물론 우리가 사용할 수 있는 자료구조가 연결 리스트만 있는 것은 아닙니다. 탐색, 나열, 삽입, 삭제를 지원하는 구조라면 무엇이든 괜찮습니다. 예를 들어 u의 이웃을 균형 이진 탐색 트리에 저장하면 uv가 간선인지 판단하는 시간을 $O(1 + \log(\deg(u)))$로 줄일 수 있고, 적절하게 구성한 해시 테이블을 사용하면 $O(1)$ 시간까지도 줄일 수 있습니다.[7]

인접 리스트를 구현하는 흔한 방식 중 하나는 ***인접 배열***입니다. 인접 배열은 하나의 배열에 모든 간선 레코드를 저장하되, 각 정점에 연결된 간선들의 레코드를 연속된

[6] 주의 깊은 학생이라면 그 이름에도 불구하고 인접 리스트가 리스트가 아니라는 점을 알아챘을 것입니다. 이 명명법은 붉은 청어 원칙의 한 예입니다. 수학에서와 마찬가지로 컴퓨터 과학에서도 붉은 청어(red herring)는 반드시 붉지도 않고 반드시 물고기도 아닙니다.

[7] 이는 들리는 것보다 *훨씬* 미묘한 문제입니다. 널리 쓰이는 해싱 기법 대부분은 빠른 질의 시간을 보장하지 *않으며*, *좋은* 해싱 방법조차 대부분 $O(1)$ *기대* 시간만 보장할 수 있습니다. 해싱에 대한 더 자세한 논의는 http://algorithms.wtf를 참고하십시오.

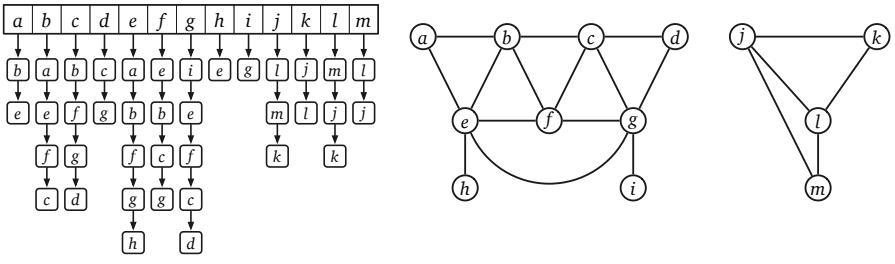


그림 5.9. 예제 그래프의 인접 리스트입니다.

구간에 배치하고, 별도의 배열에는 각 정점에 연결된 첫 번째 간선의 인덱스를 저장합니다. 또한 그림 5.10에 나타낸 것처럼 각 정점의 구간을 정렬된 상태로 유지하면 유용합니다. 그러면 두 정점 u와 v가 인접한지를 $O(\log \deg(u))$ 시간에 확인할 수 있습니다.

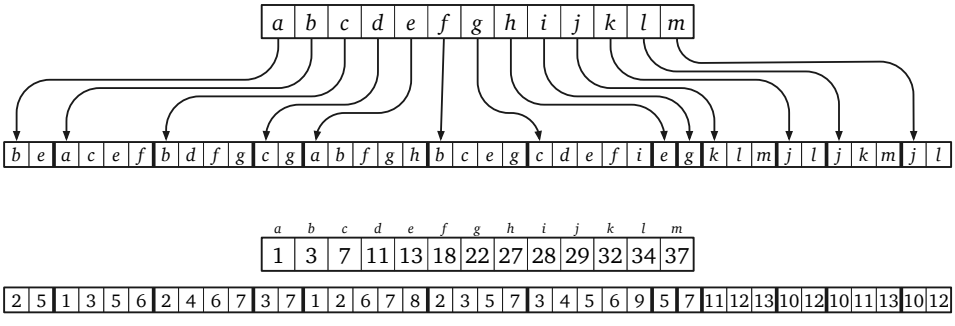


그림 5.10. 예제 그래프의 추상적인 인접 배열과, 이를 정수 배열 한 쌍으로 실제 구현한 모습입니다.

인접 행렬

그래프를 위한 또 다른 표준 자료구조는 ***인접 행렬***[8]로, 1894년 Georges Brunel이 처음 제안했습니다. 그래프 G의 인접 행렬은 0과 1로 이루어진 $V \times V$ 행렬이며, 보통 2차원 배열 $A[1..V, 1..V]$로 표현합니다. 이 행렬의 각 항목은 특정 간선이 G에 존재하는지 여부를 나타냅니다. 구체적으로, 모든 정점 u와 v에 대해 다음이 성립합니다:

- 그래프가 무방향 그래프이면, $uv \in E$일 때 그리고 오직 그때에만 $A[u, v] := 1$이며,
- 그래프가 방향 그래프이면, $u \to v \in E$일 때 그리고 오직 그때에만 $A[u, v] := 1$입니다.

무방향 그래프의 인접 행렬은 항상 *대칭*입니다. 즉, 모든 정점 u와 v에 대해 $A[u, v] = A[v, u]$가 성립합니다. uv와 vu는 같은 간선을 부르는 서로 다른 이름일 뿐이기 때문입니다. 또한 대각 성분 $A[u, u]$는 모두 0입니다. 방향 그래프의 인접 행렬은 대칭일 수도 있고 아닐 수도 있으며, 대각 성분도 0일 수도 있고 아닐 수도 있습니다.

[8] 각주 3을 참조하십시오.

	a	b	c	d	e	f	g	h	i	j	k	l	m
a	0	1	0	0	1	0	0	0	0	0	0	0	0
b	1	0	1	0	1	1	0	0	0	0	0	0	0
c	0	1	0	1	0	1	1	0	0	0	0	0	0
d	0	0	1	0	0	0	1	0	0	0	0	0	0
e	1	1	0	0	0	1	1	1	0	0	0	0	0
f	0	1	1	0	1	0	1	0	0	0	0	0	0
g	0	0	1	1	1	1	0	0	1	0	0	0	0
h	0	0	0	0	1	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	1	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0	1	1	1
k	0	0	0	0	0	0	0	0	0	1	0	1	0
l	0	0	0	0	0	0	0	0	0	1	1	0	1
m	0	0	0	0	0	0	0	0	0	1	0	1	0

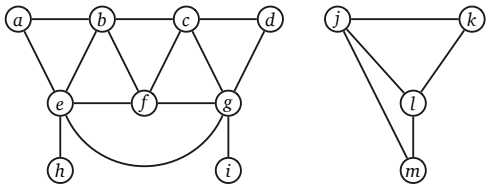


그림 5.11. 예제 그래프의 인접 행렬입니다.

인접 행렬이 주어지면, 행렬의 해당 칸을 확인하는 것만으로 두 정점이 간선으로 연결되어 있는지를 $\Theta(1)$ 시간에 판별할 수 있습니다. 또한 해당하는 행(또는 열)을 훑어서 한 정점의 모든 이웃을 $\Theta(V)$ 시간에 나열할 수 있습니다. 이 실행 시간은 최악의 경우에는 최적이지만, 어떤 정점의 이웃이 몇 개뿐이더라도 그 이웃을 모두 찾으려면 여전히 행 전체를 훑어야 합니다. 마찬가지로, 인접 행렬은 그래프에 실제로 간선이 몇 개 있는지와 관계없이 $\Theta(V^2)$ 공간을 필요로 하므로, 매우 *조밀한* 그래프에서만 공간 효율적입니다.

비교

표 5.1에 여러 표준 그래프 자료구조의 성능을 요약했습니다. 별표*는 동적 해시 테이블을 유지하는 데 드는 기대 분할 상환 시간 한계를 나타냅니다.[9]

	표준 인접 리스트 (연결 리스트)	빠른 인접 리스트 (해시 테이블)	인접성 행렬
공간	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
$uv \in E$인지 검사합니다	$O(1+\min\{\deg(u),\deg(v)\}) = O(V)$	$O(1)$	$O(1)$
$u \to v \in E$인지 검사합니다	$O(1+\deg(u)) = O(V)$	$O(1)$	$O(1)$
v의 (진출) 이웃 정점을 나열합니다	$\Theta(1+\deg(v)) = O(V)$	$\Theta(1+\deg(v)) = O(V)$	$\Theta(V)$
모든 간선을 나열합니다	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
간선 uv를 삽입합니다	$O(1)$	$O(1)^*$	$O(1)$
간선 uv를 삭제합니다	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)^*$	$O(1)$

표 5.1. 표준 그래프 자료구조에서 기본 연산에 걸리는 시간입니다.

이러한 비교에 비추어 보면, 도대체 누가 인접 행렬을 *굳이* 사용하는지 의문이 드는 것도 무리가 아닙니다. 어쨌든 해시 테이블을 사용하는 인접 리스트는 같은 연산을 같은 시간에 지원하면서도 공간은 더 적게 사용하기 때문입니다. 주된 이유는 충분히 조밀한 그래프에서는 인접 행렬이 실제로 더 단순하고 효율적이라는 점입니다. 인접

[9] "기대 분할 상환"이라는 표현을 이해하지 못하더라도 걱정하지 마십시오.

행렬은 포인터를 따라가거나 해시 함수를 계산하는 오버헤드가 없으며, 그저 연속된 메모리 블록일 뿐이기 때문입니다.

마찬가지로, 인접 리스트 구조에서 이웃을 저장할 때 균형 이진 탐색 트리나 해시 테이블 대신 굳이 *연결 리스트*를 사용하는 이유는 무엇이겠습니까? 실제로 가장 큰 이유는 거의 틀림없이 *전통*이겠지만—도널드 커누스의 코드에 충분했다면 여러분의 코드에도 충분할 것입니다!—더 원칙적인 논거도 있습니다. 그중 하나는 표준 인접 리스트가 사실 대부분의 응용에서 *충분히 좋다*는 것입니다. 대부분의 표준 그래프 알고리즘은 임의의 간선이 있는지 없는지를 실제로 묻거나 간선을 삽입 또는 삭제하려는 일이 전혀 없으므로(또는 드물므로), 그러한 연산을 지원하도록 자료구조를 최적화할 필요가 없습니다.

하지만 제 생각에 두 표준 자료구조를 모두 알아야 하는 가장 설득력 있는 이유는, 많은 그래프가 인접 행렬과 표준 인접 리스트로 *암묵적으로* 표현된다는 점입니다. 예를 들면 다음과 같습니다.

- 교차 그래프는 보통 그 바탕이 되는 기하학적 객체들의 목록으로 표현됩니다. 두 객체가 교차하는지를 상수 시간에 판정할 수 있기만 하면, 입력 그래프가 인접 행렬로 명시적으로 저장되어 있다고 *가정함*으로써 어떤 그래프 알고리즘이든 교차 그래프에 적용할 수 있습니다.

- 레코드들로 구성되고 레코드 사이가 포인터로 연결된 자료구조는 모두 방향 그래프로 볼 수 있습니다. 그래프가 표준 인접 리스트에 저장되어 있다고 *가정하면* 이러한 자료구조에도 그래프 알고리즘을 적용할 수 있습니다.

- 마찬가지로, 주어진 구성에서 가능한 모든 이동을 각각 상수 시간에 열거할 수 있다면, 구성 그래프가 *마치* 표준 인접 리스트로 표현되어 있는 것처럼 여기고 구성 그래프에 어떤 그래프 알고리즘이든 적용할 수 있습니다.

마지막 두 예에서는 임의의 정점에서 나가는 간선들을 그 차수에 비례하는 시간에 열거할 수 있지만, 두 정점이 인접한지를 상수 시간에 판별할 수 있다고는 *장담할 수 없습니다*. (이 레코드에서 저 레코드로 가는 포인터가 있습니까? 이 구성에서 저 구성으로 한 번의 수만에 갈 수 있습니까?) 게다가 각 레코드의 포인터들이나 주어진 구성에서 나가는 수들을 더 효율적인 자료구조로 재조직할 여유가 대개 없습니다. 따라서 이웃들을 연결 리스트에 저장하는 표준 인접 리스트가 적절한 모델 자료구조입니다.

이 책의 나머지 부분에서는 달리 명시하지 않는 한, 그래프 알고리즘의 모든 시간 한계는 입력 그래프가 표준 인접 리스트로 표현된다고 가정합니다. 마찬가지로 달리 명시하지 않는 한, 연습 문제에서 그래프 알고리즘을 설계하고 분석하라고 요구할 때에는 입력 그래프가 표준 인접 리스트로 표현되어 있다고 가정해야 합니다.

5.5 아무거나 우선 탐색

지금까지는 그래프에 대한 *국소적* 연산만 다루었습니다. 그래프에 대해 물을 수 있는 가장 근본적인 *전역적* 질문은 아마도 ***도달 가능성***일 것입니다. 그래프 G와 G의 정점 s가 주어졌을 때, 도달 가능성 질문은 s에서 어떤 정점들에 도달할 수 있는지를 묻습니다. 즉, 어떤 정점 v에 대해 s에서 v로 가는 경로가 존재합니까? 우선은 무방향 그래프만

고려합시다. 방향 그래프는 이 절의 끝부분에서 간단히 다루겠습니다. 무방향 그래프에서 s로부터 도달 가능한 정점들은 정확히 s와 같은 연결 요소에 속한 정점들입니다.

아마도 가장 자연스러운 도달 가능성 알고리즘은—적어도 우리처럼 재귀적으로 사고하는 데 익숙한 사람들에게는—***깊이 우선 탐색***일 것입니다. 이 알고리즘은 재귀적으로 작성할 수도 있고 반복적으로 작성할 수도 있습니다. 어느 쪽이든 완전히 같은 알고리즘이며, 유일한 차이는 비재귀 버전에서는 "재귀" 스택을 실제로 볼 수 있다는 점입니다.

```
RecursiveDFS(v):
  if v is unmarked
      mark v
      for each edge vw
          RecursiveDFS(w)
```

```
IterativeDFS(s):
  Push(s)
  while the stack is not empty
      v ← Pop
      if v is unmarked
          mark v
          for each edge vw
              Push(w)
```

깊이 우선 탐색은 제가 ***아무거나 우선 탐색***이라고 부르는 일반적인 그래프 순회 알고리즘 계열의 한 종류(아마도 가장 흔한 종류)일 뿐입니다. 일반적인 순회 알고리즘은 후보 간선들의 집합을 어떤 자료구조에 저장하는데, 저는 이 자료구조를 "가방"이라고 부르겠습니다. "가방"에서 중요한 성질은 무언가를 넣을 수 있고 나중에 다시 꺼낼 수 있다는 것뿐입니다. 스택은 가방의 한 특정한 유형이지만, 결코 유일한 유형은 아닙니다. 일반적인 알고리즘은 다음과 같습니다:

```
WhateverFirstSearch(s):
  put s into the bag
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          for each edge vw
              put w into the bag
```

저는 WhateverFirstSearch가 s에서 도달할 수 있는 모든 노드를 표시하고 그 밖의 것은 표시하지 않는다고 주장합니다. 이 알고리즘이 G의 각 정점을 *많아야* 한 번 표시한다는 것은 명백합니다. 연결 그래프의 모든 노드를 *적어도* 한 번 방문한다는 것을 보이기 위해 알고리즘을 약간 수정합니다. 수정된 부분은 굵은 빨간 글씨로 표시되어 있습니다. 수정된 알고리즘은 가방에 정점을 넣어 두는 대신 정점의 쌍을 저장합니다. 이렇게 수정하면 정점 v를 처음 방문할 때마다 이전에 방문한 이웃 정점 중 어느 것이 v를 가방에 넣었는지 기억할 수 있습니다. 이 앞선 정점을 v의 *부모*라고 부릅니다.

```
WhateverFirstSearch(s):
  put (∅, s) in bag
  while the bag is not empty
      take (p, v) from the bag                (⋆)
      if v is unmarked
          mark v
          parent(v) ← p
          for each edge vw                    (†)
              put (v, w) into the bag         (⋆⋆)
```

보조정리 5.1. *WhateverFirstSearch(s)는 s에서 도달 가능한 모든 정점을 표시하며, 오직 그 정점들만 표시합니다. 또한 parent(v) ≠ ∅인 정점과 그 부모의 쌍 (v, parent(v)) 전체의 집합은 s를 포함하는 요소의 신장 트리를 정의합니다.*

증명: 먼저 s에서 v까지의 최단 경로 거리에 대한 귀납법으로, 알고리즘이 s에서 도달 가능한 모든 정점 v를 표시함을 논증합니다. 알고리즘은 s를 표시합니다. v를 s에서 도달 가능한 다른 임의의 정점이라 하고, $s \to \cdots \to u \to v$를 s에서 v까지 간선 수가 최소인 임의의 경로라고 합시다. (v는 s에서 도달 가능하므로 그러한 경로가 반드시 존재합니다.) 접두사 경로 $s \to \cdots \to u$는 s에서 u까지의 최단 경로보다 짧으므로, 귀납 가정에 의해 알고리즘은 u를 표시합니다. 알고리즘은 u를 표시할 때 즉시 쌍 (u, v)를 가방에 넣어야 하므로, 나중에 반드시 (u, v)를 가방에서 꺼내게 되며, 그 시점에 알고리즘은 v가 이미 표시되어 있지 않다면 즉시 v를 표시합니다.

$parent(v) \neq \emptyset$인 모든 쌍 $(v, parent(v))$는 실제로 기저 그래프 G의 간선입니다. 표시된 임의의 정점 v에 대해 부모 간선들로 이루어진 경로 $v \to parent(v) \to parent(parent(v)) \to \cdots$가 결국 s로 되돌아간다고 주장하며, 이 주장을 정점이 표시되는 순서에 대한 귀납법으로 증명합니다. 자명하게 s는 s에서 도달 가능하므로, v를 그 밖의 임의의 표시된 정점이라고 합시다. v의 부모는 v가 표시되기 전에 표시되어야 하므로, 귀납 가정에 의해 부모 경로 $parent(v) \to parent(parent(v)) \to \cdots$는 s로 이어집니다. 여기에 부모 간선 $s \to parent(s)$ 하나를 더하면 주장이 성립합니다.

앞의 주장에 따르면 알고리즘이 표시한 모든 정점은 s에서 도달 가능하며, 모든 부모 간선의 집합은 연결 그래프를 이룹니다. s를 제외한 표시된 모든 노드는 부모가 유일하므로, 부모 간선의 개수는 표시된 정점의 개수보다 정확히 하나 적습니다. 따라서 부모 간선들은 트리를 이룹니다. □

분석

순회 알고리즘의 실행 시간은 "가방"에 어떤 자료구조를 사용하느냐에 따라 달라지지만, 몇 가지 일반적인 관찰을 할 수 있습니다. 가방에 항목 하나를 넣거나 가방에서 항목 하나를 꺼내는 데 필요한 시간을 T라고 합시다. for 루프 (†)는 표시된 각 정점마다 정확히 한 번씩 실행되므로 많아야 V번 실행됩니다. s의 연결 요소에 있는 각 간선 uv는 가방에 정확히 두 번 들어갑니다. 한 번은 쌍 (u, v)로, 또 한 번은 쌍 (v, u)로 들어가므로 (⋆⋆) 줄은 많아야 $2E$번 실행됩니다. 마지막으로, 가방에서는 넣은 것보다 더 많은 것을 꺼낼 수 없으므로 (⋆) 줄은 많아야 $2E + 1$번 실행됩니다. 따라서 기반 그래프 G가 표준 인접 리스트로 저장되어 있다고 가정하면,

WhateverFirstSearch는 $O(V + ET)$ *시간*에 실행됩니다. (G가 인접 행렬로 저장되어 있다면 WhateverFirstSearch의 실행 시간은 $O(V^2 + ET)$로 늘어납니다.)

5.6 주요 변형

스택: 깊이 우선

"가방"을 *스택*으로 구현하면 원래의 깊이 우선 탐색 알고리즘을 다시 얻게 됩니다. 스택은 삽입(push)과 삭제(pop)를 각각 $O(1)$ 시간에 지원하므로, 이 알고리즘은 $O(V+E)$ ***시간***에 실행됩니다. 부모 간선들이 이루는 신장 트리를 ***깊이 우선 신장 트리***라고 부릅니다. 트리의 정확한 모양은 시작 정점과 for 루프 (†) 안에서 이웃을 방문하는 순서에 따라 달라지지만, 일반적으로 깊이 우선 신장 트리는 길고 가느다랗습니다. 깊이 우선 탐색의 몇 가지 중요한 성질과 응용은 6장에서 살펴보겠습니다.

큐: 너비 우선

"가방"을 *큐*로 구현하면 ***너비 우선 탐색***이라는 또 다른 그래프 순회 알고리즘을 얻습니다. 큐는 삽입(push)과 삭제(pull)를 각각 $O(1)$ 시간에 지원하므로, 이 알고리즘은 $O(V + E)$ ***시간***에 실행됩니다. 이 경우 부모 간선들이 이루는 ***너비 우선 신장 트리***는 시작 정점 s에서 같은 연결 요소에 속한 다른 모든 정점까지의 ***최단 경로***를 포함합니다. 최단 경로는 8장에서 자세히 살펴보겠습니다. 이번에도 너비 우선 신장 트리의 정확한 모양은 시작 정점과 for 루프 (†)에서 이웃을 방문하는 순서에 따라 달라지지만, 일반적으로 너비 우선 신장 트리는 높이가 낮고 가지가 무성합니다.

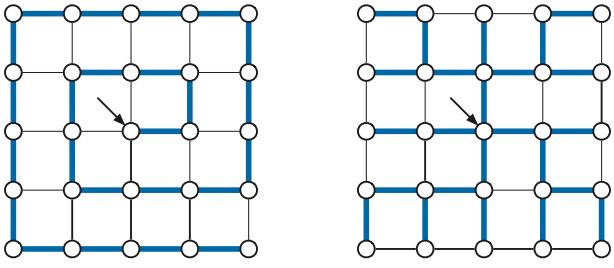


그림 5.12. 같은 그래프의 깊이 우선 신장 트리와 너비 우선 신장 트리로, 둘 다 중심 정점에서 시작합니다.

우선순위 큐: 최선 우선

마지막으로, "가방"을 *우선순위 큐*로 구현하면 ***최선 우선 탐색***(best-first search)이라는 또 다른 알고리즘 계열을 얻습니다. 우선순위 큐는 각 간선의 사본을 최대 하나만 저장하므로 간선을 삽입하거나 우선순위 값이 가장 작은 간선을 추출하는 데 $O(\log E)$ 시간이 걸리며, 이는 최선 우선 탐색이 $O(V + E \log E)$ ***시간***에 실행된다는 것을 의미합니다.

저는 최선 우선 탐색을 단일 알고리즘이 아닌 "알고리즘 계열"로 설명합니다. 간선에 우선순위를 부여하는 방법이 여러 가지 있고, 어떤 방법을 선택하느냐에 따라

알고리즘의 동작이 달라지기 때문입니다. 아래에서 잘 알려진 세 가지 변형을 설명하겠지만, 이 밖에도 많은 변형이 있습니다. 세 예시 모두에서 입력 그래프의 모든 간선 uv 또는 $u \to v$가 음이 아닌 가중치 $w(uv)$ 또는 $w(u \to v)$를 가진다고 가정합니다.

첫째, 입력 그래프가 무방향 그래프이고 각 간선의 가중치를 우선순위로 사용하면, 최선 우선 탐색은 s의 연결 요소의 ***최소 신장 트리***를 구성합니다. 놀랍게도, 모든 간선 가중치가 서로 다르기만 하면 결과로 얻는 트리는 시작 정점이나 이웃을 방문하는 순서에 의존하지 *않습니다*. 이 경우 최소 신장 트리는 실제로 유일합니다. 최선 우선 탐색의 이러한 구체화는 흔히 (그러나 늘 그렇듯이 잘못) *Prim 알고리즘*이라고 알려져 있습니다. 이 알고리즘과 다른 최소 신장 트리 알고리즘은 7장에서 더 자세히 다룹니다.

경로의 *길이*를 그 경로에 속한 간선들의 가중치의 합으로 정의합니다. 최선 우선 탐색을 사용하면 가중 그래프에서 ***최단 경로***도 다음과 같이 계산할 수 있습니다. 표시된 모든 정점 v는 거리 $dist(v)$를 저장합니다. 처음에는 $dist(s) = 0$으로 설정합니다. 다른 모든 정점 v에 대해, 부모를 $parent(v) \leftarrow p$로 설정할 때 $dist(v) \leftarrow dist(p) + w(p \to v)$도 함께 설정하고, 간선 $v \to w$를 우선순위 큐에 삽입할 때에는 우선순위 $dist(v) + w(v \to w)$를 사용합니다. 모든 간선 가중치가 양수라고 가정하면, $dist(v)$는 s에서 v까지의 최단 경로의 길이입니다. 최선 우선 탐색의 이러한 구체화는 흔히 *데이크스트라 알고리즘*이라고 알려져 있습니다(하지만 늘 그렇듯이 엄밀히 말하면 잘못된 이름입니다). 이 알고리즘은 8장에서 다시 살펴보겠습니다.

마지막으로, 경로의 *너비*를 경로에 있는 간선 가중치의 *최솟값*으로 정의합시다. "데이크스트라" 최선 우선 탐색 알고리즘을 간단히 수정하면 s에서 도달 가능한 다른 모든 정점까지의 ***가장 넓은 경로***를 계산할 수 있습니다. 가장 넓은 경로는 ***병목 최단 경로***라고도 합니다. 표시된 모든 정점 v는 값 $width(v)$를 저장합니다. 처음에는 $width(s) = \infty$로 설정합니다. 다른 모든 정점 v에 대해, 부모를 $parent(v) \leftarrow p$로 설정할 때 $width(v) \leftarrow \min\{width(p), w(p \to v)\}$도 함께 설정합니다. 또한 간선 $v \to w$를 우선순위 큐에 삽입할 때는 우선순위로 $\min\{width(v), w(v \to w)\}$를 사용합니다. 가장 넓은 경로는 *최대 흐름*을 계산하는 알고리즘에서 유용하며, (짐작하셨겠지만) 최대 흐름은 10장에서 다룹니다.

연결되지 않은 그래프

WhateverFirstSearch(s)는 시작 정점 s 하나에서 도달할 수 있는 정점만 방문합니다. G의 모든 정점을 방문하려면 다음과 같은 간단한 "래퍼(wrapper)" 함수를 사용하면 됩니다.

```
WFSAll(G):
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          WhateverFirstSearch(v)
```

잠깐, 왜 이렇게 복잡하게 만드느냐고 묻는 소리가 들립니다. 그냥[10] 정점 배열을

[10] 이 단어는 거의 언제나 여러분이 중요한 무언가를 놓치고 있다는 신호입니다.

훑어보면 되지 않습니까?

```
MarkEveryVertexDuh(G):
  for all vertices v
    mark v
```

물론, 정점의 완전한 목록이 *있다면* 그렇게 할 수 있지만, 모든 그래프가 그렇게 명시적으로 표현되지는 않는다는 점을 기억하십시오.[11] 더 중요한 점은, 명시적인 정점 목록이 있더라도 이 단순한 알고리즘이 정점을 방문하는 순서는 그래프의 추상적인 구조가 아니라 *자료구조* 안에서 정점들이 놓인 순서에 따라 결정된다는 것입니다.

특히 정점을 단순히 차례로 훑는 방식과 달리, WFSAll은 한 성분의 모든 정점을 방문한 다음 그다음 성분의 모든 정점을 방문하는 식으로 입력 그래프의 성분을 하나씩 차례로 거쳐 갑니다. 이처럼 성분 단위로 순회하므로, 예를 들어 연결되지 않은 그래프의 성분 개수를 카운터 하나만으로 셀 수 있습니다.

```
CountComponents(G):
  count ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          count ← count + 1
          WhateverFirstSearch(v)
  return count
```

조금만 더 작업하면 각 정점을 단순히 표시하는 데 그치지 않고, 그 정점이 어느 성분에 속하는지 기록할 수 있습니다.

```
CountAndLabel(G):
  count ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          count ← count + 1
          LabelOne(v, count)
  return count
```

```
《성분 하나에 레이블 붙이기》
LabelOne(v, count):
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          comp(v) ← count
          for each edge vw
              put w into the bag
```

WFSAll은 모든 정점을 한 번씩 표시하고, 모든 간선을 가방에 한 번씩 넣으며, 모든 간선을 가방에서 한 번씩 꺼내므로 전체 실행 시간은 $O(V + ET)$입니다. 여기서 T는 가방 연산 한 번에 걸리는 시간입니다. 특히 모든 정점에서 깊이 우선 탐색이나 너비 우선 탐색을 실행하더라도, 그 결과로 얻는 알고리즘에는 여전히 $O(V + E)$ 시간만 필요합니다.

[11] 한편, 각 정점이 마지막으로 "표시된" 시각을 나타내는 타임스탬프를 모든 정점에 저장해 둔다면, 순회의 시작 시각을 기록해 두고 타임스탬프가 기록된 시작 시각보다 나중인 정점만 "표시된" 것으로 간주함으로써 $O(1)$ 시간에 "모든 정점의 표시를 해제"할 수 있습니다.

또한 WhateverFirstSearch는 한 연결 요소의 신장 트리를 계산하므로, WFSAll을 사용하면 그래프 전체의 신장 숲을 계산할 수 있습니다. 특히 간선 가중치를 우선순위로 사용하는 최선 우선 탐색은 $O(V + E \log E)$ 시간에 최소 가중치 신장 숲을 계산합니다.

놀랍게도 *대단히* 널리 쓰이는 알고리즘 교재 중 적어도 하나는 이 래퍼를 깊이 우선 탐색에만 사용할 수 있다고 주장합니다.[12] 이 주장은 명백히 틀렸습니다. 사실 너비 우선 탐색의 *최초* 구현은 1945년 무렵 콘라트 추제(Konrad Zuse)가 자신의 원시 언어인 *Plankalkül*로 작성한 것으로, 바로 무방향 그래프의 연결 성분을 세고 레이블을 붙이려는 목적으로 개발되었습니다.

방향 그래프

아무거나 우선 탐색은 방향 그래프에 쉽게 적용할 수 있습니다. 유일한 차이점은 정점을 표시할 때 그 정점의 모든 *나가는* 이웃을 가방에 넣는다는 것입니다. 사실 표준 인접 리스트나 인접 행렬을 사용한다면 코드를 전혀 바꿀 필요가 없습니다!

```
WhateverFirstSearch(s):
  put s into the bag
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          for each edge v→w
              put w into the bag
```

앞의 증명에 따르면 이 알고리즘은 s에서 도달 가능한 모든 정점을 표시하며, 방향 간선 $parent(v) \to p$는 모든 간선이 루트 s에서 멀어지는 방향을 향하는 루트 트리를 정의합니다. 이 간선들은 각 정점을 그 부모와 잇습니다. 그러나 그래프가 연결되어 있더라도 이제는 도달 가능성이 더 이상 대칭적이지 않기 때문에, 그래프의 *신장* 트리를 반드시 얻는다고는 할 수 없습니다.

그런가 하면 또 다른 한편으로, WhateverFirstSearch는 s에서 도달 가능한 정점들의 신장 트리를 실제로 정의합니다. 게다가 "가방"을 어떻게 구체화하느냐에 따라, 도달 가능한 이 정점들의 깊이 우선 신장 트리, 너비 우선 신장 트리, 최소 가중치 방향 신장 트리, 최단 경로 트리, 또는 최대 폭 경로 트리를 얻을 수 있습니다.

5.7 그래프 환원: 플러드 필(Flood Fill)

아무거나 우선 탐색의 가장 초기 현대적 사례 중 하나는 1950년대 중반 Edward Moore가 제안했습니다. *픽셀 맵*은 값이 색을 나타내는 2차원 배열입니다. 배열의 개별 항목은 *picture elements*(그림 요소)의 약어인 *픽셀*이라고 부릅니다.[13] 픽셀 맵의 *연결*

[12] 원문을 그대로 인용하면 다음과 같습니다. "선행자 부분 그래프가 하나의 트리를 이루는 너비 우선 탐색과 달리, 깊이 우선 탐색이 만드는 선행자 부분 그래프는 여러 개의 트리로 이루어질 수 있습니다. 탐색이 여러 출발점에서 반복될 수 있기 때문입니다."

[13] 1960년대에 현대적인 래스터 디스플레이 장치가 등장하기 전에는 픽셀이 실로 만들어졌는지 아주 작은 돌로 만들어졌는지에 따라 *stitches*(바늘땀) 또는 *tesserae*(모자이크 조각)라는 이름으로 더 흔히 알려져

*영역*은 모두 같은 색을 가진 픽셀들의 연결된 부분집합이며, 이때 두 픽셀은 가로 또는 세로로 바로 이웃해 있으면 인접한 것으로 간주합니다. 래스터 그래픽 편집 소프트웨어에서 흔히 페인트 통으로 표시되는 *플러드 필* 연산은 연결 영역의 모든 픽셀을 새로운 색으로 바꿉니다. 이 연산의 입력은 대상 영역에 속한 한 픽셀의 인덱스 i와 j, 그리고 새로운 색으로 이루어집니다.

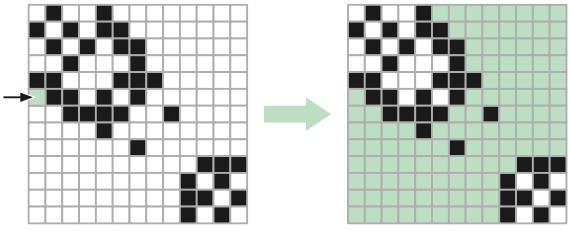


그림 5.13. 플러드 필의 예

플러드 필 문제는 정의를 따라가면 도달 가능성 문제로 환원할 수 있습니다. 정점이 개별 픽셀이고, 간선이 같은 색의 이웃한 픽셀들을 잇는 무방향 그래프 $G = (V, E)$를 정의합니다. 픽셀 맵의 각 연결된 영역은 G의 한 요소이므로, 플러드 필 문제는 G에서의 도달 가능성 문제로 ***환원됩니다***. 이 도달 가능성 문제는 주어진 픽셀 (i, j)에서 시작하여 G에서 아무거나 우선 탐색을 사용하되, 한 가지만 약간 수정하여 풀 수 있습니다. 정점을 표시할 때마다 즉시 그 색을 바꾸는 것입니다. $n \times n$ 픽셀 맵에 대해 그래프 G는 n^2개의 정점과 최대 $2n^2$개의 간선을 가지므로, 아무거나 우선 탐색은 $O(V + E) = \boldsymbol{O(n^2)}$ *시간*에 실행됩니다.

이 간단한 예는 ***환원***의 핵심 요소를 보여 줍니다. 플러드 필 문제를 처음부터 푸는 대신, 기존 알고리즘을 블랙박스 서브루틴으로 사용합니다. 아무거나 우선 탐색이 *어떻게* 동작하는지는 여기서 전혀 중요하지 않습니다. 중요한 것은 오직 그 *명세*뿐입니다. 즉, 그래프 G와 시작 정점 s가 주어지면 G에서 s로부터 도달할 수 있는 모든 정점을 표시합니다. 다른 서브루틴과 마찬가지로, 입력을 어떻게 구성하고 그 출력을 어떻게 사용할지는 여전히 설명해야 합니다. 또한 그 결과로 얻은 *우리의* 알고리즘은 알고리즘이 구성하는 중간 그래프의 정점과 간선이 아니라 *우리의* 입력 매개변수를 기준으로 분석해야 합니다.

이제 제대로 동작하는 알고리즘을 얻었으므로—하지만 *지금에서야* 비로소—이를 더 빠르게 만드는 간단한 최적화 두 가지를 적용할 수 있습니다. 하나는 실용적인 것이고, 다른 하나는 이론적인 것입니다:

- 실제 구현에서는 G에 대한 별도의 그래프 자료구조를 실제로 만들지 않습니다. 그 대신 픽셀 맵을 표준 인접 리스트*인 것처럼* 직접 사용할 수 있습니다. 어떤 픽셀이든 색이 같은 이웃을 하나당 $O(1)$ 시간에 나열할 수 있기 때문입니다. 특히 정점을 따로 "표시"할 필요가 없으며, 그 대신 픽셀의 색을 사용할 수 있습니다.

있었습니다. *pix*라는 단어는 20세기 초에 *picture(s)*의 표준 약어가 되었으며(*sox*가 *sock*의 흔한 복수형이 된 지 얼마 지나지 않은 때였습니다), 이전의 구어 표현인 *piccy*를 대체했습니다. 복셀(voxel, volume element), 텍셀(texel, texture element), 택셀(taxel, tactile element 그리고/또는 오소리)도 참고하십시오.

- 더 세밀하게 분석하면 실행 시간이 채워지는 영역의 픽셀 수, 다시 말해 정점 (i, j)를 포함하는 G의 연결 요소에 속한 정점의 수에 비례함을 알 수 있으며, 이 값은 $O(n^2)$보다 상당히 작을 수 있습니다.

연습문제

그래프

1. 다음 정의들이 모두 동치임을 증명하십시오.

   - 트리는 연결된 비순환 그래프입니다.
   - 트리는 숲의 한 연결 요소입니다. (숲은 비순환 그래프입니다.)
   - 트리는 간선이 *최대* $V - 1$개인 연결 그래프입니다.
   - 트리는 최소 연결 그래프입니다. 즉, 어떤 간선을 제거하더라도 그래프의 연결이 끊어집니다.
   - 트리는 간선이 *적어도* $V - 1$개인 비순환 그래프입니다.
   - 트리는 극대 비순환 그래프입니다. 즉, 임의의 두 정점 사이에 간선을 하나라도 추가하면 사이클이 생깁니다.
   - 트리는 모든 정점 쌍 사이에 유일한 경로가 존재하는 그래프입니다.

2. $n \geq 2$개의 정점을 가진 연결된 비순환 그래프에는 항상 차수가 1인 정점이 적어도 두 개 있음을 증명하십시오. "트리"나 "리프"라는 단어는 쓰지 말고, 트리의 잘 알려진 성질도 사용하지 마십시오. 증명은 오직 "연결"과 "비순환"의 정의만으로 이루어져야 합니다.

3. 정점 집합 V를 두 부분집합 L과 R로 분할하여 모든 간선의 한 끝점은 L에, 다른 끝점은 R에 있게 할 수 있으면, 그래프 (V, E)를 *이분* 그래프라고 합니다.

   (a) 모든 트리가 이분 그래프임을 증명하십시오.

   (b) 그래프 G가 이분 그래프일 필요충분조건은 G의 모든 사이클이 짝수 개의 간선을 가지는 것임을 증명하십시오.

   (c) 주어진 무방향 그래프가 이분 그래프인지 판별하는 효율적인 알고리즘을 설명하고 분석하십시오.

4. 비둘기 무리는 모일 때마다 본능적으로 *쪼는 순서*를 정합니다. 어떤 비둘기 쌍이든 한 비둘기가 항상 다른 비둘기를 쪼아서 먹이나 잠재적인 짝으로부터 쫓아냅니다. 같은 비둘기 쌍은 주변에 다른 비둘기가 있든 없든, 몇 년 동안 떨어져 지낸 뒤에도 항상 같은 쪼는 순서를 택합니다. 놀랍게도 전체 쪼는 순서에는 사이클이 있을 수 있습니다—예를 들어 비둘기 i가 비둘기 j를 쪼고, j가 비둘기 k를 쪼고, k가 비둘기 ℓ을 쪼고, ℓ이 비둘기 i를 쪼는 경우입니다.

(a) 유한한 비둘기 무리라면 어떤 무리든, 각 비둘기가 바로 앞 비둘기의 꽁무니를 쪼도록 행렬(어쩌면 퍼레이드?)로 늘어세울 수 있음을 증명하십시오. 부디 부탁드립니다.

(b) n마리 비둘기 집합 사이의 쪼기 관계를 나타내는 방향 그래프가 주어졌다고 가정합시다. 이 그래프에는 비둘기마다 정점이 하나씩 있으며, 간선 $i \to j$는 비둘기 i가 비둘기 j를 쫄 때, 그리고 오직 그때에만 존재합니다. (a)에서 존재가 보장된 비둘기들의 쪼기 순서를 계산하는 알고리즘을 설명하고 분석하십시오.

(c) 적어도 세 마리의 비둘기로 이루어진 임의의 집합에 대해, (a)에서 설명한 쪼기 서열이 유일하거나, 비둘기 i가 비둘기 j를 쪼고, 비둘기 j가 비둘기 k를 쪼고, 비둘기 k가 다시 비둘기 i를 쪼는 세 비둘기 i, j, k가 존재함을 증명하십시오.

5. 그래프 G의 ***오일러 투어***란 G의 모든 간선을 정확히 한 번씩 지나는 G 위의 닫힌 보행입니다.

(a) 연결 그래프 G에 오일러 투어가 존재하면 G의 모든 정점의 차수가 짝수임을 증명하십시오. (이는 오일러가 증명했습니다.)

(b) 연결 그래프 G의 모든 정점의 차수가 짝수이면 G에 오일러 투어가 존재함을 증명하십시오. (오일러는 이것을 증명하지 *않았습니다*.)

(c) 주어진 그래프에서 오일러 투어를 계산하거나, 그러한 투어가 존재하지 않음을 올바르게 보고하는 알고리즘을 설명하고 분석하십시오. (오일러는 이 부분을 엄밀한 증명 없이 대충 얼버무리고 넘어갔습니다.)

6. d차원 하이퍼큐브는 다음과 같이 정의되는 그래프입니다. 정점은 2^d개이며, 각 정점에는 서로 다른 d비트 문자열이 레이블로 붙어 있습니다. 두 정점의 레이블이 정확히 한 비트만 다르면 두 정점은 간선으로 연결됩니다.

(a) 그래프 G의 해밀턴 사이클은 G의 각 정점에 정확히 한 번씩 들어가는 G의 간선들로 이루어진 사이클입니다. 모든 $d \geq 2$에 대해 d차원 초입방체가 해밀턴 사이클을 가짐을 증명하십시오.

(b) 어떤 하이퍼큐브에 오일러 투어(모든 간선을 정확히 한 번씩 지나는 닫힌 보행)가 존재합니까? *[힌트: 이것은 매우 쉽습니다.]*

순회 알고리즘

7. 방향 그래프 G가 *강하게 연결되어 있다*는 것은, 임의의 두 정점 u와 v에 대해 G 안에 u에서 v로 가는 경로가 있고 G 안에 v에서 u로 가는 경로도 있다는 뜻임을 떠올려 보십시오.

무방향 그래프 G가 입력으로 주어질 때, G의 각 간선에 방향을 부여하여 그 결과로 얻는 방향 그래프가 강하게 연결되도록 할 수 있는지 판정하는 알고리즘을 설명하십시오.

8. G를 연결 그래프라 하고, T를 어떤 노드 v를 루트로 하는 G의 깊이 우선 신장 트리라고 합시다. T가 v를 루트로 하는 G의 너비 우선 신장 트리이기도 하다면 $G = T$임을 증명하십시오.

9. Epprich 교수와 Goodstein 교수는 일반적인 아무거나 우선 탐색 알고리즘을 다음과 같이 최적화할 것을 제안합니다. 이들의 알고리즘은 가방에서 꺼낸 정점이 표시되어 있는지 확인하는 대신 정점을 가방에 넣기도 전에 미리 확인하며, 이를 통해 각 정점이 가방에 최대 한 번만 들어가도록 보장합니다. 또한 이들의 알고리즘은 각 정점이 표시될 때 그 정점의 부모를 지정합니다.

```
EagerWFS(s):
  mark s
  put s into the bag
  while the bag is not empty
      take v from the bag
      for each edge vw
          if w is unmarked
              mark w
              parent(w) ← v
              put w into the bag
```

   (a) EagerWFS(s)가 s에서 도달 가능한 모든 노드를 표시하고 그 외에는 아무것도 표시하지 않음을 증명하십시오. 이와 동치로, EagerWFS(s)가 계산한 부모 간선 $v \to parent(v)$가 s를 포함하는 성분의 신장 트리를 이룸을 증명하십시오.

   (b) 가방을 큐로 구현하면 EagerWFS가 너비 우선 탐색과 동등하다는 것, 즉 두 알고리즘이 같은 정점들을 같은 순서로 표시하고 같은 신장 트리를 구성한다는 것을 증명하십시오. *[힌트: 큐의 정의는 무엇입니까?]*

   (c) 가방으로 어떤 자료구조를 사용하든(따라서 특히 가방이 스택인 경우에도) EagerWFS는 깊이 우선 탐색과 *결코* 동등하지 않음을 증명하십시오.

   EagerWFS와 RecursiveDFS는 모두 각 정점 v에서 간선 vw를 고려하는 순서를 지정하지 않으며, 간선 순서가 다르면 서로 다른 신장 트리가 만들어질 수 있습니다. 따라서 어떤 구체적인 그래프 G에 대해, RecursiveDFS가 만들어 내는 G의 신장 트리 중 어느 것도 EagerWFS로는 (*어떤* 가방 자료구조를 사용하더라도) 구성할 수 없음을, 또는 그 반대가 성립함을 논증해야 합니다.

10. 아무거나 우선 탐색을 일반적인 알고리즘 부류로서 설명한 가장 초기의 출판물 중 하나는 1975년 Edsger Dijkstra, Leslie Lamport, Alain Martin, Carel Scholten, Elisabeth Steffens가 자동 가비지 컬렉터 설계의 일환으로 발표한 것입니다. 이들의 알고리즘은 표시된 정점과 표시되지 않은 정점을 관리하는 대신, 각 정점마다 흰색, 회색, 검은색 중 하나의 색을 유지합니다. 늘 그렇듯이, 다음 알고리즘에서는 고정된 기저 그래프 G를 가정합니다.

```
ThreeColorSearch(s):
  color all nodes white
  color s gray
  while at least one vertex is gray
      ThreeColorStep()
```

```
ThreeColorStep():
  v ← any gray vertex
  if v has no white neighbors
      color v black
  else
      w ← any white neighbor of v
      parent(w) ← v
      color w gray
```

(a) ThreeColorSearch가 다음 불변식을 항상 유지함을 증명하십시오: 어떤 검은색 정점도 흰색 정점과 이웃하지 않습니다. *[힌트: 어렵지 않을 것입니다.]*

(b) ThreeColorSearch(s)가 종료된 후, s에서 도달할 수 있는 모든 정점은 검은색이고 s에서 도달할 수 없는 모든 정점은 흰색이며, 부모 간선 $v \to parent(v)$가 s를 포함하는 연결 요소의 루트 있는 신장 트리를 정의함을 증명하십시오.

[힌트: 직관적으로 말하면, 검은색 노드는 "표시된" 노드이고 회색 노드는 "가방 안에 있는" 노드입니다. 그러나 우리가 정식화한 WhateverFirstSearch와 달리, 삼색 알고리즘은 한 노드에서 나가는 모든 간선을 한꺼번에 처리하지 않아도 됩니다.]

(c) 회색 정점의 집합을 표준 스택으로 관리하는 다음 ThreeColorSearch의 변형이 깊이 우선 탐색과 동등함을 증명하십시오. *[힌트: ThreeColorStackStep의 마지막 두 줄의 순서가 중요합니다!]*

```
ThreeColorStackSearch(s):
  color all nodes white
  color s gray
  push s onto the stack
  while at least one vertex is gray
      ThreeColorStackStep()
```

```
ThreeColorStackStep():
  pop v from the stack
  if v has no white neighbors
      color v black
  else
      w ← any white neighbor of v
      parent(w) ← v
      color w gray
      push v onto the stack
      push w onto the stack
```

(d) 회색 정점의 집합을 표준 큐로 유지하는 다음 ThreeColorSearch의 변형이 너비 우선 탐색과 동등하지 **않음**을 증명하십시오. *[힌트: ThreeColorQueueStep의 마지막 두 줄은 순서를 바꾸어도 상관없습니다!]*

```
ThreeColorQueueSearch(s):
  color all nodes white
  color s gray
  push s into the queue
  while at least one vertex is gray
      ThreeColorQueueStep()
```

```
ThreeColorQueueStep():
  pull v from the queue
  if v has no white neighbors
      color v black
  else
      w ← any white neighbor of v
      parent(w) ← v
      color w gray
      push v into the queue
      push w into the queue
```

♥(e) 이제 ThreeColorSearch가 실행되는 동안 다른 프로세스가 G에 간선을 추가하고 있다고 가정합시다. 새로 추가된 간선은 (a)에서 설명한 색 불변식을 위반할 수 있고, 그 결과 알고리즘의 정확성을 무너뜨릴 수 있습니다—특히 ThreeColorSearch가 종료될 때 s에서 도달 가능한 일부 정점이 흰색일 수 있습니다. "흰색"이 "도달 불가능하므로 삭제해도 안전함"을 뜻한다고 믿고 있다면 이는 재앙과 같은 결과를 낳을 것입니다.

그러나 다른 프로세스가 색 불변식을 명시적으로 보존한다면, 여전히 삼색 알고리즘을 사용하여 도달할 수 없는 정점을 안전하게 식별할 수 있습니다. 두 동시 알고리즘을 다음과 같이 모델링합니다. GarbageCollect에서 둘 중 어느 쪽을 수행할지에 대한 선택과, 어떤 정점 u와 w를 Mutate할지에 대한 선택은 전적으로 주 알고리즘이 제어할 수 없는 부분입니다.[14]

```
GarbageCollect(s):
  color all vertices white
  color s gray
  while at least one vertex is gray
      either
          CollectStep()
      or
          Mutate()
```

```
CollectStep():
  v ← any gray vertex
  if v has no white neighbors
      color v black
  else
      w ← any white neighbor of
  v
      color w gray
```

```
Mutate():
  u ← any vertex
  w ← any vertex
  if uw is not an edge
      add edge uw
      if u is black and w is white
          color u gray
      if u is white and w is black
          color w gray
```

[14] 이것은 Lua나 Go 같은 다중 스레드 언어에서 실제로 사용되는 "마크 앤 스윕(mark and sweep)" 가비지 컬렉션 알고리즘을 *극단적으로* 단순화한 것입니다. 다중 스레드 동적 메모리 관리에 대한 더 깊이 있는 논의는 안타깝게도 이 책의 범위를 벗어나며, 제1계명만은 예외입니다. **그대는 가비지 컬렉터를 손수 만들지 마십시오.**

GarbageCollect가 결국 종료되며, 종료 시 s에서 도달 가능한 모든 정점은 검은색으로, s에서 도달할 수 없는 모든 정점은 흰색으로 칠해져 있음을 증명하십시오.

♥(f) Mutate가 검은색 정점을 회색으로 다시 칠하는 대신, 일부 *흰색* 정점을 회색으로 칠하여 색 불변식을 유지한다고 가정합시다:

```
Mutate():
  u ← any vertex
  w ← any vertex
  if uw is not an edge
      add edge uw
      if u is black and w is white
          color w gray
      if u is white and w is black
          color u gray
```

GarbageCollect가 결국 종료되며, 종료 시점에 s는 검은색으로 칠해져 있고, 검은색 정점에서 도달 가능한 모든 정점은 검은색으로, 검은색 정점에서 도달할 수 없는 모든 정점은 흰색으로 칠해져 있음을 증명하십시오.

환원

11. ***숫자 미로***는 양의 정수로 이루어진 $n \times n$ 격자입니다. 말은 왼쪽 위 모서리에서 출발하며, 여러분의 목표는 말을 오른쪽 아래 모서리로 옮기는 것입니다. 매 차례 말을 위, 아래, 왼쪽, 오른쪽 중 한 방향으로 옮길 수 있으며, 말을 옮길 수 있는 거리는 말이 현재 놓인 칸의 숫자로 정해집니다. 예를 들어 말이 3이 적힌 칸에 있다면, 말을 위로 세 칸, 아래로 세 칸, 왼쪽으로 세 칸, 또는 오른쪽으로 세 칸 옮길 수 있습니다. 그러나 말을 보드의 가장자리(간선) 밖으로 옮기는 것은 절대 허용되지 않습니다.

주어진 숫자 미로를 푸는 데 필요한 최소 이동 횟수를 반환하거나, 그 미로에 해가 없음을 올바르게 보고하는 효율적인 알고리즘을 설명하고 분석하십시오. 예를 들어 그림 5.14의 숫자 미로가 주어지면, 여러분의 알고리즘은 정수 8을 반환해야 합니다.

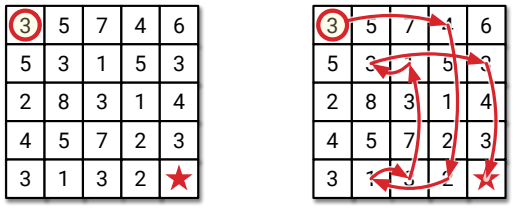


그림 5.14. 여덟 번의 이동으로 풀 수 있는 5 × 5 숫자 미로입니다.

12. ***뱀과 사다리***는 늦어도 16세기에 인도에서 유래한 고전 보드게임입니다. 게임판은 $n \times n$ 격자 모양의 칸으로 이루어져 있으며, 각 칸에는 1부터 n^2까지 연속된 번호가 매겨져 있습니다. 번호는 왼쪽 아래 모서리에서 시작하여 아래에서 위로 한 행씩 올라가며, 행마다 진행 방향이 왼쪽과 오른쪽으로 번갈아 바뀝니다. 이 격자에서 항상 서로 다른 행에 있는 특정 칸 쌍들은 "뱀"(아래로 이어짐) 또는 "사다리"(위로 이어짐)로 연결되어 있습니다. 각 칸은 최대 하나의 뱀 또는 사다리의 끝점이 될 수 있습니다.

    왼쪽 아래 모서리에 있는 칸 1에 말을 하나 놓고 시작합니다. 한 번 이동할 때마다 어떤 고정된 상수 k에 대해 말을 최대 k칸까지 전진시킵니다. 이동을 마친 말이 뱀의 *위쪽* 끝에 놓이면, 말은 그 뱀의 아래쪽 끝으로 미끄러져 내려갑니다. 마찬가지로, 이동을 마친 말이 사다리의 *아래쪽* 끝에 놓이면, 말은 그 사다리의 꼭대기까지 올라갑니다.

    토큰이 격자의 마지막 칸에 도달하는 데 필요한 최소 이동 횟수를 계산하는 알고리즘을 설명하고 분석하십시오.

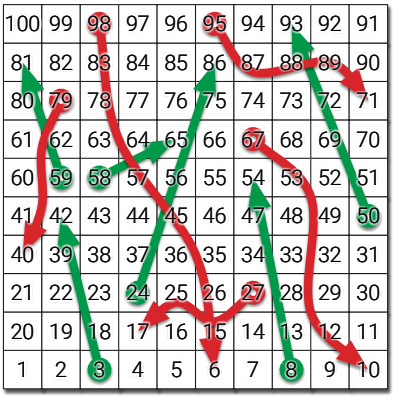


그림 5.15. 뱀과 사다리 게임판입니다. 위로 향하는 곧은 화살표는 사다리이고, 아래로 향하는 물결 모양 화살표는 뱀입니다.

13. 악명 높은 몽골의 퍼즐 전사 비드라흐 이트키 레다는 1473년에 다음과 같은 퍼즐을 고안했습니다. 이 퍼즐은 각 칸에 양의 정수가 적힌 $n \times n$ 정사각형 격자, 그리고 빨간색 말 하나와 파란색 말 하나로 이루어져 있습니다. 두 말은 항상 격자의 서로 다른 칸에 놓입니다. 두 말은 격자의 왼쪽 위 모서리와 오른쪽 아래 모서리에서 출발하며, 퍼즐의 목표는 두 말의 위치를 서로 맞바꾸는 것입니다.

    한 번의 차례에서 두 토큰 중 하나를 위, 오른쪽, 아래, 왼쪽 중 한 방향으로 ***다른*** *토큰이 정하는 거리만큼* 옮길 수 있습니다. 예를 들어 빨간 토큰이 3이라고 적힌 칸에 있다면, 파란 토큰을 위로 3칸, 왼쪽으로 3칸, 오른쪽으로 3칸, 또는 아래로 3칸 옮길 수 있습니다. 그러나 어느 토큰도 격자 밖으로 옮길 수 없으며, 한 번 옮기고 난 뒤에는 두 토큰이 같은 칸에 놓일 수 없습니다.

주어진 Vidrach Itky Leda 퍼즐을 푸는 데 필요한 최소 이동 횟수를 반환하거나, 퍼즐에 해가 없음을 올바르게 보고하는 효율적인 알고리즘을 설명하고 분석하십시오. 예를 들어, 그림 5.16의 퍼즐이 주어지면 여러분의 알고리즘은 수 5를 반환해야 합니다.

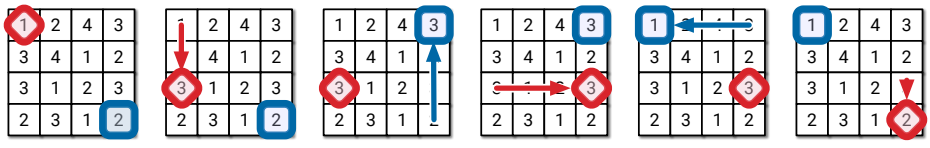
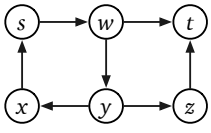


그림 5.16. 4 × 4 Vidrach Itky Leda 퍼즐을 다섯 수 만에 푸는 풀이입니다.

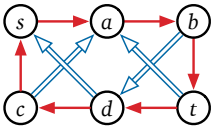
14. 방향 그래프 $G = (V, E)$와 두 정점 s, t가 주어졌다고 가정합시다. G에서 s로부터 t로 가는 보행(정점이나 간선, 또는 둘 다 반복할 수 있습니다) 중 길이가 3으로 나누어떨어지는 보행이 존재하는지 판별하는 알고리즘을 설명하고 분석하십시오.

    예를 들어, 아래 그래프와 지정된 정점 s, t가 주어지면, 보행 $s \to w \to y \to x \to s \to w \to t$의 길이가 6이므로 여러분의 알고리즘은 TRUE를 반환해야 합니다.



15. 일부 간선은 빨간색이고 나머지 간선은 파란색인 방향 그래프 G가 주어졌다고 가정합시다. G에서 한 정점 s로부터 다른 정점 t까지, 연속한 세 간선의 색이 모두 같은 경우가 없는 최단 보행을 찾는 알고리즘을 설명하십시오. 즉, 보행에 빨간 간선 두 개가 연달아 나타나면 다음 간선은 반드시 파란색이어야 하고, 파란 간선 두 개가 연달아 나타나면 다음 간선은 반드시 빨간색이어야 합니다.

    예를 들어, 다음 그래프가 입력으로 주어지면 여러분의 알고리즘은 정수 7을 반환해야 합니다. $s \to a \to b \Rightarrow d \to c \Rightarrow a \to b \to t$가 s에서 t로 가는 가장 짧은 적법한 보행이기 때문입니다.



16. 각 간선이 빨간색, 흰색, 파란색 중 하나로 칠해진 방향 그래프 G를 생각해 봅시다. G의 보행에서 간선 색의 순서가 빨간색, 흰색, 파란색, 빨간색, 흰색, 파란색, … 순으로 이어지면 그 보행을 *프랑스 국기 보행*이라고 합니다. 더 형식적으로 말하면, 모든 정수 i에 대해 간선 $v_i \to v_{i+1}$이 $i \bmod 3 = 0$이면 빨간색, $i \bmod 3 = 1$이면 흰색, $i \bmod 3 = 2$이면 파란색일 때 보행 $v_0 \to v_1 \to \cdots \to v_k$를 프랑스 국기 보행이라고 합니다.

주어진 정점 v에서 프랑스 국기 보행을 통해 도달할 수 있는 G의 모든 정점을 찾는 알고리즘을 설명하십시오.

17. m개의 은하 간 순간이동 통로로 연결된 n개의 은하가 있습니다. 각 순간이동 통로는 두 은하를 잇고 양방향으로 지나갈 수 있습니다. 또한 각 순간이동 통로 e에는 $c(e)$달러의 비용이 매겨져 있으며, 여기서 $c(e)$는 양의 정수입니다. 순간이동 통로는 여러 번 사용할 수 있지만, 사용할 때마다 통행료를 내야 합니다.

주디는 은하 s에서 은하 t로 이동하려고 합니다. 그런데 순간이동은 그다지 유쾌한 경험이 아니어서 순간이동 횟수를 최소한으로 줄이고 싶어 합니다. 하지만 잔돈을 들고 다니는 것도 유쾌하지 않기 때문에, 주디는 총비용이 5달러의 배수가 되기를 원합니다.

(a) 총비용이 5달러의 배수가 되도록 주디가 은하 s에서 은하 t까지 이동할 때 필요한 최소 순간이동 횟수를 계산하는 알고리즘을 설명하고 분석하십시오.

(b) (a)번 문제를 푸십시오. 단, 이번에는 주디에게 순간이동 통로 정확히 하나를 무료로 이용할 수 있는 쿠폰이 있다고 가정하십시오.

18. Three Seashells는 연결된 무방향 그래프 G 위에서 하는 1인용 게임입니다. 처음에는 서로 다른 세 시작 정점 a, b, c에 토큰 세 개를 하나씩 놓습니다. 매 턴마다 각 토큰을 현재 정점에서 간선을 따라 인접한 정점으로 옮겨서, *세 토큰 모두를 반드시* 움직여야 합니다. 각 턴이 끝났을 때 세 토큰은 *반드시* 서로 다른 세 정점 위에 있어야 합니다. 목표는 토큰들을 세 목표 정점 x, y, z 위로 옮기는 것이며, 어느 토큰이 어느 목표 정점에 도착하는지는 상관없습니다.

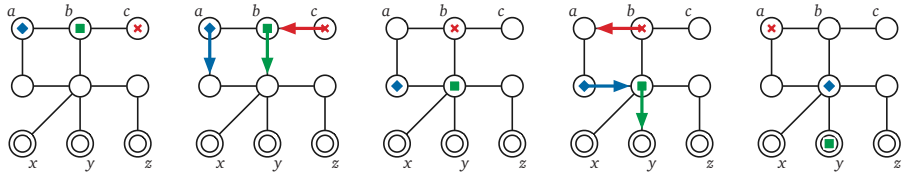


그림 5.17. 세 조개껍데기(Three Seashells) 퍼즐의 초기 배치와 풀이의 처음 두 수입니다.

주어진 세 조개껍데기(Three Seashells) 퍼즐을 풀 수 있는지 판별하는 알고리즘을 설명하고 분석하십시오. 입력은 그래프 G, 시작 정점 a, b, c, 목표 정점 x, y, z로 이루어집니다. 출력은 TRUE 또는 FALSE 중 하나인 단일 비트입니다.

19. G를 연결된 무방향 그래프라고 합시다. G에서 임의로 고른 두 정점에 동전 두 개를 하나씩 놓고 시작하여, 가능한 한 적은 횟수만 움직여 두 동전을 같은 정점에 모으려 한다고 가정합시다. 매 단계마다 각 동전은 *반드시* 인접한 정점으로 이동해야 합니다.

(a) 두 동전이 같은 정점에 놓이는 배치에 도달하기 위한 최소 단계 수를 계산하거나, 그러한 배치에 도달할 수 없음을 올바르게 보고하는 알고리즘을 설명하고

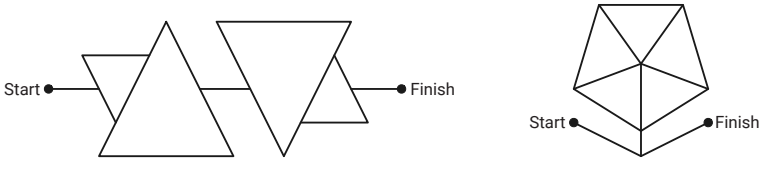
분석하십시오. 알고리즘의 입력은 그래프 $G = (V, E)$와 두 정점 $u, v \in V$로 구성됩니다(두 정점은 서로 다를 수도 있고 같을 수도 있습니다).

(b) 이제 동전이 세 개 있다고 가정합시다. 두 동전이 같은 정점에 놓이는 배치에 도달하기 위한 최소 단계 수를 계산하거나, 그러한 배치에 도달할 수 없음을 올바르게 보고하는 알고리즘을 설명하고 분석하십시오.

(c) 마지막으로, 동전이 *마흔두* 개 있다고 가정합시다. 42개의 동전을 모두 같은 정점으로 옮기는 것이 가능한지 판정하는 알고리즘을 설명하고 분석하십시오. 이번에도 *매* 단계마다 모든 동전이 움직여야 합니다. 만점을 받으려면 알고리즘이 $O(V + E)$ 시간에 실행되어야 합니다.

20. 제 딸의 초등학교 수학 문제집 중 한 권[15]에는 다음과 같은 유형의 퍼즐이 여러 개 실려 있습니다:

> 출발점에서 도착점까지 예각으로만 이루어진 경로를 그려 아래의 각 각도 미로를 완성하십시오.



임의의 예각 미로를 푸는 알고리즘을 설명하고 분석하십시오.

연결된 무방향 그래프 G가 주어집니다. 이 그래프의 정점은 평면 위의 점이고 간선은 선분입니다. 간선들은 끝점을 제외하고는 서로 교차하지 않습니다. 예를 들어 글자 X를 그린 그림은 정점 5개와 간선 4개로 이루어지며, 위의 첫 번째 미로는 정점 18개와 간선 21개로 이루어집니다. 또한 두 정점 Start와 Finish가 주어집니다.

알고리즘은 G에 Start에서 Finish까지 예각만을 갖는 보행이 존재하면 True를, 그렇지 않으면 False를 반환해야 합니다. 형식적으로 말하면, G를 지나는 보행은 그 보행에서 연속하는 임의의 두 간선 $u \to v \to w$에 대해 $\angle uvw = \pi$이거나 $0 < \angle uvw < \pi/2$일 때 유효합니다. 주어진 두 선분 사이의 각이 평각인지, 둔각인지, 직각인지, 예각인지를 $O(1)$ 시간에 판별할 수 있는 서브루틴이 있다고 가정하십시오.

21. 평면 위에 n개의 수평 및 수직 선분의 집합과 두 점 s, t가 주어졌다고 가정합시다. 주어진 선분 중 어느 것과도 교차하지 않는 s에서 t로의 경로가 존재하는지 판별하는 효율적인 알고리즘을 설명하십시오.

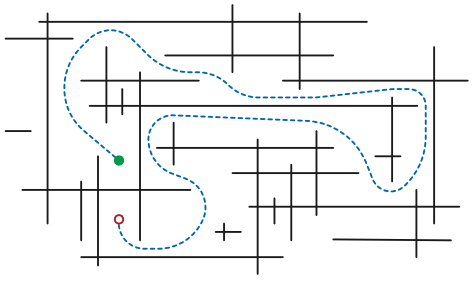


그림 5.18. 수평 선분과 수직 선분으로 이루어진 미로에서 두 점을 잇는 경로입니다.

각 수평 선분은 왼쪽과 오른쪽 x-좌표, 그리고 하나뿐인 y-좌표로 지정됩니다. 마찬가지로 각 수직 선분은 하나뿐인 x-좌표와 위쪽과 아래쪽 y-좌표로 지정됩니다. 마지막으로 점 s와 t는 각각 x-좌표와 y-좌표로 지정됩니다.

22. 모든 진부한 로맨스 영화에는 길고 애타는 이별 끝에 두 연인이 먼 거리를 사이에 두고 갑자기 서로를 발견하는 장면이 있습니다. 두 사람은 흔들림 없이 눈을 맞춘 채 천천히 서로에게 다가갑니다. 그동안 음악이 흘러나오고, 비가 그치고, 구름 사이로 햇살이 비치고, 음악이 고조되고, 모두가 무지개와 새끼 고양이와 초콜릿 유니콘과 함께 춤추기 시작하고….[16]

    컴퓨터 과학의 유서 깊은 전통에 따라 앨리스와 밥이라는 이름을 가진 한 연인이 즐겨 찾는 공원의 동쪽 입구와 서쪽 입구로 각각 들어서자마자 서로 눈을 마주쳤다고 가정합시다. 두 사람은 서로를 향해 곧장 달려갈 수 없습니다. 대신 동쪽 입구와 서쪽 입구 사이에서 공원을 지그재그로 가로지르는 경로를 벗어나지 말아야 합니다. 적절한 극적 긴장감을 유지하기 위해 앨리스와 밥은 항상 정확히 동서 방향의 한 직선 위에 놓이도록 경로를 따라 이동해야 합니다.

    지그재그 경로는 두 배열 $X[0..n]$과 $Y[0..n]$으로 나타낼 수 있습니다. 이 두 배열에는 경로의 꼭짓점들의 x좌표와 y좌표가 남서쪽 끝점부터 남동쪽 끝점까지 순서대로 들어 있습니다. X 배열은 오름차순으로 정렬되어 있으며, $Y[0] = Y[n]$입니다. 경로는 이 꼭짓점들을 잇는 직선 선분들이 차례로 이어진 것입니다.

    (a) $Y[0] = Y[n] = 0$이고 나머지 모든 인덱스 i에 대해 $Y[i] > 0$이라고 가정합시다. 즉, 경로의 두 끝점은 경로 위의 다른 모든 점보다 엄격하게 아래에 있습니다. 이 조건을 만족하는 임의의 경로 P에 대해 앨리스와 밥이 *항상* 만날 수 있음을 증명하십시오. *[힌트: 경로 위에서 두 사람이 있을 수 있는 모든 위치를 모델링하는 그래프를 기술하십시오. 이 그래프의 정점은 무엇입니까? 간선은 무엇입니까?* ***악수 보조정리****를 사용하십시오. 모든 그래프에서 차수가 홀수인 정점의 개수는 짝수입니다.]*

[15] Jason Batterson and Shannon Rogers, *Beast Academy Math: Practice 3A*, 2012. 더 많은 예시는 https://www.beastacademy.com/resources/printables.php에서 보실 수 있습니다.

[16] 재미있는 사실: *Whiplash*와 *La La Land*의 감독인 데이미언 셔젤은 프린스턴 대학교 컴퓨터 과학 교수이자 일렉트릭 기타리스트인 버나드 셔젤의 아들입니다.

(b) 경로의 양 끝점이 다른 모든 정점보다 아래에 있지 *않다면*, 앨리스와 밥은 여전히 만날 수도 있고 만나지 못할 수도 있습니다. 배열 $X[0..n]$과 $Y[0..n]$이 입력으로 주어졌을 때, 두 사람 중 누구도 동서 방향의 눈 맞춤을 끊거나 경로를 벗어나지 않으면서 앨리스와 밥이 만날 수 있는지 판정하는 알고리즘을 설명하십시오.

♥(c) (b)번 문항에 대해 $O(n)$ 시간에 실행되는 알고리즘을 설명하십시오.

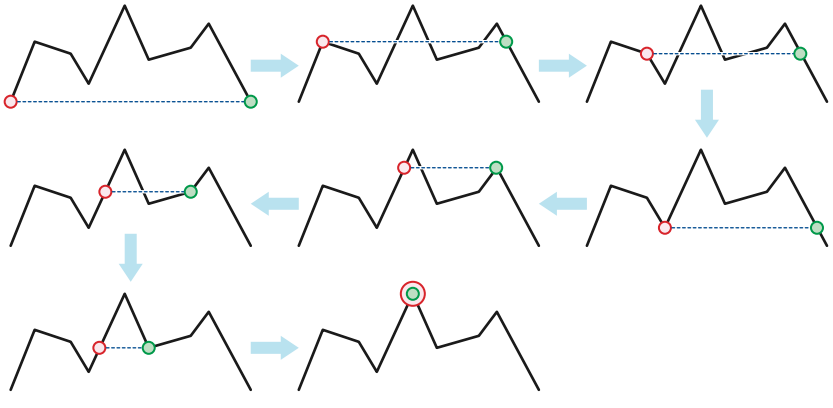


그림 5.19. 앨리스와 밥이 만납니다. 앨리스는 2단계에서 뒤로 걷고, 밥은 5단계와 6단계에서 뒤로 걷습니다.

23. 유명한 퍼즐 제작자인 덴마크인 카니엘은 $n \times n$ 정사각형 격자 위에서 두 개의 말을 가지고 하는 1인용 게임을 고안했습니다. 격자의 일부 칸은 *장애물*로 표시되어 있고, 격자 칸 하나는 *대상*으로 표시되어 있습니다. 매 차례마다 플레이어는 말 하나를 현재 위치에서 위, 아래, 오른쪽, 왼쪽 중 한 방향으로 *가능한 한 멀리* 움직여야 하며, 말은 (1) 판의 경계 간선, (2) 장애물 칸, (3) 다른 말 중 하나에 부딪히기 직전에 멈춥니다. 목표는 두 말 중 하나를 대상 칸 위로 옮기는 것입니다.

예를 들어, 그림 5.20에 나온 퍼즐은 빨간 토큰을 장애물에 닿을 때까지 아래로 옮기고, 이어서 초록 토큰을 빨간 토큰에 닿을 때까지 왼쪽으로 옮긴 다음, 빨간 토큰을 왼쪽, 아래, 오른쪽, 위로 차례로 옮겨서 풀 수 있습니다. 빨간 토큰이 여섯 번째 이동에서 대상 칸에 멈추는 것은 초록 토큰이 대상 칸 바로 위에 있기 *때문*입니다.

이 퍼즐의 주어진 사례가 풀 수 있는지 판정하는 알고리즘을 설명하고 분석하십시오. 입력은 정수 n, 장애물 위치 목록, 대상 위치, 그리고 토큰들의 초기 위치로 구성됩니다. 알고리즘의 출력은 하나의 불 값으로, 주어진 퍼즐을 풀 수 있으면 TRUE, 그렇지 않으면 FALSE입니다. *[힌트: 그래프를 구성하는 데 필요한 시간을 잊지 마십시오.]*

♥24. ***Rectangle Walk***는 새로 나온 추상 퍼즐 게임으로, Steam, iOS, Android, Xbox One, Playstation 5, Nintendo Wii U, Atari 2600, Palm Pilot, Commodore 64, TRS-80, Sinclair ZX-1, DEC PDP-8, PLATO, Zuse Z3, Duramesc, 오드너 아리

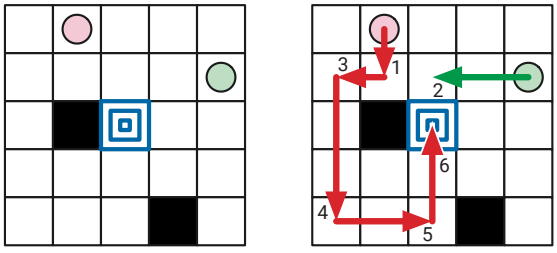


그림 5.20. 여섯 번의 이동으로 풀 수 있는 덴마크인 카니엘의 퍼즐 예시입니다. 원은 토큰의 초기 위치를 나타내고, 검은 칸은 장애물이며, 가운데 칸이 대상입니다.

스모미터, 해석 기관, 자카드 직기, Horologium Mirabile Lundense, 라이프니츠 계단식 계산기, 알자자리의 로봇 악단, 언사의 자동인형, 안티키테라 기계, 매듭 끈, 이상고 뼈, 그리고 돌무더기에서 단돈 99¢에 즐길 수 있습니다.

이 게임은 검은색과 흰색 칸으로 이루어진 $n \times n$ 격자 위에서 진행됩니다. 플레이어는 다음 조건에 따라 이 격자 위에서 직사각형을 움직입니다.

- 직사각형은 격자에 맞춰 정렬되어야 합니다. 즉, 위, 아래, 왼쪽, 오른쪽 좌표가 모두 정수여야 합니다.
- 직사각형은 $n \times n$ 격자 안에 들어가야 하며, 적어도 하나의 격자 칸을 포함해야 합니다.
- 직사각형에는 검은 칸이 포함되어서는 안 됩니다.
- 플레이어는 한 번의 수로 현재 직사각형 r을, r을 포함하거나 r에 포함되는 임의의 직사각형 r'으로 바꿀 수 있습니다.

처음에 플레이어의 직사각형은 오른쪽 위 모서리에 있는 1×1 정사각형입니다. 플레이어의 목표는 가능한 한 적은 횟수의 이동으로 왼쪽 아래 모서리에 있는 1×1 정사각형에 도달하는 것입니다.

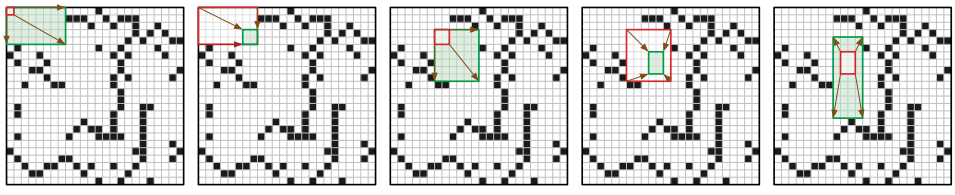


그림 5.21. 직사각형 걷기(Rectangle Walk)의 처음 다섯 단계입니다.

주어진 비트맵에서 가장 짧은 직사각형 걷기(Rectangle Walk)의 길이를 계산하는 알고리즘을 설명하고 분석하십시오. 입력은 배열 $M[1..n, 1..n]$이며, $M[i,j] = 1$은 검은 칸을, $M[i,j] = 0$은 흰 칸을 나타냅니다. 유효한 직사각형 걷기가 존재한다고 가정합니다. 특히 $M[1,1] = M[n,n] = 0$입니다. 예를 들어 위에 보인 비트맵이 주어지면 알고리즘은 정수 18을 반환해야 합니다. *[힌트: 그래프를 구성하는 데 걸리는 시간도 잊지 마십시오!!]*

25. ***레이스트랙***(*그래프 레이서* 또는 *벡터 랠리*라고도 합니다)은 제프가 5학년 때 버스에서 하던, 종이와 연필로 즐기는 2인용 레이싱 게임입니다.[17] 이 게임은 모눈종이 한 장에 그린 트랙 위에서 진행합니다. 플레이어들은 번갈아 가며 트랙을 도는 자동차의 움직임을 나타내는 일련의 격자점을 선택하며, 이때 아래에서 설명할 몇 가지 제약 조건을 따라야 합니다.

속도	위치
(0, 0)	(1, 5)
(1, 0)	(2, 5)
(2, −1)	(4, 4)
(3, 0)	(7, 4)
(2, 1)	(9, 5)
(1, 2)	(10, 7)
(0, 3)	(10, 10)
(−1, 4)	(9, 14)
(0, 3)	(9, 17)
(1, 2)	(10, 19)
(2, 2)	(12, 21)
(2, 1)	(14, 22)
(2, 0)	(16, 22)
(1, −1)	(17, 21)
(2, −1)	(19, 20)
(3, 0)	(22, 20)
(3, 1)	(25, 21)

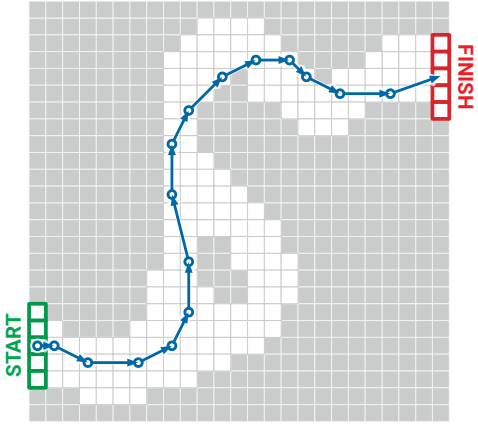


그림 5.22. 25 × 25 트랙에서 16단계로 완주한 Racetrack 주행입니다. 이것은 이 트랙에서 가장 짧은 주행이 *아닙니다*.

각 자동차에는 *위치*와 속도가 있으며, 둘 다 정수 x좌표와 y좌표로 이루어집니다. 격자 칸의 한 부분집합은 *출발 구역*으로 표시되고, 또 다른 부분집합은 *도착 구역*으로 표시됩니다. 각 자동차의 초기 위치는 플레이어가 출발 구역 안의 한 곳으로 선택하며, 각 자동차의 초기 속도는 항상 $(0, 0)$입니다. 각 단계에서 플레이어는 자동차 속도의 두 좌표 중 하나 또는 둘 다를 선택적으로 증가시키거나 감소시킬 수 있습니다. 다시 말해, 속도의 각 성분은 한 단계에서 최대 1만큼 변할 수 있습니다. 그런 다음 자동차의 이전 위치에 새 속도를 더하여 자동차의 새 위치가 결정됩니다. 새 위치는 반드시 트랙 안에 있어야 하며, 그렇지 않으면 자동차가 충돌하고 해당 플레이어는 경주에서 패배합니다. 어느 자동차든 처음으로 도착 구역 안의 위치에 도달하면 경주가 끝납니다.

경주로가 $n \times n$ 비트 배열로 표현된다고 가정합시다. 여기서 각 0 비트는 트랙 안쪽의 격자점을, 각 1 비트는 트랙 바깥쪽의 격자점을 나타내며, "출발 구역"은 첫 번째 열이고 "결승 구역"은 마지막 열입니다.

주어진 경주 트랙의 출발선에서 결승선까지 자동차를 이동시키는 데 필요한 최소 단계 수를 구하는 알고리즘을 설명하고 분석하십시오.

26. *주사위 굴리기 미로*는 표준 육면체 주사위(각 면에 숫자가 적힌 정육면체)와 정사각형 칸으로 이루어진 격자를 사용하는 퍼즐입니다. 격자가 탁자 위에 놓여

[17] 실제 게임은 여기서 설명하는 버전보다 조금 더 복잡합니다. 훌륭한 온라인 버전이 http://harmmade.com/vectorracer/에 있습니다.

있다고 상상하십시오. 주사위는 항상 격자의 한 칸 위에 놓이며 그 칸을 정확히 덮습니다. 한 단계마다 주사위를 아래쪽 모서리 중 하나를 축으로 90도 *굴려서* 북, 남, 동, 서 중 한 방향으로 한 칸 떨어진 인접한 칸으로 옮길 수 있습니다.

격자의 일부 칸은 *막혀* 있을 수 있으며, 주사위는 막힌 칸 위에 절대 놓일 수 없습니다. 다른 칸에는 숫자가 *레이블*로 붙어 있을 수 있으며, 주사위가 레이블이 붙은 칸 위에 놓일 때마다 주사위의 윗면에 있는 숫자는 그 레이블과 같아야 합니다. 레이블도 붙어 있지 않고 표시도 되어 있지 않은 칸은 *자유* 칸입니다. 주사위를 굴려 격자의 가장자리 밖으로 내보낼 수는 없습니다. 이러한 제약 조건 아래에서 주사위를 왼쪽 아래 칸에 놓고 오른쪽 위 칸까지 굴려 갈 수 있다면, 그 주사위 굴리기 미로는 *풀 수 있는* 미로입니다.

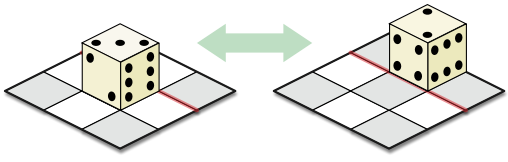


그림 5.23. 주사위 굴리기

그림 5.24에 주사위 굴리기 미로 네 개가 나와 있습니다. 1과 6이 서로 마주보는 면에 있는 표준 주사위를 사용한다고 가정하면, 처음 두 미로만 풀 수 있습니다. 예를 들어 첫 번째 미로는 윗면에 1이 오도록 주사위를 왼쪽 아래 칸에 놓은 다음, 주사위를 동쪽, 북쪽, 북쪽, 동쪽 순서로 굴리면 풀 수 있습니다.

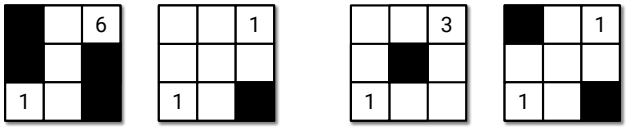


그림 5.24. 주사위 굴리기 미로 네 개입니다. 이 중 처음 두 개만 풀 수 있습니다.

(a) 입력이 2차원 배열 $L[1..n, 1..n]$으로 주어진다고 가정합시다. 각 원소 $L[i, j]$에는 i번째 행, j번째 열에 있는 칸의 레이블이 저장되며, 0은 그 칸이 비어 있음을, -1은 그 칸이 막혀 있음을 뜻합니다. 주어진 주사위 굴리기 미로를 풀 수 있는지 판정하는 다항 시간 알고리즘을 설명하고 분석하십시오.

♥(b) 이제 미로가 레이블이 붙은 칸과 막힌 칸의 목록으로 *암시적*으로 주어진다고 가정합시다. 구체적으로, 입력은 미로의 높이와 너비를 나타내는 정수 M과 배열 $S[1..n]$으로 이루어지며, 각 항목 $S[i]$는 칸 (x, y)의 레이블이 L임을 나타내는 세 값의 쌍 (x, y, L)이라고 가정합시다. 명시적 인코딩에서와 마찬가지로 레이블 -1은 해당 칸이 막혀 있음을 나타냅니다. 빈 칸은 S에 전혀 나열되지 않습니다. 주어진 주사위 굴리기 미로를 풀 수 있는지 판단하는 효율적인 알고리즘을 설명하고 분석하십시오. 만점을 받으려면 알고리즘의 실행 시간이 입력 크기 n에 대한 다항식이어야 합니다.

[힌트: 처음에 주사위를 놓는 방법은 어느 정도 자유롭게 정할 수 있습니다. 초기 위치를 올바르게 선택해야만 풀 수 있는 주사위 굴리기 미로도 있습니다.]

♥27. 각 간선이 빨간색 또는 파란색으로 칠해진 임의의 방향 그래프 G와 함께 두 개의 특별한 정점 s와 t가 주어졌다고 가정합시다.

(a) s에서 t로 가는 보행(walk) 가운데 그 보행을 따라 나타나는 빨간 간선과 파란 간선의 패턴이 회문(palindrome)을 이루는 보행을 계산하거나, 그러한 보행이 존재하지 않음을 올바르게 보고하는 알고리즘을 설명하십시오.

(b) 보행을 따라 나타나는 빨간 간선과 파란 간선의 패턴이 회문이 되도록 하는 s에서 t로의 *가장 짧은* 보행을 계산하거나, 그러한 보행이 존재하지 않음을 올바르게 보고하는 알고리즘을 설명하십시오.

[힌트: 회문은 마지막으로 어디에서 보았습니까?]

♠♥28. 미국에서는 "체커(checkers)"라고도 불리는 드래프츠(draughts)는 밝은색과 어두운색이 번갈아 칠해진 $m \times m$ 격자판 위에서 진행되는 게임입니다.[18] 이 게임은 보통 8×8 또는 10×10 판에서 진행되지만, 규칙은 어떤 크기의 판에도 쉽게 일반화됩니다. 각 어두운 칸에는 검은색 또는 흰색의 게임 말(미국에서는 보통 *체커(checker)*라고 부릅니다)이 최대 하나 놓이며, 밝은 칸은 항상 비어 있습니다. 한 플레이어("백")는 흰 말을 움직이고, 다른 플레이어("흑")는 검은 말을 움직입니다. 플레이어는 자신의 마지막 말이 판에서 제거되면 패배합니다.

다음과 같은 간단한 버전의 게임을 생각해 봅시다. 본질적으로 미국식 체커 또는 영국식 드래프트와 같지만, 모든 말이 킹인 게임입니다.[19] 말은 네 대각선 방향 중 어느 쪽으로든 움직일 수 있습니다. 각 차례에 플레이어는 자신의 말 하나를 대각선으로 한 칸 떨어진 빈 칸으로 *이동*시키거나, 자신의 말 하나로 연속된 *점프*를 합니다. 각 점프에서 말은 임의의 대각선 방향으로 두 칸 떨어진 빈 칸으로 이동하되, 그 사이의 칸에 상대편 색의 말이 있을 때에만 점프할 수 있습니다. 이 상대편 말은 *잡혀서* 즉시 판에서 제거됩니다. 같은 차례의 모든 점프는 같은 말로 해야 합니다.

백이 *단 한 번의 차례에* 흑 말을 모두 잡아 게임에서 이길 수 있는지 판정하는 알고리즘을 설명하십시오. 입력은 판의 너비(m), 백 말들의 위치 목록, 흑 말들의 위치 목록으로 이루어집니다. 만점을 받으려면 알고리즘이 $O(n)$ 시간에 실행되어야 하며, 여기서 n은 전체 말의 개수입니다. *[힌트: 탐욕 전략—막힐 때까지 아무 점프나 하는 것—은 이기는 점프 순서가 존재하더라도 그것을 항상 찾아내지는 **못합니다**. 문제 5를 참고하십시오. 홀짝성, 홀짝성, 홀짝성.]*

[18] 중세 잉글랜드 정부의 회계관들이 사용하던 계산대는 검은 칸이 체크무늬로 배치된 초록색 천으로 덮여 있었으며, 원반 모양의 계산용 말을 이 칸들에 놓아 값을 나타냈습니다. 이러한 이유로 영국 정부의 회계관들은 10세기부터 통틀어 *재무부(Exchequer)*라고 불려 왔습니다. 재무부는 19세기에 들어서도 한참 동안 실제 계산대를 사용하여 세금 납부액을 집계했습니다.

[19] 드래프트의 다른 변형 대부분에는 "날아다니는 킹"이 있습니다. 이 킹은 영국식/미국식 게임의 킹과 *매우* 다르게 움직이며, 12장에서 보게 되듯이 이 문제를 *훨씬* 더 어렵게 만듭니다.

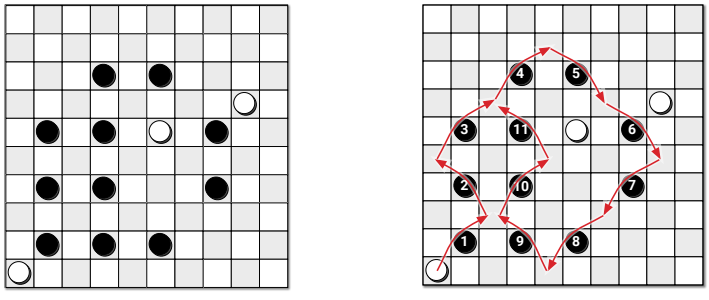


그림 5.25. 백이 한 수 만에 이깁니다.

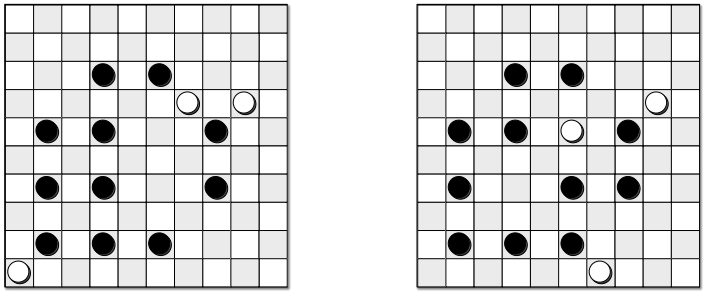


그림 5.26. 백은 이 두 국면 중 어느 쪽에서도 한 수 만에 이길 수 없습니다.

그 집은 이리저리 구불구불하고,
지나갈 길이 참으로 기묘하니—
미궁처럼 얽히어 지어졌기 때문입니다—
이에 제 마음속에 한 가지 방책이 있습니다.
실 한 타래를 가지고, 그가 들어간 그대로
같은 길을 곧바로 되짚어 돌아올 수 있도록,
그가 온 대로 언제나 실을 따라가게 하는 것입니다.

— 제프리 초서, *착한 여인들의 전설* (1385년경)

"Com'è bello il mondo e come sono brutti i labirinti!" dissi sollevato.
"Come sarebbe bello il mondo se ci fosse una regola per girare nei labirinti," rispose il mio maestro.
["세상은 참으로 아름답고, 미로는 참으로 추합니다!" 저는 안도하며 말했습니다.
"미로를 헤쳐 나가는 절차가 있다면 세상은 얼마나 아름답겠습니까." 스승님께서 대답하셨습니다.]

— 움베르토 에코, *Il nome della rosa* (1980)
윌리엄 위버의 영어 번역본(*The Name of the Rose*) (1983)

6

깊이 우선 탐색

앞 장에서는 무방향 그래프와 방향 그래프를 막론하고 임의의 그래프를 순회하는 범용 알고리즘인 아무거나-우선 탐색(whatever-first search)을 살펴보았습니다. 이 장에서는 이 알고리즘의 특정한 구체화인 *깊이 우선 탐색*에 초점을 맞추며, 주로 방향 그래프에서 이 알고리즘이 어떻게 동작하는지를 다룹니다.

깊이 우선 탐색은 "스택을 사용하는 아무거나 우선 탐색"이라고 정확하게 설명할 수 있지만, 이 알고리즘은 보통 명시적인 스택을 사용하지 않고 재귀적으로 구현합니다:

```
DFS(v):
  if v is unmarked
      mark v
      for each edge v→w
          DFS(w)
```

노드를 재귀적으로 탐색하기 *전에* 그 노드가 표시되어 있는지 확인하면 이 알고리즘을 (실제로는) 약간 더 빠르게 만들 수 있습니다. 이렇게 수정하면 각 정점 v에 대해 DFS(v)를 한 번만 호출하게 됩니다. 또한 PreVisit과 PostVisit이라는 두 블랙박스

서브루틴을 도입하여 정점과 간선에 관한 다른 유용한 정보를 계산하도록 알고리즘을 더 수정할 수 있습니다. 이 두 서브루틴은 당분간 구체적으로 정하지 않고 남겨 둡니다.

```
DFS(v):
  mark v
  PreVisit(v)
  for each edge vw
      if w is unmarked
          parent(w) ← v
          DFS(w)
  PostVisit(v)
```

방향 그래프 G에서 노드 w가 다른 노드 v로부터 *도달 가능*하다는 것, 더 간단히 말해 v가 w에 도달할 수 있다는 것은 G에 v에서 w로 가는 방향 경로가 있을 때, 그리고 오직 그때에만 성립한다는 사실을 떠올려 봅시다. $\boldsymbol{reach(v)}$가 v로부터 도달 가능한 정점들의 집합(v 자신도 포함)을 나타낸다고 합시다. G의 모든 정점의 표시를 지운 다음 DFS(v)를 호출하면, 표시된 정점들의 집합은 정확히 $reach(v)$입니다.

무방향 그래프에서 도달 가능성은 대칭적입니다. 즉, w가 v에 도달할 수 있을 때에만, 그리고 그때에 한해 v가 w에 도달할 수 있습니다. 따라서 무방향 그래프 G의 모든 정점의 표시를 해제한 뒤 DFS(v)를 호출하면 v가 속한 연결 요소 전체를 순회하며, 부모 포인터는 그 연결 요소의 신장 트리를 정의합니다.

아래 그림에서 보듯이 방향 그래프에서는 상황이 더 미묘합니다. 그래프가 "연결되어" 있더라도 서로 다른 정점은 그래프의 서로 다른 부분, 경우에 따라서는 서로 겹치는 부분에 도달할 수 있습니다. DFS(v)가 지정하는 부모 포인터는 v를 루트로 하고 정점 집합이 정확히 $reach(v)$인 트리를 정의하지만, 이 트리가 반드시 그래프의 신장 트리인 것은 아닙니다.

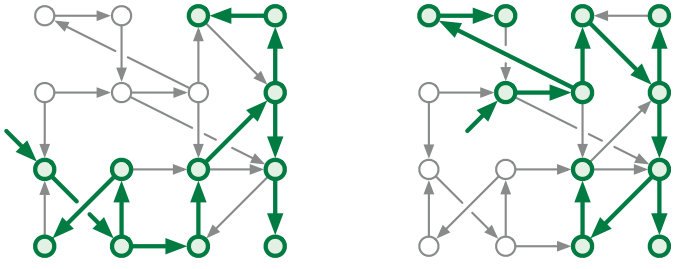


그림 6.1. 같은 방향 그래프에서 서로 다른 정점을 루트로 하는 깊이 우선 트리들입니다.

늘 그렇듯이, 그림 6.2의 왼쪽에 나타낸 표준 래퍼 함수를 사용하면 입력 그래프가 연결되어 있지 않더라도 입력 그래프 *전체*를 순회하도록 도달 가능성 알고리즘을 확장할 수 있습니다. 여기서는 PreVisit 함수와 PostVisit 함수에 필요한 전처리를 수행하기 위해 범용 블랙박스 서브루틴 Preprocess를 추가합니다.

또는 그래프를 수정할 수 있다면, G의 다른 모든 정점으로 향하는 간선을 가진 새로운 *출발* 정점 s를 추가한 뒤 그림 6.2의 오른쪽에 나타낸 것처럼 DFS(s)를 한 번만 호출할 수 있습니다. 이때 결과로 얻는 부모 포인터는 항상 *확장된* 입력 그래프의

```
DFSALL(G):
  PREPROCESS(G)
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          DFS(v)
```

```
DFSALL(G):
  PREPROCESS(G)
  add vertex s
  for all vertices v
      add edge s→v
      unmark v
  DFS(s)
```

그림 6.2. 깊이 우선 탐색을 위한 표준 래퍼 알고리즘의 두 가지 정식화

신장 트리를 정의하지만, *원래의* 입력 그래프의 신장 트리를 정의하지는 않습니다. 그 밖에는 두 래퍼 함수가 본질적으로 동일하게 동작하므로, 어느 쪽을 택할지는 전적으로 편의의 문제입니다.[1]

이번에도 이 알고리즘은 무방향 그래프와 방향 그래프에서 조금 다르게 동작합니다. 앞 장에서 보았듯이, 무방향 그래프에서는 DFSALL을 조금 고쳐 그래프의 연결 성분 개수를 쉽게 셀 수 있습니다. 특히 DFSALL이 계산하는 부모 포인터는 입력 그래프의 신장 숲을 정의하며, 이 신장 숲에는 연결 성분마다 신장 트리가 하나씩 들어 있습니다. 그러나 방향 그래프에서는 그래프가 "연결되어" 있더라도 DFSALL이 1개에서 V개 사이의 몇 개든 "연결 성분"을 발견할 수 있습니다. 그 개수는 그래프의 정확한 구조와 래퍼 알고리즘이 정점을 방문하는 순서에 따라 달라집니다.

6.1 전위 순회와 후위 순회

루트가 있는 *트리*의 전위 순회와 후위 순회는 이미 익숙하시리라 생각합니다. 두 순회 모두 깊이 우선 탐색으로 계산할 수 있습니다. 그림 6.3에서 보듯이 카운터를 전달하는 방식으로 임의의 방향 그래프—연결되어 있지 않더라도—에 대해서도 비슷한 순회 순서를 정의할 수 있습니다. 이와 동등하게, 우리의 일반적인 깊이 우선 탐색 알고리즘에 다음 서브루틴 PREPROCESS, PREVISIT, POSTVISIT을 사용할 수도 있습니다.

```
PREPROCESS(G):
  clock ← 0
```

```
PREVISIT(v):
  clock ← clock + 1
  v.pre ← clock
```

```
POSTVISIT(v):
  clock ← clock + 1
  v.post ← clock
```

두 방식 중 어느 쪽이든 이 알고리즘은 v를 재귀 스택에 푸시한 직후에 $v.pre$를 할당하고(시계도 진행시킵니다), v를 재귀 스택에서 팝하기 직전에 $v.post$를 할당합니다(역시 시계를 진행시킵니다). 따라서 임의의 두 정점 u와 v에 대해 구간 $[u.pre, u.post]$와 $[v.pre, v.post]$는 서로소이거나 한쪽이 다른 쪽에 포함됩니다. 더욱이 $[u.pre, u.post]$가 $[v.pre, v.post]$를 포함하는 것은 DFS(u)를 실행하는 도중에 DFS(v)가 호출될 때, 그리고 오직 그때뿐이며, 이는 부모 포인터로 이루어진 최종 숲에서 u가 v의 조상일 때, 그리고 오직 그때뿐이라는 것과 동치입니다.

[1] 이 두 래퍼 함수가 동등하다는 것은 ***깊이*** 우선 탐색의 고유한 특징입니다. 특히 모든 정점을 방문하기 위해 *너비* 우선 탐색을 for 루프로 감싸는 방식은 출발 정점을 추가하고 s에서 너비 우선 탐색을 호출하는 방식과 같은 순회 순서를 만들어 내지 *않습니다*.

```
DFSALL(G):
    clock ← 0
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            clock ← DFS(v, clock)
```

```
DFS(v, clock):
    mark v
    clock ← clock + 1;  v.pre ← clock
    for each edge v→w
        if w is unmarked
            w.parent ← v
            clock ← DFS(w, clock)
    clock ← clock + 1;  v.post ← clock
    return clock
```

그림 6.3. 깊이 우선 탐색을 이용한 전위 순서와 후위 순서의 정의.

DFSALL이 그래프의 모든 노드에 레이블을 붙이고 나면, 레이블 $v.pre$는 정점들의 ***전위 순서***를 정의하고, 레이블 $v.post$는 정점들의 ***후위 순서***를 정의합니다.[2] 몇 가지 자명한 예외를 제외하면, DFS가 각 정점에서 나가는 간선을 고려하는 순서와 DFSALL이 정점을 고려하는 순서에 따라 모든 그래프에는 서로 다른 전위 순서와 후위 순서가 여러 개 있습니다.

이 장의 나머지 부분에서는 $v.pre$를 v의 *시작 시각*(덜 형식적으로는 "v가 시작할 때"), $v.post$를 v의 *종료 시각*(덜 형식적으로는 "v가 종료할 때"), 그리고 시작 시각과 종료 시각 사이의 구간을 v의 *활성 구간*(덜 형식적으로는 "v가 활성 상태인 동안")이라고 부르겠습니다.

정점과 간선의 분류

DFSALL이 실행되는 동안 입력 그래프의 각 정점 v는 다음 세 가지 상태 중 하나를 가집니다:

- DFS(v)가 아직 호출되지 않았다면, 즉 $clock < v.pre$이면 v는 ***새*** 정점입니다;
- DFS(v)가 호출되었지만 아직 반환되지 않았으면, 즉 $v.pre \le clock < v.post$이면 ***활성*** 상태입니다;
- DFS(v)가 반환되었으면, 즉 $v.post \le clock$이면 ***완료*** 상태입니다.

시작 시각과 종료 시각은 재귀 스택에 대한 푸시 및 팝에 대응하므로, 정점이 활성 상태인 것은 그 정점이 재귀 스택 위에 있을 때이며 오직 그때뿐입니다. 따라서 활성 노드들은 항상 G 안에서 하나의 방향 경로를 이룹니다.

입력 그래프의 간선은 활성 구간이 교차하는 방식에 따라 서로 다른 네 가지 부류로 나뉩니다. 마음에 드는 간선 $u \to v$를 하나 고정합시다.

- DFS(u)가 시작될 때 v가 새 정점이면, DFS(v)는 DFS(u)가 실행되는 동안 직접 또는 몇 단계의 중간 재귀 호출을 거쳐 반드시 호출됩니다. 어느 경우든 u는 깊이 우선 숲에서 v의 진조상이며, $u.pre < v.pre < v.post < u.post$가 성립합니다.
  - DFS(u)가 DFS(v)를 직접 호출하면 $u = v.parent$, 즉 u는 v의 부모이고, $u \to v$를 ***트리 간선***이라고 합니다.

[2] 혼란스럽게도, 이 두 순서를 *둘 다* "깊이 우선 순서"라고 부르는 경우가 있습니다. 그렇게 부르지 마십시오.

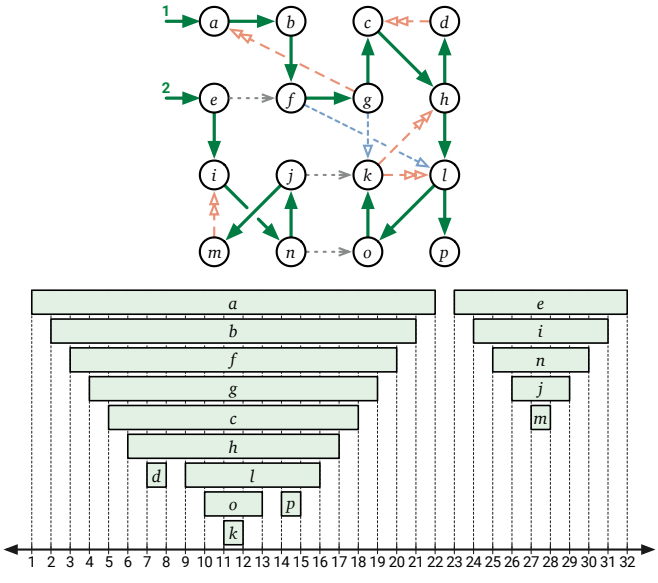


그림 6.4. 방향 그래프의 깊이 우선 숲과 그에 대응하는 정점들의 활성 구간으로, 전위 순서 *abfgchdlokpeinjm*과 후위 순서 *dkoplhcgfbamjnie*를 정의합니다. 숲 간선은 실선으로 나타냈으며, 점선 간선은 그림 6.5에서 설명합니다.

  - 그렇지 않으면 $u \to v$를 ***전진 간선***이라고 부릅니다.

- DFS(u)가 시작될 때 v가 활성 상태라면 v는 이미 재귀 스택에 있으며, 이는 반대 방향의 중첩 순서 $v.pre < u.pre < u.post < v.post$를 뜻합니다. 또한 G에는 v에서 u로 가는 방향 경로가 반드시 있어야 합니다. 이 조건을 만족하는 간선을 ***역방향 간선***이라고 합니다.
- DFS(u)가 시작될 때 v가 이미 완료된 상태라면, 곧바로 $v.post < u.pre$가 성립합니다. 이 조건을 만족하는 간선을 ***교차 간선(cross edge)***이라고 합니다.
- 마지막으로, 네 번째 순서인 $u.post < v.pre$는 불가능합니다.

이러한 간선 분류는 그림 6.5에 나타나 있습니다. 다시 말하지만, 간선이 실제로 어떻게 분류되는지는 DFSALL이 정점을 고려하는 순서와 DFS가 각 정점에서 나가는 간선을 고려하는 순서에 따라 달라집니다.

마지막으로, 다음 핵심 보조정리는 순회하는 동안의 정점 상태를 이용해 임의의 깊이 우선 숲에서 조상과 자손을 특징짓습니다.

보조정리 6.1. *임의의 방향 그래프 G에 대한 임의의 깊이 우선 순회를 하나 고정합니다. G의 모든 정점 u와 v에 대해 다음 명제들은 서로 동치입니다.*

(a) 깊이 우선 숲에서 u는 v의 조상입니다.

(b) $u.pre \le v.pre < v.post \le u.post$.

(c) DFS(v)가 호출된 직후에 u는 활성 상태입니다.

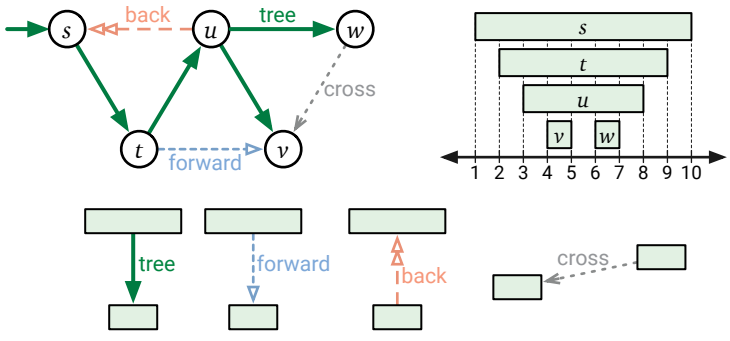


그림 6.5. 깊이 우선 탐색에 의한 간선의 분류.

(d) DFS(u)가 호출되기 직전에, u에서 v로 가는 경로로서 (u와 v를 포함한) 모든 정점이 새 정점인 경로가 존재합니다.

증명: 먼저 깊이 우선 숲에서 u가 v의 조상이라고 가정합시다. 그러면 정의에 따라 u에서 v로 가는 트리 간선들의 경로 P가 존재합니다. 경로 길이에 대한 귀납법에 따라 P의 모든 정점 w에 대해 $u.pre \le w.pre < w.post \le u.post$가 성립합니다. 따라서 DFS($u$)가 호출되기 전에는 P의 모든 정점이 새 상태입니다. 특히 $u.pre \le v.pre < v.post \le u.post$가 성립하며, 이는 DFS($v$)가 실행되는 동안 u가 활성 상태임을 의미합니다.

부모 포인터는 재귀 호출에 대응하므로, $u.pre \le v.pre < v.post \le u.post$이면 u는 v의 조상입니다.

DFS(v)가 호출된 직후에 u가 활성 상태라고 가정합시다. 그러면 $u.pre \le v.pre < v.post \le u.post$이며, 이는 u에서 출발하여 재귀 스택에 있는 (있다면) 중간 노드들을 거쳐 v에 이르는, (0개 이상의) 트리 간선으로 이루어진 경로가 존재함을 뜻합니다.

마지막으로, u가 v의 조상이 아니라고 가정합시다. u에서 v로 가는 임의의 경로 P를 고정하고, x를 P에서 u의 자손이 아닌 첫 번째 정점이라 하고, w를 P에서 x의 바로 앞 정점이라 합시다. 간선 $w \to x$에 의해 $x.pre < w.post$가 보장되고, w가 u의 자손이므로 $w.post < u.post$이며, 따라서 $x.pre < u.post$입니다. 그렇지 않다면 x가 u의 자손이 되므로, $x.pre < u.pre$가 성립합니다. 활성 구간들은 올바르게 중첩되어 있으므로, 가능한 경우는 다음 두 가지뿐입니다:

- $u.post < x.post$이면, DFS(u)가 호출될 때 x는 활성 상태입니다.
- $x.post < u.pre$이면, DFS(u)가 호출될 때 x는 이미 종료된 상태입니다.

따라서 u에서 v로 가는 모든 경로에는 DFS(u)가 호출될 때 새 정점이 아닌 정점이 포함되어 있다고 결론지을 수 있습니다. □

6.2 사이클 탐지

방향 비순환 그래프(directed acyclic graph), 줄여서 ***dag***는 방향 사이클이 없는 방향 그래프입니다. dag에서 들어오는 간선이 없는 정점을 **소스**라고 하고, 나가는 간선이

없는 정점을 *싱크*라고 합니다. 연결된 간선이 전혀 없는 고립 정점은 소스이면서 동시에 싱크입니다. 모든 dag에는 소스와 싱크가 적어도 하나씩 있지만, 각각 둘 이상 있을 수도 있습니다. 예를 들어 정점이 n개이고 간선이 없는 그래프에서는 모든 정점이 소스이고 모든 정점이 싱크입니다.

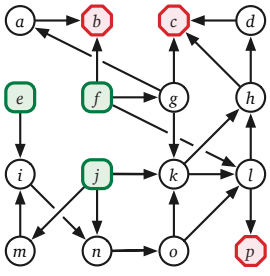


그림 6.6. 방향 비순환 그래프입니다. 정점 e, f, j는 소스이고, 정점 b, c, p는 싱크입니다.

앞서 살펴본 경우 분석에서, 어떤 간선 $u \to v$에 대해 $u.post < v.post$이면 그래프에 v에서 u로 가는 방향 경로가 존재하고, 따라서 간선 $u \to v$를 지나는 방향 *사이클*이 존재한다는 것을 떠올려 보십시오. 그러므로 정점들의 후위 순서를 계산한 다음 각 간선을 전수 조사로 확인하면, 주어진 방향 그래프 G가 dag인지 여부를 $O(V + E)$ 시간에 판별할 수 있습니다.

또는 정점에 번호를 매기는 대신 각 정점의 상태를 명시적으로 유지하다가, 활성 정점으로 향하는 간선을 발견하는 즉시 False를 반환할 수도 있습니다. 이 알고리즘도 $O(V + E)$ 시간에 실행됩니다. 그림 6.7을 참고하십시오.

```
IsAcyclic(G):
  for all vertices v
      v.status ← New
  for all vertices v
      if v.status = New
          if IsAcyclicDFS(v) = False
              return False
  return True
```

```
IsAcyclicDFS(v):
  v.status ← Active
  for each edge v→w
      if w.status = Active
          return False
      else if w.status = New
          if IsAcyclicDFS(w) = False
              return False
  v.status ← Finished
  return True
```

그림 6.7. 그래프가 비순환인지 판별하는 선형 시간 알고리즘입니다.

6.3 위상 정렬

방향 그래프 G의 *위상 순서*란 모든 간선 $u \to v$에 대해 $u \prec v$가 성립하도록 정점들 위에 정한 전순서 $\prec$입니다. 덜 형식적으로 말하면, 위상 순서는 모든 간선이 왼쪽에서 오른쪽을 향하도록 정점들을 하나의 수평선 위에 배열하는 것입니다. 그래프 G에 방향 사이클이 있으면 위상 순서가 불가능하다는 것은 명백합니다—사이클에서 가장 오른쪽에 있는 정점에는 왼쪽을 향하는 간선이 있을 것이기 때문입니다!

한편, 임의의 방향 그래프 G의 임의의 후위 순서를 생각해 봅시다. 앞선 분석에 따르면, 어떤 간선 $u \to v$에 대해 $u.post < v.post$이면 G에는 v에서 u로 가는 방향 경로가 존재하고, 따라서 $u \to v$를 지나는 방향 사이클이 존재합니다. 이를 달리 말하면, G가 비순환이면 모든 간선 $u \to v$에 대해 $u.post > v.post$입니다. 이로부터 모든 방향 비순환 그래프 G에는 위상 순서가 존재한다는 결론이 나옵니다. 특히 G의 임의의 후위 순서를 뒤집은 것은 G의 위상 순서입니다.

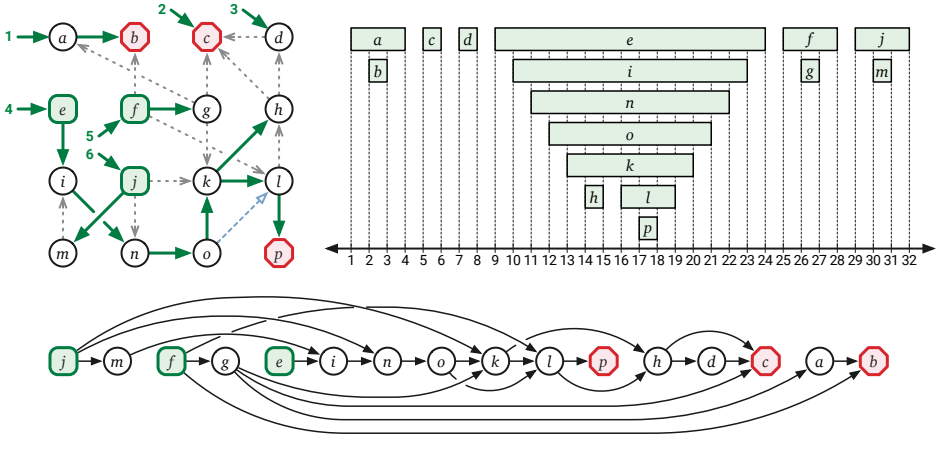


그림 6.8. 그림 6.6에 있는 dag의 역후위 순서입니다.

위상 순서를 별도의 자료구조에 저장해야 한다면, 그림 6.9에 나타낸 것처럼 정점들을 후위 순서의 역순으로 배열에 기록하기만 하면 되며, 이는 $O(V + E)$ 시간에 가능합니다.

암시적 위상 정렬

하지만 위상 순서를 별도의 자료구조에 기록하는 것은 대개 과한 일입니다. 위상 정렬의 대부분의 응용에서 정점들의 정렬된 목록 자체는 우리의 실제 목표가 아닙니다. 그보다는 위상 순서 또는 역위상 순서로 그래프의 각 정점에서 어떤 정해진 계산을 수행하고자 합니다. 이러한 응용에서는 위상 순서를 *기록*할 필요가 전혀 없습니다!

```
TopologicalSort(G):
  for all vertices v
      v.status ← New
  clock ← V
  for all vertices v
      if v.status = New
          clock ← TopSortDFS(v, clock)
  return S[1 .. V]
```

```
TopSortDFS(v, clock):
  v.status ← Active
  for each edge v→w
      if w.status = New
          clock ← TopSortDFS(v, clock)
      else if w.status = Active
          fail gracefully
  v.status ← Finished
  S[clock] ← v
  clock ← clock − 1
  return clock
```

그림 6.9. 명시적 위상 정렬

방향 비순환 그래프를 *역*위상 순서로 처리하려면, 각 정점을 그 정점에 대한 재귀적 깊이 우선 탐색이 끝나는 시점에 처리하면 충분합니다. 따지고 보면 위상 순서는 후위 순서를 뒤집은 것과 같기 때문입니다!

```
PostProcess(G):
  for all vertices v
      v.status ← New
  for all vertices v
      if v is unmarked
          PostProcessDFS(v)
```

```
PostProcessDFS(v):
  v.status ← Active
  for each edge v→w
      if w.status = New
          PostProcessDFS(w)
      else if w.status = Active
          fail gracefully
  v.status ← Finished
  Process(v)
```

입력 그래프에 사이클이 없다는 것을 이미 알고 있다면, 정점의 탐색 상태를 기록하는 대신 정점에 표시만 하면 되므로 알고리즘을 더욱 단순화할 수 있습니다.

```
PostProcessDag(G):
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          PostProcessDagDFS(s)
```

```
PostProcessDagDFS(v):
  mark v
  for each edge v→w
      if w is unmarked
          PostProcessDagDFS(w)
  Process(v)
```

이것은 표준 깊이 우선 탐색 알고리즘에서 PostVisit의 이름을 Process로 바꾼 것일 뿐입니다!

방향 비순환 그래프에서 매우 흔히 쓰이는 연산이므로, 저는 때때로 dag의 후위 순서 처리를 다음과 같이 관용적으로 표현합니다:

```
PostProcessDag(G):
  for all vertices v in postorder
      Process(v)
```

예를 들어, 앞서 살펴본 명시적 위상 정렬 알고리즘은 다음과 같이 작성할 수 있습니다.

```
TopologicalSort(G):
  clock ← V
  for all vertices v in postorder
      S[clock] ← v
      clock ← clock − 1
  return S[1..V]
```

dag를 *정방향* 위상 순서로 처리하려면 정점들의 위상 순서를 배열에 기록한 뒤 간단한 for 반복문을 실행하면 됩니다. 또는 각 $v \to w$를 그 역방향 간선 $w \to v$로 바꾸어 얻은 G의 ***역그래프***, 즉 $\boldsymbol{rev}(\boldsymbol{G})$로 표기하는 그래프에 깊이 우선 탐색을 적용할 수도 있습니다. 방향 사이클을 뒤집으면 방향이 반대인 또 다른 방향 사이클이 되므로,

dag의 역그래프 역시 dag입니다. G의 모든 소스는 $rev(G)$의 싱크이고 그 역도 성립합니다. 따라서 귀납적으로 $rev(G)$의 모든 위상 순서는 G의 어떤 위상 순서를 뒤집은 것입니다.[3] (표준 인접 리스트로 표현된) 임의의 방향 그래프의 역그래프는 $O(V+E)$ 시간에 계산할 수 있습니다. 이 구성의 세부 사항은 쉬운 연습 문제로 남겨 둡니다.

6.4 메모이제이션과 동적 계획법

우리의 위상 정렬 알고리즘은 폭넓은 부류의 동적 계획법 알고리즘의 본보기라고 할 수 있습니다. 점화식의 ***의존성 그래프***는 재귀적 부분 문제마다 정점을 하나씩 가지며, 첫 번째 부분 문제를 계산하는 데 두 번째 부분 문제의 재귀적 계산이 필요하면 한 부분 문제에서 다른 부분 문제로 향하는 간선을 가진다는 점을 떠올려 보십시오. 의존성 그래프는 반드시 비순환이어야 합니다. 그렇지 않으면 단순한 재귀 알고리즘은 결코 멈추지 않을 것입니다.

메모이제이션으로 점화식을 계산하는 것은 그 어떤 점화식이든 의존성 그래프에서 깊이 우선 탐색을 수행하는 것과 *정확히* 같습니다. 특히, 의존성 그래프의 정점은 대응하는 부분 문제의 값이 이미 계산되었으면 "표시"됩니다. 블랙박스 서브루틴 PreVisit과 PostVisit은 실제 값 계산을 대신하는 역할을 합니다. 그림 6.10을 참조하십시오.

```
MEMOIZE(x):
  if value[x] is undefined
      initialize value[x]

      for all subproblems y of x
          MEMOIZE(y)
          update value[x] based on value[y]
      finalize value[x]
```

```
DFS(v):
  if v is unmarked
      mark v
      PREVISIT(x)
      for all edges v→w
          DFS(w)

      POSTVISIT(x)
```

그림 6.10. 메모이제이션된 재귀는 깊이 우선 탐색입니다. 깊이 우선 탐색은 메모이제이션된 재귀입니다.

이 비유를 더 밀고 나가면, 점화식을 *동적 계획법으로* 계산하는 것은 점화식의 의존성 그래프에 있는 모든 부분 문제를 역위상 순서로 계산하는 것과 같습니다—모든 부분 문제는 자신이 의존하는 부분 문제들을 처리한 *후에* 고려됩니다. 따라서 *모든* 동적 계획법 알고리즘은 그 바탕이 되는 점화식의 의존성 그래프를 후위 순회하는 것과 동등합니다!

그러나 대부분의 동적 계획법 알고리즘과 위상 정렬 사이에는 몇 가지 사소한 차이가 있습니다. 첫째, 대부분의 동적 계획법 알고리즘에서 의존성 그래프는 *암묵적*입니다—노드와 간선이 메모리에 명시적으로 저장되는 것이 아니라, 그 바탕에 있는 점화식에 의해 인코딩됩니다. 하지만 이 차이는 정말로 사소합니다. 재귀적 부분 문제를 각각 상수 시간에 열거할 수 있기만 하면, 의존성 그래프가 인접 리스트에 명시적으로 저장되어 있는 *것처럼* 정확히 그대로 의존성 그래프를 순회할 수 있습니다.

[3] G의 역그래프의 후위 순서와 G의 후위 순서를 뒤집은 것은 둘 다 G의 위상 순서이지만, 반드시 서로 같지는 않습니다.

```
DynamicProgramming(G) :
  for all subproblems x in postorder
      initialize value[x]
      for all subproblems y of x
          update value[x] based on value[y]
      finalize value[x]
```

그림 6.11. 동적 계획법은 후위 순회입니다.

더 중요한 점은 대부분의 동적 계획법 점화식이 고도로 구조화된 의존성 그래프를 가진다는 것입니다. 예를 들어 5장에서 논의했듯이, 편집 거리 점화식의 의존성 그래프는 대각선이 있는 규칙적인 격자이고, 최적 이진 탐색 트리의 의존성 그래프는 오른쪽과 위쪽으로 향하는 가능한 모든 간선을 가진 상삼각 격자입니다. 이러한 규칙적인 구조 덕분에 적절한 평가 순서를 보통 중첩 반복문의 모음 형태로 알고리즘에 직접 고정해 넣을 수 있으므로, 실행 시점에 의존성 그래프를 위상 정렬할 필요가 없습니다. 앞에서 우리는 역위상 순서를 *평가 순서*라고 불렀습니다.

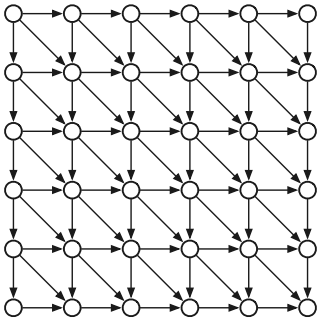


그림 6.12. 편집 거리 점화식의 의존성 **방향 비순환 그래프(dag)**입니다.

DAG에서의 동적 계획법

반대로, 의존성 그래프가 덜 구조화된 문제에서도 깊이 우선 탐색을 사용하여 동적 계획법 알고리즘을 만들 수 있습니다. 예를 들어 ***최장 경로*** 문제를 생각해 봅시다. 이 문제는 간선에 가중치가 있는 방향 그래프 G에서 한 노드 s로부터 다른 노드 t까지 가는 경로 중 총 가중치가 *최대*인 경로를 구하는 문제입니다. 일반적인 방향 그래프에서 최장 경로 문제는 NP-난해입니다(외판원 문제로부터의 간단한 환원으로 보일 수 있습니다. 12장을 참고하십시오). 그러나 입력 그래프 G가 비순환 그래프라면 이 문제는 쉬워지며, 다음과 같이 G의 최장 경로를 선형 시간에 계산할 수 있습니다.

대상 정점 t를 고정하고, 임의의 노드 v에 대해 $\boldsymbol{LLP(v)}$가 G에서 v부터 t까지의 최장 경로의 길이(Length of the Longest Path)를 나타낸다고 합시다. G가 dag이면 이 함수는 다음 점화식을 만족합니다.

$$LLP(v) = \begin{cases} 0 & v = t\text{이면,} \\ \max\{\ell(v \to w) + LLP(w) \mid v \to w \in E\} & \text{그렇지 않으면,} \end{cases}$$

여기서 $\ell(v \to w)$는 간선 $v \to w$에 주어진 가중치("길이")를 나타내며, $\max \varnothing = -\infty$입니다. 특히 v가 *싱크*이지만 t와 같지 않으면 $LLP(v) = -\infty$입니다.

이 점화식의 의존성 그래프는 입력 그래프 G 자체입니다. 즉, 부분 문제 $LLP(v)$가 부분 문제 $LLP(w)$에 의존하는 것은 $v \to w$가 G의 간선일 때이며, 오직 그때뿐입니다. 따라서 s에서 시작하여 G를 깊이 우선 탐색하면 이 재귀 함수를 $O(V + E)$ 시간에 계산할 수 있습니다. 이 알고리즘은 각 길이 $LLP(v)$를 해당 노드 v의 추가 필드에 메모이제이션합니다.

```
LongestPath(v, t):
  if v = t
      return 0
  if v.LLP is undefined
      v.LLP ← −∞
      for each edge v→w
          v.LLP ← max{v.LLP, ℓ(v→w) + LongestPath(w, t)}
      return v.LLP
```

원칙적으로, 이 메모이제이션된 재귀 알고리즘은 위상 정렬을 통해 동적 계획법 알고리즘으로 변환할 수 있습니다:

```
LongestPath(s, t):
  for each node v in postorder
      if v = t
          v.LLP ← 0
      else
          v.LLP ← −∞
          for each edge v→w
              v.LLP ← max{v.LLP, ℓ(v→w) + w.LLP}
  return s.LLP
```

두 알고리즘은 사실상 동일하다고 볼 수 있습니다—첫 번째 알고리즘의 재귀와 두 번째 알고리즘의 for 루프는 "같은" 깊이 우선 탐색을 나타냅니다! 두 방식 중 어느 쪽을 택할지는 전적으로 편의의 문제입니다.

최적의 결정 *순서*를 구하는 거의 모든 동적 계획법 문제는 이와 연관된 어떤 dag에서 최적 *경로*를 찾는 문제로 바꾸어 표현할 수 있습니다. 예를 들어, 제 2장과 제 3장에서 살펴본 텍스트 분할, 부분집합 합, 최장 증가 부분 수열, 편집 거리 문제는 모두 dag에서 최장 경로나 최단 경로를 찾는 문제로 다시 정식화할 수 있으며, 이때 정점이나 간선에 가중치가 있을 수도 있습니다. 각 경우에 해당 dag는 바탕이 되는 점화식의 의존성 그래프입니다. 반면, 최적 이진 탐색 트리나 트리의 최대 독립 집합을 찾는 것과 같은 "트리 모양" 동적 계획법 문제는 dag에서 최적 *경로*를 찾는 문제로 바꾸어 표현할 수 없습니다.

6.5 강한 연결성

방향 그래프에서 연결성의 올바른 정의로 돌아가 봅시다. 방향 그래프 G에 u에서 v로 가는 방향 경로가 있으면 G에서 한 정점 u가 다른 정점 v에 도달할 수 있다고 하며,

$reach(u)$는 u가 도달할 수 있는 모든 정점의 집합을 나타낸다는 것을 떠올려 보십시오. 두 정점 u와 v는 u가 v에 도달할 수 있고 v가 u에 도달할 수 있을 때 ***강하게 연결되어*** 있다고 합니다. 방향 그래프는 모든 정점 쌍이 강하게 연결되어 있을 때, 그리고 오직 그때에만 강하게 연결되어 있습니다.

지루하게 정의를 따라가 보면, 무방향 그래프에서의 연결성과 마찬가지로 강한 연결성이 임의의 방향 그래프의 정점 집합 위에서 동치 관계임을 알 수 있습니다. 이 관계의 동치류를 G의 ***강하게 연결된 요소***—또는 더 간단히 ***강한 연결 요소***—라고 부릅니다. 동등하게, G의 강한 연결 요소는 G의 극대 강하게 연결된 부분 그래프입니다. 방향 그래프 G가 강하게 연결되어 있을 필요충분조건은 G가 정확히 하나의 강한 연결 요소를 가지는 것입니다. 반대쪽 극단에서, G가 dag일 필요충분조건은 G의 모든 강한 연결 요소가 정점 하나로 이루어지는 것입니다.

강한 연결 요소 그래프 $\boldsymbol{scc(G)}$는 G의 각 강한 연결 요소를 하나의 정점으로 축약하고 평행 간선들을 하나로 합쳐서 얻는 또 다른 방향 그래프입니다. (강한 연결 요소 그래프는 G의 *메타 그래프* 또는 *응축*이라고 부르기도 합니다.) $scc(G)$가 항상 dag임을 증명하는 것은 어렵지 않습니다(힌트, 힌트). 따라서 적어도 원리적으로는 G의 강한 연결 요소들을 위상 정렬할 수 있습니다. 즉, 모든 *역방향* 간선이 같은 강한 연결 요소 안의 두 간선을 잇도록 정점들을 정렬할 수 있습니다.

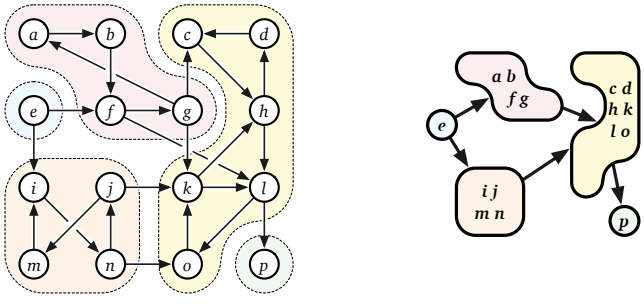


그림 6.13. 그래프 G의 강한 연결 요소와 강한 연결 요소 그래프 $scc(G)$.

정점 하나 v의 강한 연결 요소는 $O(V+E)$ 시간에 간단히 계산할 수 있습니다. 먼저 아무거나 우선 탐색으로 $reach(v)$를 계산합니다. 그다음 G의 역그래프를 탐색하여 $reach^{-1}(v) = \{u \mid v \in reach(u)\}$를 계산합니다. 마지막으로, v의 강한 연결 요소는 교집합 $reach(v) \cap reach^{-1}(v)$입니다. 특히, 그래프 전체가 강하게 연결되어 있는지도 $O(V+E)$ 시간에 판정할 수 있습니다.

마찬가지로, 앞의 알고리즘을 표준 래퍼 함수와 결합하면 방향 그래프의 모든 강한 연결 요소를 계산할 수 있습니다. 그러나 이렇게 얻은 알고리즘은 $O(VE)$ 시간에 실행됩니다. 강한 연결 요소는 최대 V개이고, 그래프가 dag인 경우에도 각 요소를 발견하는 데 $O(E)$ 시간이 걸리기 때문입니다. 분명 이보다 더 잘할 수 있을 것입니다! 어쨌든 모든 강한 연결 요소가 단일 정점인지 판정하는 데에는 $O(V+E)$ 시간만 있으면 됩니다.

6.6 선형 시간에 강한 연결 요소 찾기

사실 강한 연결 요소를 $O(V + E)$ 시간에 계산하는 알고리즘은 여러 가지가 있으며, 이들은 모두 다음 관찰에 기반합니다.

보조정리 6.2. *임의의 방향 그래프 G에 대한 깊이 우선 순회를 하나 고정합시다. G의 각 강한 연결 요소 C에는 C 안에 부모가 없는 노드가 정확히 하나 있습니다. (이 노드는 다른 강한 연결 요소에 부모가 있거나, 부모가 아예 없습니다.)*

증명: C를 G의 임의의 강한 연결 요소라고 합시다. C에 속한 한 정점 $v \in C$에서 다른 정점 $w \in C$로 가는 임의의 경로를 생각해 봅시다. 이 경로 위의 모든 정점은 w에 도달할 수 있으므로 C의 모든 정점에 도달할 수 있습니다. 대칭적으로, 이 경로 위의 모든 노드는 v로부터 도달 가능하므로 C의 모든 정점으로부터 도달 가능합니다. 따라서 이 경로 위의 모든 정점 또한 C에 속한다고 결론지을 수 있습니다.

C에서 시작 시각이 가장 이른 정점을 v라고 합시다. v에 부모가 있다면 $parent(v)$는 v보다 먼저 시작하므로 C에 속할 수 없습니다.

이제 w를 C에 속한 다른 정점이라고 합시다. DFS(v)가 호출되기 직전에는 C의 모든 정점이 새 정점이므로, v에서 w로 가는 새 정점들의 경로가 존재합니다. 따라서 보조정리 6.1에 의해 w는 깊이 우선 숲에서 v의 자손입니다. v에서 w로 가는 트리 간선 경로 위의 모든 정점은 C에 속합니다. 특히 w의 부모도 C에 속하므로 $parent(w) \in C$입니다. □

앞의 보조정리에 따르면 방향 그래프 G의 각 강한 연결 요소는 G의 모든 깊이 우선 숲에서 연결된 부분 트리를 이룹니다. 특히 임의의 강한 연결 요소 C에 대해, C에서 시작 시각이 가장 이른 정점은 C의 모든 정점의 최소 공통 조상입니다. 이 정점을 C의 **루트**라고 부릅니다.

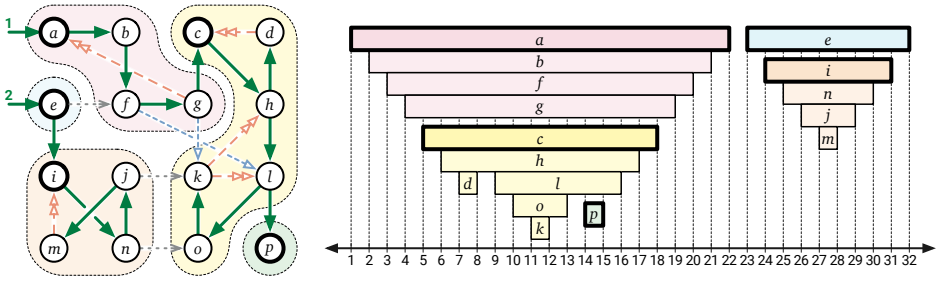


그림 6.14. 강한 연결 요소는 깊이 우선 숲에서 연속적으로 나타납니다.

두 가지 알고리즘을 소개하겠습니다. 두 알고리즘은 모두 같은 직관적인 개요를 따릅니다. C를 $scc(G)$에서 싱크인 G의 임의의 강한 연결 요소라고 합시다. 이러한 C를 ***싱크 요소***라고 부릅니다. 동치로, C에 속한 임의의 정점의 도달 범위가 정확히 C이면 C는 싱크 요소입니다. 어떤 싱크 요소에 속한 정점 v를 (어떻게든) 찾고, v에서 도달 가능한 정점들을 찾은 다음, 그 싱크 요소를 입력 그래프에서 제거하는 과정을 정점이 하나도 남지 않을 때까지 반복하면 G의 모든 강한 연결 요소를 찾을 수 있습니다. 하지만 싱크 요소에 속한 정점을 어떻게 찾아야 할지 분명하지 않으므로, 이것은 아직 완전한 알고리즘이라고 할 수 없습니다!

```
StrongComponents(G):
  count ← 0
  while G is non-empty
      C ← ∅
      count ← count + 1
      v ← any vertex in a sink component of G    《마법입니다!》
      for all vertices w in reach(v)
          w.label ← count
          add w to C
      remove C and its incoming edges from G
```

그림 6.15. 강한 연결 요소를 계산하는, 알고리즘에 거의 가까운 방법입니다.

코사라주와 샤리르의 알고리즘

언뜻 보면 싱크 요소에 속한 정점을 *빠르게* 찾는 것은 꽤 어려워 보입니다. 하지만 깊이 우선 탐색을 사용하면 소스 요소, 즉 $scc(G)$의 소스에 대응하는 G의 강한 연결 요소에 속한 정점은 실제로 꽤 쉽게 찾을 수 있습니다.

보조정리 6.3. *G의 임의의 후위 순서에서 마지막 정점은 G의 소스 컴포넌트에 속합니다.*

증명: G의 깊이 우선 순회를 하나 고정하고, 그 결과로 얻은 후위 순서에서 마지막 정점을 v라고 합시다. 그러면 DFS(v)는 래퍼 알고리즘 DFSALL이 DFS를 직접 호출하는 마지막 호출이어야 합니다. 또한 v는 깊이 우선 숲에 속한 트리 중 하나의 루트이므로, $x.post > v.pre$를 만족하는 모든 노드 x는 v의 자손입니다. 마지막으로, v는 자신이 속한 강한 연결 요소 C의 루트입니다.

논의를 위해 $x \notin C$이고 $y \in C$인 간선 $x \to y$가 존재한다고 가정합시다. 그러면 x는 y에 도달할 수 있고 y는 v에 도달할 수 있으므로, x는 v에 도달할 수 있습니다. v는 C의 루트이므로 정점 y는 v의 자손이며, 따라서 $v.pre < y.pre$입니다. 간선 $x \to y$에 의해 $y.pre < x.post$가 보장되므로 $v.pre < x.post$입니다. 이로부터 x가 v의 자손임이 따라 나옵니다. 그런데 그렇다면 v는 (트리 간선을 통해) x에 도달할 수 있으며, 이는 $x \notin C$라는 가정에 모순됩니다. □

모든 방향 그래프 G에 대해 $rev(scc(G)) = scc(rev(G))$임은 쉽게 확인할 수 있습니다(힌트, 힌트). 따라서 $rev(G)$의 후위 순서에서 *마지막* 정점은 원래 그래프 G의 *싱크* 요소에 속합니다. 따라서 래퍼 함수가 $rev(G)$의 역 후위 순서를 따르도록 하여 그래프를 한 번 더 순회하면, DFS를 호출할 때마다 G의 강한 연결 요소를 정확히 하나씩 방문합니다.[4]

지금까지의 내용을 모두 종합하면 그림 6.16에 제시된 알고리즘을 얻습니다. 이 알고리즘은 임의의 방향 그래프의 강한 연결 요소를 $O(V + E)$ 시간에 세고 레이블을 붙입니다. 이 알고리즘은 1978년 Rao Kosaraju가 발견했지만 출판하지는 않았으며, 이후 1981년 Micha Sharir가 독립적으로 재발견했습니다.[5] Kosaraju-Sharir 알고리

[4] 다시 말하지만, $rev(G)$의 역 후위 순서는 G의 후위 순서와 같지 않습니다.

[5] 같은 알고리즘이 Kosaraju보다도 앞서 러시아 문헌에 등장한다는 소문이 있지만, 아직 그 소문을 뒷받침할 신뢰할 만한 출처는 찾지 못했습니다.

즘은 두 단계로 이루어집니다. 첫 번째 단계에서는 $rev(G)$에 대해 깊이 우선 탐색을 수행하면서, 각 정점의 처리가 끝날 때 그 정점을 스택에 넣습니다. 두 번째 단계에서는 원래 그래프 G에 대해 *아무거나* 우선 순회를 수행하는데, 이때 정점들을 스택에 나타나는 순서대로 살펴봅니다. 이 알고리즘은 각 정점에 (두 번째 깊이 우선 순회를 기준으로 한) 그 정점이 속한 강한 연결 요소의 루트를 레이블로 붙입니다.

그림 6.17은 예제 그래프에서 Kosaraju-Sharir 알고리즘이 실행되는 모습을 보여줍니다. 알고리즘을 조금만 수정하면 강한 연결 요소 그래프 $scc(G)$도 $O(V + E)$ 시간에 계산할 수 있습니다.

```
KosarajuSharir(G):
  S ← new empty stack
  for all vertices v
      unmark v
      v.root ← None
《1단계: rev(G)에서 후위 순서로 스택에 넣기》
  for all vertices v
      if v is unmarked
          PushPostRevDFS(v, S)
《2단계: 스택 순서대로 다시 DFS 수행》
  while S is non-empty
      v ← Pop(S)
      if v.root = None
          LabelOneDFS(v, v)
```

```
PushPostRevDFS(v, S):
  mark v
  for each edge u→v    《뒤집혔습니다!》
      if u is unmarked
          PushPostRevDFS(u, S)
  Push(v, S)
```

```
LabelOneDFS(v, r):
  v.root ← r
  for each edge v→w
      if w.root = None
          LabelOneDFS(w, r)
```

그림 6.16. 코사라주-샤리르 강한 연결 요소 알고리즘

♥Tarjan 알고리즘

강한 연결 요소를 계산하는, 이보다 앞서 나왔지만 훨씬 더 미묘한 선형 시간 알고리즘이 1972년 밥 Tarjan에 의해 발표되었습니다.[6] 직관적으로, Tarjan의 알고리즘은 G의 소스 요소를 찾아 그것을 "삭제"한 다음, 나머지 강한 연결 요소를 "재귀적으로" 찾습니다. 그러나 전체 계산은 단 한 번의 깊이 우선 탐색 동안 이루어집니다.

어떤 방향 그래프 G의 임의의 깊이 우선 순회를 하나 고정합니다. 각 정점 v에 대해, $\boldsymbol{low(v)}$를 v에서 트리 간선으로 이루어진 경로를 따라간 뒤 *최대 한 개*의 비트리 간선을 거쳐 도달할 수 있는 모든 정점 중 가장 작은 *시작 시각*으로 정의합니다. v는 0개의 트리 간선과 0개의 비트리 간선을 거쳐 자기 자신에 도달할 수 있으므로, 자명하게 $low(v) \le v.pre$입니다. Tarjan은 싱크 요소를 이 low 함수로 특징지을 수 있다는 것을 관찰했습니다.

보조정리 6.4. *정점 v가 G의 싱크 요소의 루트일 필요충분조건은 $low(v) = v.pre$이고 v의 모든 진자손 w에 대해 $low(w) < w.pre$인 것입니다.*

[6] 전설에 따르면, Kosaraju는 알고리즘 강의 *도중에* 자신의 알고리즘을 발견했다고 합니다. 그는 Tarjan의 알고리즘을 발표하기로 되어 있었지만 노트를 잊어버리는 바람에 즉석에서 다른 것을 만들어 내야 했습니다. 이 이야기에서 제가 유일하게 놀랍다고 느끼는 점은 아무도 이 이야기를 Sharir나 Tarjan에 대해서는 하지 않는다는 것입니다.

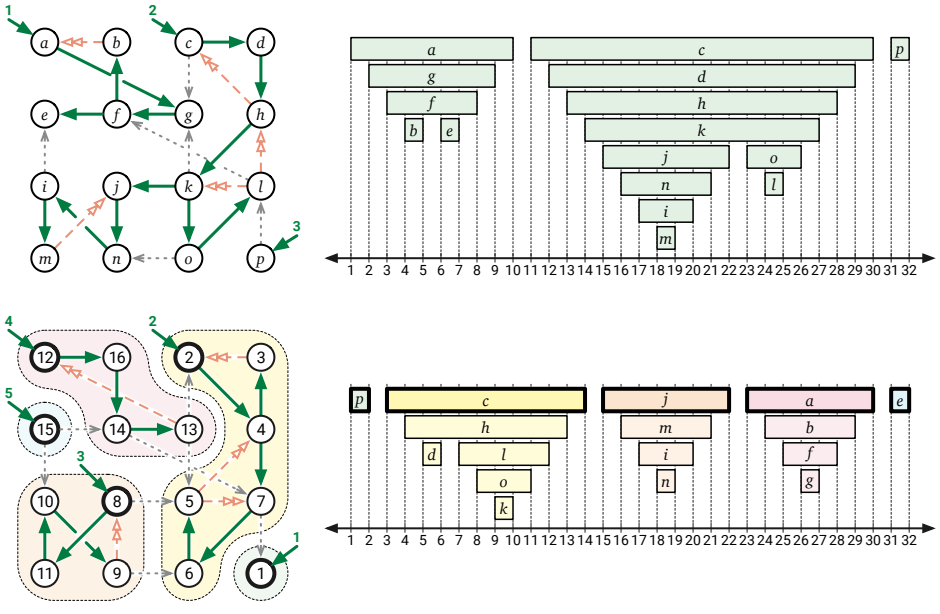


그림 6.17. Kosaraju-Sharir 알고리즘의 동작 과정입니다. 위: 역방향 그래프에 대한 깊이 우선 순회입니다. 아래: 원래 그래프에 대한 깊이 우선 순회로, 첫 번째 순회에서 얻은 후위 순서의 역순으로 루트 정점들을 방문합니다.

증명: 먼저 v를 $low(v) = v.pre$인 정점이라고 합시다. 그러면 w가 v의 자손이고 $x.pre < v.pre$인 간선 $w \to x$는 존재하지 않습니다. 한편 v는 $y.pre > v.post$인 어떤 정점 y에도 도달할 수 없습니다. 따라서 v는 자신의 자손에만 도달할 수 있으며, 그러므로 v의 모든 자손도 v의 자손에만 도달할 수 있습니다. 특히 v는 (부모가 있다면) 자신의 부모에 도달할 수 없으므로, v는 자신이 속한 강한 연결 요소의 루트입니다.

이제 추가로 v의 모든 후손 w에 대해 $low(w) < w.pre$라고 가정합시다. 그러면 각 후손 w는 $x.pre < w.pre$를 만족하는 다른 정점 x(이 정점은 v의 또 다른 후손이어야 합니다)에 도달할 수 있습니다. 따라서 귀납법에 의해 v의 모든 후손은 v에 도달할 수 있습니다. 그러므로 v의 후손들은 v를 루트로 하는 강한 연결 요소 C를 이룹니다. 더욱이 v는 C 바깥의 어떤 정점에도 도달할 수 없으므로 C는 싱크 요소여야 합니다.

반면에 v가 싱크 요소 C의 루트라고 가정합시다. 그러면 v가 다른 정점 w에 도달할 수 있는 것은 $w \in C$일 때, 그리고 오직 그때뿐입니다. 그런데 v는 자신의 모든 자손에 도달할 수 있고, C의 모든 정점은 v의 자손이므로, v의 자손들이 C를 이룹니다. 만약 다른 어떤 노드 $w \in C$에 대해 $low(w) = w.pre$라면, w는 C의 또 다른 루트가 될 것인데, 이는 불가능합니다. □

깊이 우선 탐색을 이용하면 모든 정점 v에 대해 $low(v)$를 간단하게 계산할 수 있습니다. 그림 6.18을 참고하십시오.

보조정리 6.4에 따르면, FindLow를 실행한 후 (전역 아무거나 우선 탐색을 통해) $O(V + E)$ 시간에 모든 싱크 요소의 루트를 식별할 수 있고, 이어서 (각 루트에서 아무거나 우선 탐색을 호출하여) $O(V + E)$의 추가 시간에 그 싱크 요소들을 표시하고

삭제한 다음 재귀할 수 있습니다. 안타깝게도 이렇게 얻은 알고리즘은 각각 정점 하나만 제거하는 반복을 V번 수행해야 할 수도 있으므로, 단순하게 계산하면 전체 실행 시간은 $O(VE)$가 됩니다.

```
FindLow(G):
  clock ← 0
  for all vertices v
      unmark v
      for all vertices v
          if v is unmarked
              FindLowDFS(v)
```

```
FindLowDFS(v):
  mark v
  clock ← clock + 1
  v.pre ← clock
  v.low ← v.pre
  for each edge v→w
      if w is unmarked
          FindLowDFS(w)
          v.low ← min{v.low, w.low}
      else
          v.low ← min{v.low, w.pre}
```

그림 6.18. 모든 정점 v에 대해 $low(v)$를 계산합니다.

이 전략의 속도를 높이기 위해, Tarjan의 알고리즘은 (재귀 스택과는 별도로) 정점들을 담는 보조 스택을 유지합니다. 새로운 정점 v를 시작할 때마다 그 정점을 스택에 넣습니다. 정점 v를 마칠 때마다 $v.low$를 $v.pre$와 비교합니다. 그러면 $v.low = v.pre$임을 *처음* 발견하는 순간, 우리는 다음 세 가지를 알게 됩니다.

- 정점 v는 싱크 요소 C의 루트입니다.
- C의 모든 정점은 보조 스택의 맨 위에 연속으로 나타납니다.
- 보조 스택에 있는 C의 정점 중 *가장 깊은* 정점은 v입니다.

이 시점에서 보조 스택에서 정점을 하나씩 꺼내다가 v를 꺼내면 멈추는 방식으로 C에 속한 정점들을 식별할 수 있습니다.

C에 속한 정점들을 삭제하고 남은 그래프의 강한 연결 요소를 재귀적으로 계산할 수도 있습니다. 그러나 그렇게 하면 v를 방문하기 전에 수행한 모든 계산을 *그대로* 반복하게 되므로 낭비입니다. 대신 C에 속한 각 정점에 *레이블*을 붙여 v를 그 강한 연결 요소의 루트로 지정하고, 이후 깊이 우선 탐색의 나머지 과정에서는 레이블이 붙은 정점을 무시합니다. 형식적으로 말하면, 이 수정으로 $low(v)$의 정의가 바뀝니다. 새 정의는 트리 간선으로 이루어진 경로를 따라간 뒤 비트리 간선을 최대 하나 거쳐 v가 도달할 수 있는 정점 중 ***v와 같은 강한 연결 요소에 속한*** 모든 정점의 시작 시간 가운데 가장 작은 값입니다. 하지만 정확성을 증명할 때는 레이블이 붙은 정점을 무시하면 알고리즘이 그 정점들을 실제로 삭제했을 때와 정확히 같게 동작한다는 점을 관찰하는 편이 더 쉽습니다.

마지막으로, Tarjan의 알고리즘은 그림 6.19에 나와 있으며, FindLow(그림 6.18)에서 필요한 수정 사항은 굵은 빨간색으로 표시되어 있습니다. 알고리즘의 실행 시간은 두 부분으로 나눌 수 있습니다. 각 정점은 S에 한 번 push되고 S에서 한 번 pop되므로, 보조 스택(빨간색 부분)을 유지하는 데 드는 총 시간은 $O(V)$입니다. 보조 스택 유지를 무시하면, 알고리즘의 나머지 부분은 표준적인 깊이 우선 탐색일 뿐입니다. 따라서 이 알고리즘은 ***O(V + E) 시간***에 실행된다는 결론을 내릴 수 있습니다.

```
TARJAN(G):
  clock ← 0
  S ← new empty stack
  for all vertices v
      unmark v
      v.root ← NONE
  for all vertices v
      if v is unmarked
          TARJANDFS(v)
```

```
TARJANDFS(v):
  mark v
  clock ← clock + 1
  v.pre ← clock
  v.low ← v.pre
  PUSH(S, v)
  for each edge v→w
      if w is unmarked
          TARJANDFS(w)
          v.low ← min{v.low, w.low}
      else if w.root = NONE
          v.low ← min{v.low, w.pre}
  if v.low = v.pre
      repeat
          w ← POP(S)
          w.root ← v
      until w = v
```

그림 6.19. Tarjan의 강한 연결 요소 알고리즘.

연습문제

깊이 우선 탐색, 위상 정렬, 강한 연결 요소

0. (a) 방향 그래프의 역 $rev(G)$를 $O(V+E)$ 시간에 계산하는 알고리즘을 설명하십시오.

   (b) 모든 방향 그래프 G에 대해 강한 연결 요소 그래프 $scc(G)$가 비순환임을 증명하십시오.

   (c) 모든 방향 그래프 G에 대해 $scc(rev(G)) = rev(scc(G))$임을 증명하십시오.

   (d) 임의의 방향 그래프 G를 고정합니다. G의 임의의 정점 v에 대해, $S(v)$는 v를 포함하는 G의 강한 연결 요소를 나타낸다고 합시다. G의 모든 정점 u와 v에 대해, G에서 u가 v에 도달할 수 있는 것은 $scc(G)$에서 $S(u)$가 $S(v)$에 도달할 수 있는 것과 필요충분조건임을 증명하십시오.

1. 방향 그래프 G의 모든 정점 쌍 u와 v에 대해 v에서 u에 도달할 수 있거나 u에서 v에 도달할 수 있으면(또는 둘 다 가능하면), G를 *준연결*(semi-connected) 그래프라고 합니다.

   (a) 유일한 소스(source)를 가지지만 반연결(semi-connected)이 *아닌* 방향 비순환 그래프의 예를 드십시오.

   (b) 주어진 방향 *비순환* 그래프가 반연결(semi-connected)인지 판별하는 알고리즘을 설명하고 분석하십시오.

   (c) 임의의 방향 그래프가 반연결(semi-connected)인지 판별하는 알고리즘을 설명하고 분석하십시오.

2. Sham-Poobanana 시 경찰청은 시내의 모든 도로를 일방통행으로 지정했습니다. 혼란에 빠진 운전자들의 불만이 빗발치는데도 시장은 Sham-Poobanana의 어느 교차로에서든 다른 어느 교차로로든 합법적으로 운전해서 갈 수 있다고 주장합니다.

   (a) 시는 시장의 주장이 참인지 검증하거나 반박해야 합니다. 이 문제를 그래프 관점에서 정형화한 다음, 이를 해결하는 알고리즘을 설명하고 분석하십시오.

   (b) (a)의 알고리즘을 실행해 본 뒤, 시장은 자신이 ~~거짓말을 했다는~~ 잘못 알고 있었다는 사실을 마지못해 인정합니다. 교차로 x에서 합법적으로 도달할 수 있는 모든 교차로 y에 대해 y에서 다시 x로 합법적으로 운전해 돌아올 수 있다면, 교차로 x를 *좋은* 교차로라고 부릅시다. 이제 시장은 Sham-Poobanana의 교차로 중 95% 이상이 좋은 교차로라고 주장합니다. 시장의 주장을 검증하거나 반박하는 효율적인 알고리즘을 설명하고 분석하십시오.

   만점을 받으려면 두 알고리즘 모두 선형 시간에 실행되어야 합니다.

3. 유일한 소스 s와 유일한 싱크 t를 가진 방향 비순환 그래프 G가 주어졌다고 가정합시다. 정점 $v \notin \{s, t\}$는 s에서 t로 가는 모든 경로가 v를 지나는 경우, 또는 동치로 v를 삭제하면 s에서 t에 도달할 수 없게 되는 경우 ***(s,t)-단절 정점***이라고 부릅니다. G의 모든 (s,t)-단절 정점을 찾는 알고리즘을 설명하고 분석하십시오.

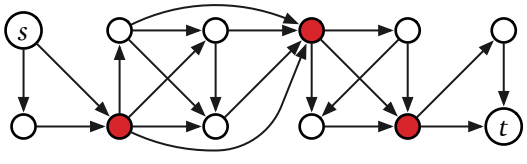


그림 6.20. (s,t)-컷 정점이 세 개 있는 방향 비순환 그래프입니다.

4. 연결된 무방향 그래프 G의 정점 v에 대해, (G에서 v를 제거하여 얻은) 부분 그래프 $G - v$가 연결되어 있지 않으면 v를 ***단절 정점***이라고 합니다.

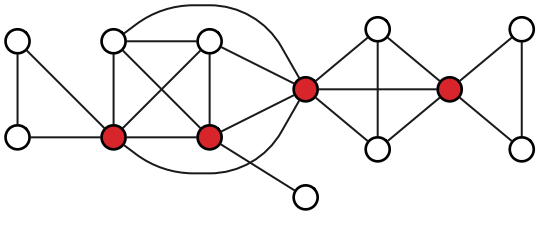


그림 6.21. 단절 정점이 네 개인 무방향 그래프입니다.

   (a) 무방향 그래프 G와 정점 v가 주어졌을 때, v가 G의 단절 정점인지 판정하는 선형 시간 알고리즘을 설명하십시오. 각 정점에 대해 이 알고리즘을 시도하여 모든 단절 정점을 찾는 데 걸리는 실행 시간은 얼마입니까?

   (b) T를 무방향 그래프 G의 깊이 우선 신장 트리라고 합시다.

i. T의 루트가 G의 단절 정점일 필요충분조건은 루트가 T에서 두 개 이상의 자식을 갖는 것임을 증명하십시오.

ii. 루트가 아닌 정점 v가 G의 단절 정점일 필요충분조건은 (T에서) v의 각 자식마다 그 자식의 (T에서의) 자손 중 적어도 하나가 (T에서) v의 어떤 진조상과 (G에서) 이웃이라는 것임을 증명하십시오.

[힌트: T가 깊이 우선 신장 트리가 아니거나 G가 방향 그래프이면 이 주장들은 더 이상 성립하지 않습니다.]

(c) 주어진 무방향 그래프에서 모든 단절 정점을 $O(V + E)$ 시간에 찾아내는 알고리즘을 설명하십시오.

5. 연결된 무방향 그래프 G의 간선 e에 대해, (G에서 e를 제거하여 얻은) 부분 그래프 $G - e$가 연결되어 있지 않으면 e를 *다리*(또는 *단절 간선*)라고 합니다.

(a) G와 간선 e가 주어졌을 때, e가 다리인지 아닌지를 판별하는 선형 시간 알고리즘을 설명하십시오. 각 간선마다 여러분의 알고리즘을 시도하여 모든 다리를 찾는 데 드는 실행 시간은 얼마입니까?

(b) T를 G의 임의의 신장 트리라고 합시다. G의 모든 다리가 T의 간선이기도 함을 증명하십시오. 이 주장으로부터 G에는 다리가 최대 $V - 1$개 존재함을 알 수 있습니다. 이 정보를 이용하면 모든 다리를 찾는 (a)의 알고리즘을 어떻게 개선할 수 있습니까?

(c) 이제 T의 루트를 임의의 정점 r로 정한다고 가정합시다. 임의의 정점 v에 대해 v를 루트로 하는 T의 부분 트리를 T_v로 나타냅시다. 예를 들어 $T_r = T$입니다. uv를 T의 임의의 간선이라 하고, u는 v의 부모라고 합시다. uv가 G의 다리일 필요충분조건은 uv가 T_v에 정확히 한 끝점을 갖는 G의 유일한 간선이라는 것임을 증명하십시오.

(d) G의 모든 다리를 찾는 선형 시간 알고리즘을 서술하십시오. *[힌트: T를 G의 깊이 우선 신장 트리라고 합시다.]*

6. 방향 그래프 G의 ***추이적 폐포*** G^T는 G와 같은 정점들을 가지는 방향 그래프로서, G에 u에서 v로 가는 방향 경로가 있을 때, 그리고 오직 그때에만 간선 $u \rightarrow v$를 포함합니다. G의 ***추이적 환원***은 추이적 폐포가 G^T인 그래프 중에서 간선의 수가 가장 적은 그래프입니다. 같은 그래프에 추이적 환원이 여러 개 있을 수 있습니다.

(a) 주어진 방향 그래프의 추이적 폐포(transitive closure)를 계산하는 효율적인 알고리즘을 설명하십시오.

(b) 방향 그래프 G가 *유일한* 추이적 환원(transitive reduction)을 가질 필요충분조건은 G가 비순환인 것임을 증명하십시오.

(c) 주어진 방향 그래프의 추이적 환원(transitive reduction)을 계산하는 효율적인 알고리즘을 설명하십시오.

7. 임의의 연결 그래프를 탐색하는 가장 오래된 알고리즘 중 하나는 1895년에 가스통 타리(Gaston Tarry)가 미로를 푸는 체계적인 절차로 제안한 것입니다.[7] 타리 알고리즘의 입력은 무방향 그래프 G입니다. 그러나 설명의 편의를 위해, 각 무방향 간선 uv를 두 개의 방향 간선 $u \to v$와 $v \to u$로 형식적으로 나눕니다. (실제 구현에서는 이 분할이 자명합니다. 알고리즘은 주어진 G의 인접 리스트를 단지 G가 방향 그래프*인 것처럼* 사용할 뿐입니다.)

```
TARRY(G):
  unmark all vertices of G
  color all edges of G white
  s ← any vertex in G
  RECTARRY(s)
```

```
RECTARRY(v):
  mark v                                  ⟪v를 "방문"⟫
  if there is a white arc v→w
      if w is unmarked
          color w→v green
      color v→w red                       ⟪"통과" v→w⟫
      RECTARRY(w)
  else if there is a green arc v→w
      color v→w red                       ⟪"통과" v→w⟫
      RECTARRY(w)
```

Tarry의 알고리즘이 v를 표시할 때마다 정점 v를 "방문"한다고 비공식적으로 말하며, 간선 $v \to w$를 빨간색으로 칠하고 RECTARRY(w)를 재귀적으로 호출할 때 그 간선을 "순회"한다고 말합니다. 앞서 살펴본 그래프 순회 알고리즘과 달리, Tarry의 알고리즘은 같은 정점을 여러 번 표시할 수 있습니다.

(a) Tarry 알고리즘이 $O(V + E)$ 시간에 실행되도록 구현하는 방법을 설명하십시오.

(b) G의 어떤 방향 간선도 두 번 이상 지나지 않음을 증명하십시오.

(c) 알고리즘이 정점 v를 k번째로 방문할 때, v로 들어오는 간선 중 정확히 몇 개가 빨간색이고, v에서 나가는 간선 중 정확히 몇 개가 빨간색입니까? *[힌트: 시작 정점 s는 다른 정점들과 따로 고려하십시오.]*

(d) 시작 정점 s를 제외한 각 정점 v는 최대 $\deg(v)$번 방문되고, s는 최대 $\deg(s)+1$번 방문된다는 것을 증명하십시오. 이 주장으로부터 TARRY(G)가 종료한다는 사실이 곧바로 따라 나옵니다.

(e) TARRY(G)가 마지막으로 방문하는 정점이 시작 정점 s임을 증명하십시오.

(f) TARRY(G)가 방문하는 모든 정점 v에 대해, TARRY(G)가 종료될 때 v로 들어오는 간선과 v에서 나가는 간선이 모두 빨간색임을 증명하십시오. *[힌트: s부터 시작하여 정점들을 처음 표시되는 순서대로 고려하고, 귀납법으로 이 주장을 증명하십시오.]*

[7] 이보다 더 오래된 그래프 순회 알고리즘으로는 1882년 샤를 트레모(Charles Trémaux), 1873년 크리스티안 비너(Christian Wiener), 1873년 카를 히어홀처(Carl Hierholzer)가 기술한 것이 있으며, 1736년 레온하르트 오일러(Leonhard Euler)도 (암묵적으로) 기술한 바 있습니다. 특히 비너의 알고리즘은 연결된 무방향 그래프에서의 깊이 우선 탐색과 동등합니다.

(g) TARRY(G)가 G의 모든 정점을 방문함을 증명하십시오. 이 명제와 앞의 명제로부터 TARRY(G)가 G의 모든 간선을 정확히 한 번씩 지나간다는 사실이 따라 나옵니다.

8. Tarry의 그래프 순회 알고리즘을 다음과 같이 변형한 알고리즘을 생각해 봅시다. 이 변형은 초록색 간선을 빨간색으로 다시 칠하지 않고 순회하며, 모든 정점에 두 개의 숫자 레이블을 부여합니다.

```
TARRY2(G):
  unmark all vertices of G
  color all edges of G white
  s ← any vertex in G
  RECTARRY2(s, 1)
```

```
RECTARRY2(v, clock):
  if v is unmarked
      v.pre ← clock; clock ← clock + 1
      mark v
  if there is a white arc v→w
      if w is unmarked
          color w→v green
      color v→w red
      RECTARRY2(w, clock)
  else if there is a green arc v→w
      v.post ← clock; clock ← clock + 1
      RECTARRY2(w, clock)
```

다음 주장을 증명하거나 반증하십시오: TARRY2(G)가 종료할 때, 초록색 간선들은 신장 트리를 정의하고 레이블 $v.pre$와 $v.post$는 전위 순서 레이블링과 후위 순서 레이블링을 정의하며, 이들은 모두 G에 대한 하나의 깊이 우선 탐색과 일치합니다. 다시 말해, TARRY2가 간선들을 완전히 다른 순서로 방문함에도 불구하고 깊이 우선 탐색과 같은 *출력*을 생성한다는 것을 증명하거나 반증하십시오.

9. 여러분에게는 n개의 자물쇠 상자와 m개의 금 열쇠가 있습니다. 각 열쇠는 *많아야* 한 개의 상자를 엽니다. 하지만 각 상자는 열쇠 하나로 열릴 수도 있고, 여러 열쇠로 열릴 수도 있으며, 어떤 열쇠로도 열리지 않을 수도 있습니다. 일단 잠긴 상자를 여는 방법은 두 가지뿐입니다. 제대로 여는 방법(이를 위해서는 맞는 열쇠 하나를 손에 쥐고 있어야 합니다)과 망치로 산산조각 내는 방법입니다.

   반짝이는 물건을 가지고 노는 것을 좋아하는 어린 남동생이 어찌 된 일인지 여러분의 열쇠를 모두 상자 안에 넣고 잠가 버렸습니다! 다행히 집의 보안 시스템이 모든 과정을 녹화해 두었으므로, 각 상자 안에 어떤 열쇠가 들어 있는지(또는 아무 열쇠도 없는지) 정확히 알고 있습니다. 열쇠는 금으로 만들어져 있으므로 모든 열쇠를 상자에서 다시 꺼내야 합니다. 적어도 상자 하나는 부숴야 한다는 것은 분명합니다.

   (a) 어린 동생이 망치를 찾아내서는 상자 하나를 잔뜩 기대에 찬 눈으로 바라보고 있습니다. 동생이 고른 상자를 제외한 어떤 상자도 부수지 않고 모든 열쇠를 꺼낼 수 있는지 판별하는 알고리즘을 설명하고 분석하십시오.

   (b) 모든 열쇠를 되찾기 위해 부숴야 하는 상자의 최소 개수를 계산하는 알고리즘을 설명하고 분석하십시오.

10. 여러분이 알고리즘 강의를 맡고 있다고 가정해 봅시다. 두 번째 중간고사에서 여러분은 학생들에게 그래프 그림을 하나 나누어 주고, 특정 정점을 루트로 하는 너비 우선 탐색 트리와 깊이 우선 탐색 트리를 표시하라고 요구합니다. 안타깝게도 채점을 시작하고 나서야 학생들에게 준 그래프에 그러한 신장 트리가 여러 개 있다는 사실을 깨닫습니다—일일이 나열하기에는 너무나 많습니다. 그러므로 그 대신 각 학생의 답안이 올바른지 판별할 방법이 필요합니다!

    다음 각 문제에서 연결 그래프 G, 시작 정점 s, 그리고 G의 신장 트리 T가 주어진다고 가정합시다.

    (a) G가 *무방향* 그래프라고 가정합시다. T가 s를 루트로 하는 *깊이* 우선 신장 트리인지 판별하는 알고리즘을 서술하고 분석하십시오.

    (b) G가 *무방향* 그래프라고 가정합시다. T가 s를 루트로 하는 *너비* 우선 신장 트리인지 판별하는 알고리즘을 기술하고 분석하십시오. *[힌트: T가 가중치 없는 최단 경로 트리라는 것만으로는 충분하지 않습니다. 네, 이 문제가 실린 장은 이 장이 맞습니다!]*

    (c) G가 *방향* 그래프라고 가정합시다. T가 s를 루트로 하는 *너비* 우선 신장 트리인지 판정하는 알고리즘을 기술하고 분석하십시오. *[힌트: (b)번을 먼저 푸십시오.]*

    (d) G가 *방향* 그래프라고 가정합시다. T가 s를 루트로 하는 *깊이* 우선 신장 트리인지 판정하는 알고리즘을 설명하고 분석하십시오.

11. JavaScript, Python, Perl, Ruby를 비롯한 여러 현대 프로그래밍 언어에는 ***병렬 대입***이라는 기능이 있으며, 이 기능을 사용하면 여러 대입 연산을 코드 한 줄로 작성할 수 있습니다. 예를 들어 Python 코드 `x,y = 0,1`은 `x`를 0으로, `y`를 1로 동시에 설정합니다. 대입문 우변의 값은 모두 변수의 *이전* 값으로 결정됩니다. 따라서 Python 코드 `a,b = b,a`는 `a`와 `b`의 값을 서로 맞바꾸며, 다음 Python 코드는 `n`번째 피보나치 수를 계산합니다.

```
def fib(n):
    prev, curr = 1, 0
    while n > 0:
        prev, curr, n = curr, prev+curr, n-1
    return curr
```

    여러분이 작성하는 인터프리터가 모든 병렬 대입을 그와 동등한 개별 대입의 나열로 변환해야 한다고 가정합시다. 예를 들어, 병렬 대입 `a,b = 0,1`은 어느 순서로든 직렬화할 수 있습니다—`a=0; b=1`로도, `a=0; b=1`로도 가능합니다—그러나 병렬 대입 `x,y = x+1,x+y`는 오직 `y=x+y; x=x+1`로만 직렬화할 수 있습니다. 직렬화에는 하나 이상의 추가 임시 변수가 필요할 수 있습니다. 예를 들어, `a,b = b,a`를 직렬화하려면 임시 변수가 하나 필요하고, `x,y = x+y,x-y`를 직렬화하려면 임시 변수가 두 개 필요합니다.

(a) 주어진 병렬 대입을 추가 임시 변수 없이 직렬화할 수 있는지 판정하는 알고리즘을 설명하십시오.

(b) 주어진 병렬 대입을 *정확히 하나의* 추가 임시 변수로 직렬화할 수 있는지 판정하는 알고리즘을 설명하십시오.

주어진 병렬 대입문에는 단순 정수 변수만 사용되고(포인터나 배열을 통한 간접 참조는 없음), 어떤 변수도 좌변에 두 번 이상 나타나지 않으며, 우변의 식에는 부수 효과가 없다고 가정하십시오. 대입문을 파싱하는 세부 사항은 신경 쓰지 말고, 적절한 그래프 표현을 가정하기만 하십시오(단, 그 표현을 설명하십시오!).

동적 계획법

12. 방향 비순환 그래프 G가 주어졌다고 가정합시다. 이 그래프의 노드는 작업을 나타내고 간선은 선행 제약 조건을 나타냅니다. 즉, 각 간선 $u \to v$는 작업 v가 시작되기 전에 작업 u가 완료되어야 함을 의미합니다. 또한 각 노드 v에는 작업 v를 실행하는 데 필요한 시간을 나타내는 가중치 $T(v)$가 있습니다.

(a) G의 모든 작업을 실행할 수 있는 가장 짧은 시간 구간을 구하는 알고리즘을 설명하십시오.

(b) 첫 번째 작업이 시각 0에 시작한다고 가정합시다. 각 정점 v에 대해 작업 v가 시작될 수 있는 가장 이른 시각을 구하는 알고리즘을 설명하십시오.

(c) 이제 모든 정점 v에 대해, 선행 제약 조건을 위반하거나 (a)에서 계산한 전체 완료 시간을 늘리지 않으면서 작업 v를 시작할 수 있는 *가장* 늦은 시각을 구하는 알고리즘을 설명하십시오. 이때 v를 제외한 모든 작업은 (b)에서 계산한 가장 이른 시작 시각에 시작한다고 가정합니다.

13. G를 유일한 소스 s와 유일한 싱크 t를 갖는 방향 비순환 그래프라고 합시다.

(a) G의 *해밀턴 경로*란 G의 모든 정점을 포함하는 G 안의 방향 경로입니다. G에 해밀턴 경로가 있는지 판별하는 알고리즘을 설명하십시오.

(b) G의 *정점*에 가중치가 있다고 가정합시다. s에서 t로 가는 경로 중 총 가중치가 최대인 경로를 찾는 효율적인 알고리즘을 설명하십시오.

(c) 정수 ℓ도 함께 주어진다고 가정합시다. s에서 t로 가는 경로 중 간선을 최대 ℓ개 포함하는 최대 가중치 경로를 찾는 효율적인 알고리즘을 설명하십시오. (그러한 경로가 적어도 하나 존재한다고 가정합니다.)

(d) G의 정점 중 일부가 *중요* 정점으로 표시되어 있고, 정수 k도 함께 주어진다고 가정합시다. 중요 정점을 적어도 k개 방문하는, s에서 t로 가는 최대 가중치 경로를 찾는 효율적인 알고리즘을 설명하십시오. (그러한 경로가 적어도 하나 존재한다고 가정합니다.)

(e) G에서 s부터 t까지의 경로 개수를 계산하는 알고리즘을 설명하십시오. (임의로 큰 정수의 덧셈을 $O(1)$ 시간에 수행할 수 있다고 가정하십시오.)

14. 각 정점에 어떤 고정된 알파벳의 레이블이 붙어 있는 방향 비순환 그래프를 G라 하고, 같은 알파벳 위의 문자열을 $A[1..\ell]$이라고 합시다. G의 모든 방향 경로에는 레이블이 있으며, 이 레이블은 경로에 있는 정점들의 레이블을 이어 붙여 얻은 문자열입니다.

    (a) G에서 레이블이 A인 경로를 찾거나, 그러한 경로가 없다고 올바르게 보고하는 알고리즘을 설명하십시오.

    (b) G에서 레이블이 A인 경로의 *개수*를 구하는 알고리즘을 설명하십시오. (임의로 큰 정수의 덧셈을 $O(1)$ 시간에 할 수 있다고 가정하십시오.)

    (c) 레이블이 A의 부분 수열인 G의 최장 경로를 찾는 알고리즘을 설명하십시오.

    (d) G에서 레이블이 A의 *초수열*(supersequence)인 *최단* 경로를 찾는 알고리즘을 설명하십시오.

    (e) G에서 레이블과 A 사이의 편집 거리가 최소인 경로를 찾는 알고리즘을 설명하십시오.

15. ***다각 경로***는 끝과 끝을 이어 붙인 선분들의 수열이며, 이 선분들의 끝점을 경로의 ***정점***이라고 합니다. 다각 경로의 ***길이***는 그 선분들의 길이의 합입니다. 정점이 $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$인 다각 경로는 모든 인덱스 i에 대해 $x_i < x_{i+1}$이고 $y_i < y_{i+1}$이면 ***단조 증가***한다고 합니다—비형식적으로 말하면, 경로의 각 정점이 바로 앞 정점보다 위쪽이면서 오른쪽에 있다는 뜻입니다.

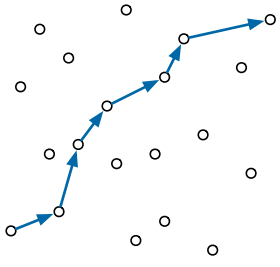


그림 6.22. 점 집합을 지나는, 정점이 일곱 개인 단조 증가 꺾은선 경로

    평면 위의 n개의 점으로 이루어진 집합 S가 두 배열 $X[1..n]$과 $Y[1..n]$으로 표현되어 주어진다고 가정합시다. 정점이 S에 속하는 단조 증가 경로 중 가장 긴 경로의 길이를 계산하는 알고리즘을 설명하고 분석하십시오. (x, y)에서 (x', y')까지의 선분 길이를 반환하는 서브루틴 $\textsc{Length}(x, y, x', y')$가 있다고 가정하십시오.

16. 방향 비순환 그래프 G의 임의의 두 노드 u와 w에 대해, ***구간*** $G[u, w]$는 G에서 u로부터 v로 가는 모든 방향 경로의 합집합입니다. 동치로, $G[u, w]$는 $v \in reach(u)$이고 $w \in reach(x)$인 모든 정점 v와, 그 정점들을 잇는 G의 모든 간선으로 이루어집니다.

    모든 정점에 양수, 음수 또는 0일 수 있는 수치 가중치가 부여된 방향 비순환 그래프 G가 주어졌다고 가정합시다.

(a) 각 구간의 가중치가 해당 구간에 속한 정점들의 가중치의 합일 때, G에서 가중치가 최대인 구간을 찾는 효율적인 알고리즘을 설명하십시오.

(b) G의 모든 구간에서 가장 큰 정점 가중치를 찾는 효율적인 알고리즘을 설명하십시오. 여러분의 알고리즘은 2차원 배열 $MaxWt[1..V, 1..V]$를 계산해야 하며, 여기서 각 항목 $MaxWt[u, w]$는 구간 $G[u, w]$에 속한 모든 정점의 가중치 중 최댓값입니다. 특히 $G[u, w]$가 비어 있으면 $MaxWt[u, w]$는 $-\infty$여야 합니다.

17. G를 정점들이 어떤 고정된 알파벳에서 온 레이블을 가지는 방향 비순환 그래프라고 합시다. G의 모든 방향 경로에는 레이블이 있으며, 이 레이블은 경로에 속한 정점들의 레이블을 이어 붙여 얻은 문자열입니다. *회문*이란 뒤집어도 원래와 같은 문자열이라는 점을 떠올리십시오.

(a) G에서 어떤 경로의 레이블이 되는 가장 긴 회문의 길이를 구하는 알고리즘을 설명하고 분석하십시오. 예를 들어, 그림 6.23의 그래프가 주어지면 여러분의 알고리즘은 회문 HANNAH의 길이인 정수 6을 반환해야 합니다.

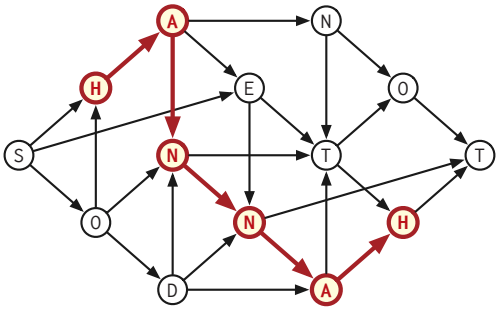


그림 6.23. 가장 긴 회문 경로 레이블의 길이가 6인 dag.

(b) G에 있는 어떤 경로의 레이블의 부분 수열이 되는 가장 긴 회문을 찾는 알고리즘을 설명하십시오.

(c) G에 소스 s와 싱크 t가 각각 하나씩만 있다고 가정합시다. G에서 s부터 t까지 가는 경로의 레이블의 초수열인 가장 짧은 회문을 찾는 알고리즘을 설명하십시오.

18. 모든 노드가 어떤 유한 알파벳에 속하는 *레이블*을 가지는 두 방향 비순환 그래프 G와 H가 주어졌다고 가정합시다. 서로 다른 노드가 같은 레이블을 가질 수도 있습니다. 두 dag 중 어느 쪽에서든 *경로*의 레이블은 그 경로를 이루는 정점들의 레이블을 이어 붙여 얻은 문자열입니다.

(a) G의 어떤 경로의 레이블이면서 동시에 H의 어떤 경로의 레이블이기도 한 가장 긴 문자열의 길이를 계산하는 알고리즘을 설계하고 분석하십시오.

(b) G의 어떤 경로의 레이블의 부분 수열이면서 H의 어떤 경로의 레이블의 부분 수열이기도 한 가장 긴 문자열의 길이를 계산하는 알고리즘을 설명하고 분석하십시오.

(c) G에 있는 어떤 경로의 레이블의 초수열이면서 H에 있는 어떤 경로의 레이블의 초수열이기도 한 가장 짧은 문자열의 길이를 계산하는 알고리즘을 설명하고 분석하십시오. *[힌트: 이 문제는 보기보다 쉽습니다.]*

19. G를 모든 정점 v가 정수 가중치 $w(v)$를 갖는 임의의 (반드시 비순환일 *필요는 없는*) 방향 그래프라고 합시다.

(a) 정점 가중치가 증가 수열을 이루는 G의 가장 긴 방향 경로를 찾는 알고리즘을 설명하십시오.

(b) G의 각 정점에서 도달할 수 있는 최대 가중치 정점을 구하는 알고리즘을 설명하고 분석하십시오. 즉, 알고리즘은 각 정점 v에 대해 $maxreach(v) := \max\{w(x) \mid x \in reach(v)\}$를 계산해야 합니다.

20. (a) n개의 정점을 가진 방향 비순환 그래프 G와 정수 $k \leq n$이 주어졌다고 가정합시다. G의 모든 정점을 방문하는 최대 k개의 정점 서로소 경로(vertex-disjoint 서로소 경로)의 집합을 찾는 효율적인 알고리즘을 설명하십시오.

(b) 이제 입력 dag G의 간선에 가중치가 있으며, 가중치는 양수, 음수 또는 0일 수 있다고 가정합시다. G의 모든 정점을 방문하는, 최대 k개의 정점 서로소 경로로 이루어진 집합 중 *총 가중치가 최소인* 집합을 찾는 효율적인 알고리즘을 설명하십시오.

여러분의 알고리즘은 어떤 작은 상수 c에 대해 $O(n^{k+c})$ 시간에 실행되어야 합니다. 정점 하나만으로도 가중치가 0인 경로입니다. ((a)에 대한 더 효율적인 알고리즘은 11장에서 살펴보겠습니다.)

21. Kris는 미국 클라이밍 전국 대회에 출전한 프로 암벽 등반가입니다. 이 대회에서 Kris는 루트 세터가 명시적으로 허용한 이동만 사용하여 클라이밍 벽의 홀드를 가능한 한 많이 사용해야 합니다.

클라이밍 벽에는 n개의 홀드가 있습니다. 크리스는 홀드 쌍 (x, y) m개로 이루어진 목록을 받는데, 각 쌍은 홀드 x에서 홀드 y로 곧바로 이동할 수 있음을 나타냅니다. 그러나 목록에 쌍 (y, x)도 포함되어 있지 않다면 y에서 x로 곧바로 이동할 수는 없습니다. 새로운 홀드를 하나 사용할 때마다 점수가 1점씩 올라가므로, 크리스는 가능한 한 많은 홀드를 사용하는 허용된 이동의 순서를 찾아내야 합니다. 규칙상 크리스는 등반 경로의 첫 홀드와 마지막 홀드를 직접 고를 수 있습니다. 또한 규칙상 각 홀드를 원하는 만큼 여러 번 사용할 수 있지만, 크리스의 점수는 각 홀드를 처음 사용할 때만 올라갑니다.

(a) 입력을 나타내는 자연스러운 그래프를 정의하십시오. 입력 그래프가 dag임이 보장될 때 크리스의 등반 문제를 해결하는 알고리즘을 기술하고 분석하십시오.

(b) 입력 그래프에 아무런 제약이 없을 때 크리스의 등반 문제를 해결하는 알고리즘을 서술하고 분석하십시오.

두 알고리즘 모두 크리스가 얻을 수 있는 최대 점수를 출력해야 합니다.

22. m개의 은하 간 순간이동 통로로 연결된 n개의 은하가 있습니다. 각 순간이동 통로는 두 은하를 잇고 양방향으로 지나갈 수 있습니다. 그러나 순간이동 통로를 운영하는 회사는 엄청난 수익을 내는 요금 체계를 만들어 두었습니다. 누구나 자신의 고향 은하에서 *더 멀어지는* 방향으로는 전혀 비용 없이 순간이동할 수 있지만, 고향 은하 *쪽으로* 순간이동하는 데에는 감당할 수 없을 만큼 비싼 요금이 듭니다.

주디는 안식년을 맞아 고향 은하에서 출발하여 가능한 한 많은 은하를 방문하는 우주 여행을 떠나기로 했습니다. 여행 경비를 아끼기 위해, 주디는 고향으로 돌아가는 맨 마지막 순간이동을 제외하고는 매 단계마다 고향 은하에서 더 멀어지는 쪽으로 순간이동하려고 합니다.

(a) 주디가 방문할 수 있는 은하의 최대 개수를 계산하는 알고리즘을 설명하고 분석하십시오. 입력은 순간이동 통로 네트워크를 나타내며 n개의 정점과 m개의 간선으로 이루어진 무방향 그래프 G, 주디의 고향 은하를 나타내는 정수 $1 \le s \le n$, 그리고 각 은하와 s 사이의 거리를 담은 배열 $D[1..n]$으로 구성됩니다.

♥(b) 우주 일주 여행을 떠나기 직전에 주디는 우주 복권에 당첨되어, 고향 은하 쪽으로 *두 번* 순간이동할 수 있을 만큼의 돈을 얻게 됩니다. 주디가 방문할 수 있는 서로 다른 은하의 최대 개수를 계산하는 새로운 알고리즘을 설명하십시오. 주디는 같은 은하를 여러 번 방문할 수 있지만, 중요한 점은 첫 번째 방문만 총합에 포함된다는 것입니다.

23. 닥터와 리버 송은 소스 s 하나와 싱크 t 하나를 가진 방향 비순환 그래프 G 위에서 게임을 하기로 합니다.[8]

각 플레이어는 G의 정점 중 하나에 토큰을 하나씩 가지고 있습니다. 게임을 시작할 때 닥터의 토큰은 소스 정점 s에 있고, 리버의 토큰은 싱크 정점 t에 있습니다. 두 플레이어는 번갈아 가며 차례를 진행하며, 닥터가 먼저 움직입니다. 닥터는 자신의 차례마다 방향 간선을 따라 토큰을 앞으로 옮기고, 리버는 자신의 차례마다 방향 간선을 따라 토큰을 *뒤로* 옮깁니다.

두 토큰이 언제든 같은 정점에서 만나면 리버가 게임에서 이깁니다. ("안녕, 자기!") 두 토큰이 만나기 전에 닥터의 토큰이 t에 도달하거나 리버의 토큰이 s에 도달하면 닥터가 게임에서 이깁니다.

두 플레이어가 모두 완벽하게 플레이한다고 가정할 때, 이 게임에서 누가 이기는지 판정하는 알고리즘을 설명하고 분석하십시오. 즉, *리버가 어떻게 움직이든* 닥터가

[8] 레이블 s와 t는 언템퍼드 스키즘(Untempered **S**chism)과 시간 소용돌이(**T**ime Vortex)의 약자이거나, 일곱 성계의 빛나는 세계(Shining World of the Seven Systems, 갈리프레이라고도 합니다)와 트렌잘로어(Trenzalore)의 약자이거나, 스카로(Skaro)와 텔로스(Telos)의 약자이거나, 아니면 또 다른 타이미-와이미한 무언가(Something else Timey-wimey)의 약자입니다. 전부 아주 복잡한 이야기이니 신경 쓰지 마십시오.

이길 수 있다면 알고리즘은 "Doctor"를 출력해야 하고, *닥터가 어떻게 움직이든* 리버가 이길 수 있다면 알고리즘은 "River"를 출력해야 합니다. (왜 이 두 가지 경우만 가능합니까?) 알고리즘의 입력은 그래프 G입니다.

♣♥24. 어떤 유한 알파벳 Σ 위에서 주어진 n글자 문자열을 $x = x_1 x_2 \ldots x_n$이라 하고, 같은 알파벳 위에서 정의된 상태가 m개인 결정적 유한 상태 기계를 A라고 합시다.

(a) A가 받아들이는 x의 가장 긴 부분 수열의 길이를 계산하는 알고리즘을 설명하고 분석하십시오. 예를 들어 A가 언어 $(\texttt{AR})^*$를 받아들이고 $x = \texttt{ABRACADABRA}$이면, 알고리즘은 부분 수열 ARAR의 길이인 숫자 4를 출력해야 합니다.

(b) A가 받아들이는 x의 초수열(supersequence) 중 가장 짧은 것의 길이를 계산하는 알고리즘을 설명하고 분석하십시오. 예를 들어 A가 언어 $(\texttt{ABCDR})^*$를 받아들이고 $x = \texttt{ABRACADABRA}$이면, 알고리즘은 숫자 25를 출력해야 합니다. 이는 초수열 ABCDRABCDRABCDRABCDRABCDR의 길이입니다.

입력 문자열의 길이 n, 유한 상태 기계의 상태 수 m, 알파벳 Σ의 크기를 기준으로 알고리즘을 분석하십시오.

25. *모든 동적 계획법 알고리즘을 방향 비순환 그래프에서 최적 경로를 찾는 문제로 모델링할 수 있는 것은 아니지만, 모든 동적 계획법 알고리즘은 실제로 그 바탕에 있는 어떤 의존성 그래프를 후위 순서로 처리합니다.*

(a) 모든 노드에 수치 탐색 키가 저장된 방향 비순환 그래프 G가 주어졌다고 가정합시다. G의 부분 그래프인 가장 큰 이진 탐색 트리를 찾는 알고리즘을 설명하고 분석하십시오.

(b) 방향 비순환 그래프 G와 두 정점 s, t가 주어졌다고 가정합시다. G에서 s로부터 t로 가는 방향 경로의 개수를 계산하는 알고리즘을 설명하십시오. (모든 산술 연산에는 $O(1)$ 시간이 걸린다고 가정하십시오.)

(c) G가 다음과 같은 특징을 갖는 방향 비순환 그래프라고 합시다.

- G에는 소스 s가 하나 있고, 싱크 $t_1, t_2, \ldots, t_k$가 여러 개 있습니다.
- 각 간선 $v \to w$에는 0과 1 사이의 가중치 $p(v \to w)$가 연관되어 있습니다.
- 싱크가 아닌 각 정점 v에 대해, v에서 나가는 모든 간선의 가중치 합은 1입니다. 즉, $\sum_w p(v \to w) = 1$입니다.

가중치 $p(v \to w)$는 G에서 소스 s로부터 어떤 싱크 t_i로 향하는 랜덤 워크를 정의합니다. 이 워크는 싱크가 아닌 임의의 정점 v에 도달하면 확률 $p(v \to w)$로 간선 $v \to w$를 따라갑니다. 모든 확률은 서로 독립입니다. 모든 인덱스 i에 대해 이 랜덤 워크가 싱크 t_i에 도달할 확률을 계산하는 알고리즘을 설명하고 분석하십시오. (각 산술 연산은 $O(1)$ 시간만 걸린다고 가정하십시오.)

여러분, 우리는 모두 함께 매달려야 합니다.
그렇지 않으면 틀림없이 하나씩 따로 목이 매달리게 될 것입니다.

— 벤저민 프랭클린, 독립 선언서
서명식에서 (1776년 7월 4일)

저는 누군가에게—누구였는지는 기억나지 않습니다만—
이 연구를 출판하려고 투고할 가치가 있는지 조언을 구했던 기억이 납니다. 거기에 쓰인 논증이 너무나
단순했기 때문입니다....다행히 그는 투고해 보라고 권해 주었고, 여러 해가 지나서야 . 비로소 제 다른 논문이 이 아주 단순한 논문만큼 널리 알려지게 되었습니다.

— 조지프 크러스컬, 자신의 최단 신장 부분 트리 알고리즘을 설명하며 (1997)

전부 다 청소하십시오!

— 앨리 브로시, "내가 절대 어른이 될 수 없는 이유",
Hyperbole and a Half, 2010년 6월 17일.

7

최소 신장 트리

연결되어 있고 방향이 없는 *가중* 그래프가 주어졌다고 가정합시다. 이는 그래프 $G = (V, E)$와 함께, 각 간선 e에 실수 *가중치* $w(e)$를 부여하는 함수 $w : E \to \mathbb{R}$가 주어진 것이며, 이 가중치는 양수, 음수, 0 중 어느 것이든 될 수 있습니다. 이 장에서는 G의 ***최소 신장 트리***, 즉 다음 함수를 최소화하는 신장 트리 T를 찾는 여러 알고리즘을 설명합니다

$$w(T) := \sum_{e \in T} w(e).$$

예시는 그림 7.1을 참조하십시오.

7.1 서로 다른 간선 가중치

문제 서술에서 성가신 미묘한 점은 가중 그래프에 최소 가중치가 같은 신장 트리가 둘 이상 있을 수 있다는 것입니다. 특히 G의 모든 간선의 가중치가 1이면, G의 모든 신장 트리가 최소

신장 트리이며, 가중치는 $V - 1$입니다. 이러한 모호성 때문에 알고리즘을 개발하기가 복잡해집니다. 최소 신장 트리가 유일하다고 그냥 *가정*할 수만 있다면 모든 것이 훨씬 간단해질 것입니다.

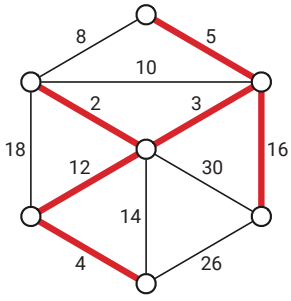


그림 7.1. 가중 그래프와 그 최소 신장 트리입니다.

다행히도 우리가 원하는 유일성을 보장하는 간단한 조건이 있습니다.

보조정리 7.1. *연결 그래프 G의 모든 간선 가중치가 서로 다르면, G는 유일한 최소 신장 트리를 가집니다.*[1]

증명: G를 두 최소 신장 트리 T와 T'를 갖는 임의의 연결 그래프라고 합시다. 이때 G의 어떤 두 간선이 같은 가중치를 갖는다는 것을 증명해야 합니다. 이 증명은 본질적으로 탐욕 교환 논증입니다.

두 신장 트리는 각각 상대 트리에 없는 간선을 하나 이상 포함해야 합니다. $T \setminus T'$에서 가중치가 최소인 간선을 e라 하고, $T' \setminus T$에서 가중치가 최소인 간선을 e'라 합시다(가중치가 같은 경우는 임의로 하나를 고릅니다). 일반성을 잃지 않고 $w(e) \le w(e')$라고 가정합시다.

부분 그래프 $T' \cup \{e\}$에는 사이클 C가 정확히 하나 있으며, 이 사이클은 간선 e를 지납니다. 이 사이클의 간선 중 T에 *속하지 않는 임의의* 간선을 e''이라고 합시다. T는 트리이므로 그러한 간선이 적어도 하나는 반드시 존재합니다. ($e'' = e'$일 수도 있고 아닐 수도 있습니다.) $e \in T$이므로 곧바로 $e'' \neq e$이고, 따라서 $e'' \in T' \setminus T$입니다. 그러므로 $w(e'') \ge w(e') \ge w(e)$가 성립합니다.

이제 신장 트리 $T'' = T' + e - e''$를 생각해 봅시다. (이 새로운 트리 T''는 T와 같을 수도 있습니다.) 곧바로 $w(T'') = w(T') + w(e) - w(e'') \le w(T')$를 얻습니다. 그런데 T'는 *최소* 신장 트리이므로 반드시 $w(T'') = w(T')$이어야 합니다. 다시 말해 T'' 역시 최소 신장 트리입니다. 따라서 $w(e) = w(e'')$라는 결론을 얻으며, 이로써 증명이 끝납니다. □

서로 다른 간선 가중치를 가정하는 알고리즘이 이미 있다면, 동점을 일관되게 처리하는 방법만 있으면 일부 간선의 가중치가 같은 그래프에서도 그 알고리즘을 그대로 실행할 수 있습니다. 그러한 방법 중 하나는 다음을 사용합니다

알고리즘에서 단순한 가중치 비교 대신 사용합니다. ShorterEdge는 서로 다를 필요가 없는 네 정점을 나타내는 네 정수 i, j, k, l을 입력으로 받아, 두 간선 (i, j)와 (k, l) 중 어느 쪽의 가중치가 "더 작은지" 판정합니다. (입력 그래프가 무방향이므로 쌍 (i, j)와 (j, i)는 같은 간선을 나타냅니다.)

[1] 이 보조정리의 역은 거짓입니다. 간선 가중치가 중복되는 연결 그래프도 유일한 최소 신장 트리를 가질 수 있습니다. 자명한 예로, G가 트리라고 가정해 봅시다!

```
ShorterEdge(i, j, k, l)
  if w(i, j) < w(k, l) then return (i, j)
  if w(i, j) > w(k, l) then return (k, l)
  if min(i, j) < min(k, l) then return (i, j)
  if min(i, j) > min(k, l) then return (k, l)
  if max(i, j) < max(k, l) then return (i, j)
《max(i,j) > max(k,l)이면》 return (k, l)
```

보조정리 7.1과 이 동점 처리 규칙에 비추어, 이 장의 나머지 부분에서는 간선 가중치가 *항상* 서로 다르며 따라서 최소 신장 트리가 *항상* 유일하다고 안심하고 가정하겠습니다. 특히 아무런 혼동 없이 ***유일한*** 최소 신장 트리에 대해 자유롭게 논의할 수 있습니다.

7.2 유일한 최소 신장 트리 알고리즘

최소 신장 트리를 계산하는 알고리즘은 많지만, 그 거의 전부가 다음과 같은 일반적인 전략의 사례입니다. 이는 그래프 순회와 비슷한 상황입니다. 그래프 순회에서도 서로 다른 여러 알고리즘이 모두 일반적인 순회 알고리즘인 아무거나 우선 탐색(whatever-first search)의 변형입니다.

범용 최소 신장 트리 알고리즘은 입력 그래프 G의 비순환 부분 그래프 F를 유지하며, 이를 *중간 신장 숲*이라고 부르겠습니다. F는 항상 다음 불변식을 만족합니다.

> F는 G의 최소 신장 트리의 부분 그래프입니다.

처음에 F는 정점 하나로 이루어진 트리 V개로 구성됩니다. 일반적인 알고리즘은 F에 있는 트리들 사이에 특정 간선을 추가하여 트리들을 연결합니다. 알고리즘이 멈추면 F는 하나의 신장 트리로 이루어지며, 우리의 불변식에 따르면 이 트리는 반드시 G의 최소 신장 트리여야 합니다. 모든 간선이 최소 신장 트리에 속하는 것은 아니므로, 자라나는 숲에 *어떤* 간선을 추가할지는 당연히 신중하게 정해야 합니다.

중간 신장 숲 F는 그것이 자라나는 과정의 어느 단계에서든 그래프의 나머지 부분에 두 가지 특별한 유형의 간선을 유도합니다.

- 어떤 간선이 F의 간선은 아니지만 두 끝점이 모두 F의 같은 컴포넌트에 속하면, 그 간선을 ***쓸모없는*** 간선이라고 합니다.
- 어떤 간선이 F의 어떤 컴포넌트에 정확히 한쪽 끝점만 두는 간선 중 가중치가 최소인 간선이면, 그 간선은 ***안전***합니다.

같은 간선이 F의 서로 다른 두 컴포넌트 모두에 대해 안전할 수도 있습니다. $G \setminus F$의 간선 중에는 안전하지도 쓸모없지도 않은 것이 있는데, 이러한 간선을 *미결정* 간선이라고 부릅니다.

모든 최소 신장 트리 알고리즘은 두 가지 간단한 관찰에 기반합니다. 첫 번째 관찰은 1957년에 Robert Prim이 증명했으며(다만 그보다 앞선 여러 알고리즘에도 이미 암묵적으로 담겨 있습니다), 두 번째 관찰은 자명합니다.

보조정리 7.2 (Prim). *그래프 G의 최소 신장 트리는 모든 안전한 간선을 포함합니다.*

증명: 사실 우리는 다음과 같은 더 강한 명제를 증명합니다. G의 정점들의 *임의의* 부분집합 S에 대하여, G의 최소 신장 트리는 S 안에 정확히 하나의 끝점을 갖는 간선 중 가중치가 최소인 간선을 포함합니다. 앞의 보조정리와 마찬가지로, 이 주장은 탐욕적 교환 논증을 사용하여 증명합니다.

G의 정점들로 이루어진 임의의 부분집합을 S라 하고, S에 끝점이 정확히 하나만 있는 간선 중 가장 가벼운 간선을 e라 합시다. (모든 간선 가중치가 서로 다르다고 가정했으므로 e는 유일합니다.) e를 포함하지 *않는* 임의의 신장 트리를 T라 합시다. 이때 T가 G의 최소 신장 트리가 *아님*을 증명해야 합니다. T는 연결되어 있으므로 e의 한 끝점에서 다른 끝점으로 가는 경로를 포함합니다. 이 경로는 S의 정점에서 시작하여 S에 속하지 않는 정점에서 끝나므로, S에 끝점이 정확히 하나만 있는 간선을 적어도 하나 포함해야 합니다. 그러한 간선 중 *아무것이나* 하나를 e'라 합시다. T에는 사이클이 없으므로 T에서 e'를 제거하면 연결 요소가 정확히 두 개인 신장 숲이 만들어지며, 각 연결 요소는 e의 끝점을 하나씩 포함합니다. 따라서 이 숲에 e를 추가하면 새로운 신장 트리 $T' = T - e' + e$를 얻습니다. e의 정의에 따라 $w(e') > w(e)$이며, 이는 T'의 총 가중치가 T의 총 가중치보다 작다는 것을 의미합니다. 따라서 T는 G의 최소 신장 트리가 아니며, 이로써 증명이 완료됩니다. □

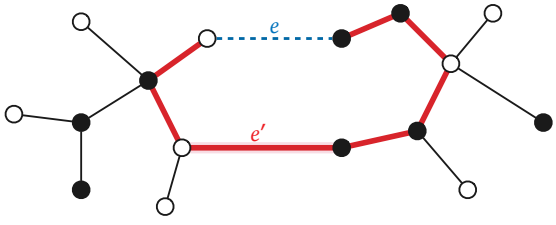


그림 7.2. 모든 안전한 간선은 최소 신장 트리에 포함됩니다. 검은색 정점은 부분집합 S에 속합니다.

보조정리 7.3. *최소 신장 트리는 쓸모없는 간선을 포함하지 않습니다.*

증명: 쓸모없는 간선을 하나라도 F에 추가하면 사이클이 생깁니다. □

우리의 일반적인 최소 신장 트리 알고리즘은 점점 자라나는 숲 F에 *안전한* 간선을 반복해서 추가합니다. 입력 그래프 G가 연결되어 있으므로, F가 아직 연결되어 있지 않다면 안전한 간선이 적어도 하나는 반드시 존재합니다. 따라서 각 반복에서 어떤 안전한 간선을 추가하든, 우리의 일반적인 알고리즘은 결국 F를 연결합니다. 귀납법에 의해 보조정리 7.2로부터 결과로 얻은 트리가 실제로 최소

신장 트리. F에 새 간선을 추가할 때마다, 결정되지 않은 간선 중 일부는 안전해질 수 있고, 결정되지 않은 다른 간선들은 쓸모없어질 수 있습니다. (한번 쓸모없어진 간선은 영원히 쓸모없는 상태로 남습니다.) 특정 알고리즘을 완전히 규정하려면, 각 반복에서 *어떤* 안전한 간선(들)을 추가할지, 그리고 그 간선들을 어떻게 찾을지를 기술해야 합니다.

7.3 보루프카 알고리즘

가장 오래되었으며 가장 단순하다고도 할 수 있는 최소 신장 트리 알고리즘은 1926년 체코의 수학자 오타카르 보루프카(Otakar Borůvka)가 발견했습니다. 인드르지흐 삭셀(Jindřich Saxel)이 가장 적은 양의 전선으로 여러 도시를 잇는 전력망을 구성하는 방법을 그에게 물은 지 약 1년 뒤의 일이었습니다.[2] 이 알고리즘은 1938년 귀스타브 쇼케(Gustav Choquet)가 재발견했습니다. 1951년에는 유제프 우카셰비치(Józef Łukaszewicz)가 이끄는 폴란드 수학자 팀이 다시 재발견했고, 1961년에는 조지 솔린(George Sollin)이 또다시 재발견했습니다. 솔린은 자신의 재발견을 발표한 적이 없지만, 그래프 알고리즘을 다룬 최초의 교과서 가운데 하나가 이를 자세히 설명하고 솔린의 공로로 밝혔습니다. 그 결과 이 알고리즘은 때때로 "솔린 알고리즘"이라고 불립니다.

Borůvka / Choquet / Florek-Łukaziewicz-Perkal-Steinhaus-Zubrzycki / Prim / Sollin / Brosh[3] 알고리즘은 한 줄로 요약할 수 있습니다:

> Borůvka: 안전한 간선을 **모두** 추가하고 재귀 호출합니다.

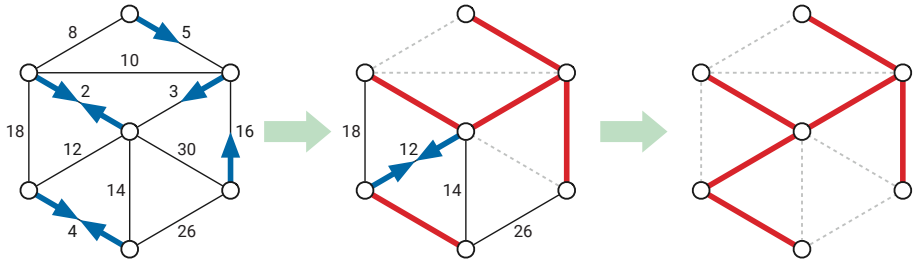


그림 7.3. 예제 그래프에서 실행한 보루프카 알고리즘입니다. 굵은 빨간색 간선은 F에 속하고, 점선 간선은 쓸모없는 간선입니다. 화살표는 각 성분의 안전한 간선을 따라 가리킵니다. 알고리즘은 단 두 번의 반복만에 종료됩니다.

다음은 보루프카 알고리즘을 좀 더 자세히 나타낸 것입니다. 이 알고리즘은 5장 (204쪽)의 CountAndLabel 알고리즘을 호출하여 F의 연결 요소 개수를 세고, 각 정점 v에 그 정점이 속한 연결 요소를 나타내는 정수 $comp(v)$로 레이블을 붙입니다.

```
BorůvkA(V, E):
  F = (V, ∅)
  count ← CountAndLabel(F)
  while count > 1
      AddAllSafeEdges(E, F, count)
      count ← CountAndLabel(F)
  return F
```

[2] 삭셀은 서모라바 전력 회사(West Moravian Power Company)의 직원이었습니다. 보루프카는 그를 "매우 재능 있고 근면한" 사람이라고 묘사했습니다. 삭셀은 훗날 유대계라는 이유로 나치에게 처형되었습니다.

[3] *Hyperbole and a Half*에 실린 글을 전부 읽어 보십시오. 그런 다음 그 책을 사십시오. 고양이에게 줄 책도 한 권 더 사십시오. 뭐라고 하셨습니까? 고양이가 없다는 말씀입니까? 대체 어떤 괴물이란 말입니까? 가서 고양이를 한 마리 데려온 다음, 그 고양이에게 *Hyperbole and a Half*를 한 권 더 사 주십시오.

이제 남은 일은 모든 안전한 간선을 찾아 F에 추가하는 방법을 설명하는 것뿐입니다. F의 성분이 둘 이상이라고 가정합니다. 그렇지 않다면 이미 끝난 것이기 때문입니다. 다음 서브루틴은 G의 모든 간선을 전수 조사하여 안전한 간선들의 배열 $safe[1..V]$를 계산합니다. 여기서 $safe[i]$는 한 끝점이 F의 i번째 성분에 속하는 간선 중 가중치가 최소인 간선입니다. 각 간선 uv에 대해, u와 v가 같은 성분에 속한다면 uv는 쓸모없는 간선이거나 이미 F에 속한 간선입니다. 그렇지 않다면 uv의 가중치를 $safe[comp(u)]$ 및 $safe[comp(v)]$의 가중치와 비교하고, 필요하면 배열의 항목을 갱신합니다. 모든 안전한 간선을 찾고 나면 각 간선 $safe[i]$를 F에 추가합니다.

```
AddAllSafeEdges(E, F, count):
  for i ← 1 to count
      safe[i] ← Null
  for each edge uv ∈ E
      if comp(u) ≠ comp(v)
          if safe[comp(u)] = Null or w(uv) < w(safe[comp(u)])
              safe[comp(u)] ← uv
          if safe[comp(v)] = Null or w(uv) < w(safe[comp(v)])
              safe[comp(v)] ← uv
  for i ← 1 to count
      add safe[i] to F
```

숲 F에는 간선이 최대 $V-1$개 있으므로 CountAndLabel을 한 번 호출할 때마다 $O(V)$ 시간이 걸립니다. 각 정점과 G의 각 간선, F의 각 성분마다 상수 시간을 쓰므로 *AddAllSafeEdges*는 $O(V+E)$ 시간에 실행됩니다. 입력 그래프가 연결되어 있으므로 $V \le E+1$이 성립합니다. 따라서 Borůvka의 while 루프는 반복마다 $O(E)$ 시간이 걸립니다.

반복할 때마다 F의 성분 수가 적어도 절반으로 줄어듭니다. 최악의 경우에는 F의 성분들이 두 개씩 짝지어 합쳐집니다. F는 처음에 V개의 성분을 가지므로, while 루프는 많아야 $O(\log V)$번 반복합니다. 따라서 Borůvka 알고리즘의 전체 실행 시간은 $O(E \log V)$라고 결론지을 수 있습니다.

여러분이 원하는 최소 신장 트리 알고리즘은 바로 이것입니다

기원이 비교적 덜 알려져 있었음에도 초기 서구의 알고리즘 연구자들은 보루프카(Borůvka) 알고리즘을 알고 있었지만, "너무 복잡하다"며 이를 무시했습니다.

그 결과, 단순하고 효율적임에도 불구하고 안타깝게도 대부분의 알고리즘 및 자료 구조 교과서는 보루프카 알고리즘을 언급조차 하지 않습니다. 이러한 누락은 심각한 실수입니다. 보루프카 알고리즘은 다른 고전적인 MST 알고리즘에 비해 몇 가지 뚜렷한 장점을 가지고 있습니다.

- Borůvka 알고리즘은 최악의 경우 실행 시간인 $O(E \log V)$보다 빠르게 동작하는 경우가 많습니다. 한 번의 반복에서 F의 성분 수가 2배보다 훨씬 큰 비율로 줄어들 수 있으므로, 반복 횟수가 최악의 경우인 $\lceil \log_2 V \rceil$보다 적어집니다.

- 보루프카 알고리즘을 약간 재정식화하면(실제로는 보루프카가 원래 제시한 방식에 더 가깝습니다), 간선이 교차하지 않도록 평면에 그릴 수 있는 그래프를 포함하여 폭넓은 부류의 흥미로운 그래프에 대해 실제로 $O(E)$ 시간에 실행됩니다. 이와 대조적으로, 나머지 두 알고리즘의 시간 분석은 모든 그래프에 적용됩니다.
- Borůvka 알고리즘은 상당한 수준의 병렬 처리가 가능합니다. 각 반복에서 F의 각 성분을 별도의 독립적인 스레드에서 처리할 수 있습니다. 이러한 암묵적 병렬성 덕분에 멀티코어 시스템이나 분산 시스템에서는 훨씬 더 빠른 성능을 얻을 수 있습니다. 반면에 나머지 두 고전적인 MST 알고리즘은 본질적으로 순차적입니다.
- 최근에 나온 몇몇 최소 신장 트리 알고리즘은 최악의 경우에도 여기서 설명하는 고전적인 알고리즘보다 빠릅니다. 이렇게 더 빠른 알고리즘은 모두 보루프카 알고리즘을 일반화한 것입니다. 요컨대 최소 신장 트리 알고리즘을 구현해야 할 일이 생긴다면 보루프카 알고리즘을 사용하십시오. 반면에 최소 신장 트리에 대해 효과적으로 *무언가를 증명하고* 싶다면 다음에 나올 두 알고리즘도 반드시 알아 두어야 합니다.

7.4 야르니크("Prim") 알고리즘

그다음으로 오래된 최소 신장 트리 알고리즘은 체코의 수학자 보이테흐 야르니크(Vojtěch Jarník)가 1929년 보루프카(Borůvka)에게 보낸 편지에서 처음 기술하였으며, 야르니크는 이듬해에 이 발견을 발표하였습니다. 이 알고리즘은 1956년 조지프 크러스컬(Joseph Kruskal)이, (논란의 여지는 있지만) 1957년 로버트 프림(Robert Prim)이, 같은 1957년 해리 로버먼(Harry Loberman)과 아널드 와인버거(Arnold Weinberger)가, 마지막으로 1958년 에츠허르 데이크스트라(Edsger Dijkstra)가 각각 독립적으로 재발견하였습니다. 프림, 로버먼과 와인버거, 데이크스트라는 모두 (결국에는) 크러스컬의 논문을 알고 있었고 인용까지 하였습니다. 그러나 크러스컬이 같은 논문에서 다른 최소 신장 트리 알고리즘 두 가지도 함께 기술하였기 때문에, *이* 알고리즘은 보통 "프림 알고리즘"이라고 불리며 때로는 "프림/데이크스트라 알고리즘"이라고도 불립니다. 1958년 무렵에는 데이크스트라의 이름이 이미 다른 알고리즘에 (부적절하게) 붙어 있었는데도 말입니다. 야르니크 알고리즘에서 중간 숲 F에는 자명하지 않은 성분 T가 하나뿐이며, 나머지 성분은 모두 고립된 정점입니다. 처음에 T는

그래프의 임의의 정점 하나로 이루어집니다. 이 알고리즘은 T가 그래프 전체를 신장할 때까지 다음 단계를 반복합니다:

> Jarník: T의 안전한 간선을 T에 반복해서 추가합니다.

야르니크 알고리즘을 구현하기 위해 T에 인접한 모든 간선을 우선순위 큐에 보관합니다. 우선순위 큐에서 가중치가 최소인 간선을 꺼내면, 먼저 그 간선의 두 끝점이 모두 T에 속하는지 확인합니다. 그렇지 않다면 그 간선을 T에 추가한 다음, 새로 인접하게 된 간선들을 우선순위 큐에 추가합니다. 다시 말해, 야르니크 알고리즘은 5장 끝에서 설명한 "최선 우선 탐색"의 한 변형입니다! 내부의 우선순위 큐를 표준 이진 힙으로 구현하면, 야르니크 알고리즘은 $O(E \log E) = O(E \log V)$ 시간에 실행됩니다.

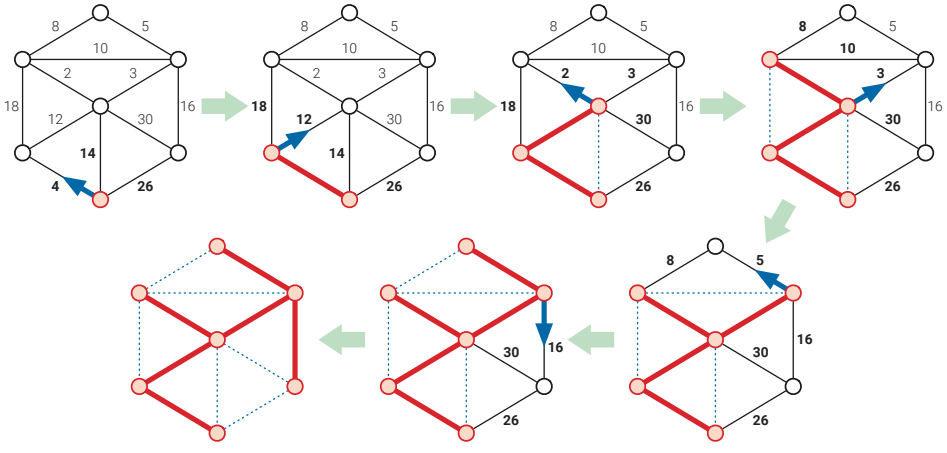


그림 7.4. 맨 아래 정점에서 시작하여 예제 그래프에 야르니크 알고리즘을 실행한 모습입니다. 각 단계에서 굵은 빨간 간선은 T에 속하고, 화살표는 T의 안전한 간선을 가리키며, 점선 간선은 쓸모없는 간선입니다.

♥야르니크 알고리즘 개선하기

1984년 Michael Fredman과 로버트 Tarjan이 처음 기술한 ***피보나치 힙***이라는 더 복잡한 우선순위 큐 자료구조를 사용하면 야르니크 알고리즘을 개선할 수 있습니다. 이진 힙과 마찬가지로 피보나치 힙은 표준 우선순위 큐 연산인 INSERT, EXTRACTMIN, DECREASEKEY를 지원합니다. 그러나 모든 연산에 $O(\log n)$ 시간이 필요한 표준 이진 힙과 달리, 피보나치 힙은 INSERT와 DECREASEKEY를 *분할 상환* 상수 시간에 지원합니다. EXTRACTMIN의 분할 상환 비용은 여전히 $O(\log n)$입니다.[4]

이 더 빠른 자료구조를 적용하기 위해, 우선순위 큐에 간선 대신 G의 *정점*을 저장하며, 각 정점 v의 우선순위는 v와 성장 중인 트리 T 사이의 가중치가 최소인 간선이거나, 그러한 간선이 없으면 ∞입니다. 알고리즘을 시작할 때 모든 정점을 우선순위 큐에 INSERT할 수 있습니다. 그런 다음 T에 새 간선을 추가할 때마다 일부 이웃 정점의 우선순위를 낮춰야 할 수도 있습니다.

설명을 쉽게 하기 위해 알고리즘을 두 부분으로 나눕니다. JARNÍKINIT은 우선순위 큐를 초기화하고, JARNÍKLOOP는 주 알고리즘입니다. 입력은 그래프의 정점과 간선, 그리고 시작 정점 s로 이루어집니다. 각 정점 v에 대해 우선순위 $priority(v)$와 함께 $w(edge(v)) = priority(v)$를 만족하는 인접 간선 $edge(v)$를 유지합니다.

```
JARNÍK(V, E, s):
  JARNÍKINIT(V, E, s)
  JARNÍKLOOP(V, E, s)
```

[4]분할 상환 시간은 자료구조 연산 하나에 걸리는 시간이 가끔 들쭉날쭉하더라도 이를 무시할 수 있게 해 주는 회계상의 기법입니다. 피보나치 힙은 I번의 INSERTS, D번의 DECREASEKEYS, X번의 EXTRACTMINS가 임의로 섞인 어떤 연산 수열이든 최악의 경우 $O(I + D + X \log n)$ 시간에 실행할 수 있습니다. 따라서 *평균적인* INSERT와 *평균적인* DECREASEKEY는 각각 상수 시간이 걸리고, *평균적인* EXTRACTMIN은 $O(\log n)$ 시간이 걸립니다. 하지만 최악의 경우에는 일부 개별 연산이 더 오래 걸릴 수 있습니다. 분할 상환은 연산 수열 전체에 대한 *통계적* 평균을 사용합니다. 여기서는 입력 데이터에도, 알고리즘에도 무작위성을 가정하지 않습니다.

```
JarníkInit(V, E, s):
  for each vertex v ∈ V ∖ {s}
      if vs ∈ E
          edge(v) ← vs
          priority(v) ← w(vs)
      else
          edge(v) ← Null
          priority(v) ← ∞
      Insert(v)
```

```
JarníkLoop(V, E, s):
  T ← ({s}, ∅)
  for i ← 1 to |V| − 1
      v ← ExtractMin
      add v and edge(v) to T
      for each neighbor u of v
          if u ∉ T and priority(u) > w(uv)
              edge(u) ← uv
              DecreaseKey(u, w(uv))
```

그림 7.5. 피보나치 힙과 함께 사용할 수 있도록 준비된 야르니크의 최소 신장 트리 알고리즘

Insert와 ExtractMin 연산은 s를 제외한 각 정점마다 한 번씩, 각각 $O(V)$번 호출되며, DecreaseKey는 각 간선마다 최대 두 번씩, $O(E)$번 호출됩니다. 따라서 피보나치 힙을 사용하면 개선된 알고리즘은 $O(E + V \log V)$ *시간*에 실행되며, 이는 $E = O(V)$가 아닌 한 보루프카 알고리즘보다 빠릅니다. 그러나 실제로는 그래프가 극도로 크고 조밀하지 않은 한, 이 개선이 이진 힙을 사용하는 단순한 구현보다 빠른 경우는 드뭅니다. 피보나치 힙 알고리즘은 상당히 복잡하며, 실행 시간과 공간 모두에 숨겨진 상수가 상당히 큽니다. 터무니없을 정도는 아니지만, 이진 힙 연산의 $O(\log n)$ 시간 한계에 숨겨진 상수 1보다는 분명히 큽니다.

7.5 크러스컬 알고리즘

마지막으로 살펴볼 최소 신장 트리 알고리즘은 1956년에 조지프 크러스컬(Joseph Kruskal)이 처음 기술한 것으로, 그는 같은 논문에서 야르니크(Jarnik)의 알고-

rithm. 크루스칼은 프린스턴 수학과에 "나돌던" 보루프카의 원 논문의 "(출처가 불분명한) 타자본 번역"에서 동기를 얻었습니다. 크루스칼은 보루프카의 알고리즘이 "불필요하게 복잡하다"고 여겼습니다.[5] 같은 알고리즘이 1957년에 해리 로버먼과 아널드 와인버거에 의해 재발견되었지만, 어찌 된 일인지 그들의 이름을 따서 다시 명명되는 일은 피했습니다.

앞서 살펴본 최소 신장 트리 알고리즘들과 마찬가지로, 크루스칼 알고리즘도 기억하기 쉬운 한 줄 설명으로 요약할 수 있습니다:

> Kruskal: 모든 간선을 가중치가 증가하는 순서로 살펴보면서, 간선이 안전하면 그 간선을 F에 추가합니다.

간선을 가중치의 오름차순으로 훑는 가장 간단한 방법은 $O(E \log E)$ 시간에 간선들을 가중치 기준으로 *정렬*한 뒤, 정렬된 간선 목록에 대해 단순한 for 루프를 사용하는 것입니다. 곧 살펴보겠지만, 이 사전 정렬 단계가 알고리즘의 실행 시간을 좌우합니다.

[5] 공정하게 말하자면, 보루프카의 첫 번째 논문은 *정말로* 불필요하게 복잡했습니다. 그 이유 중 하나는 이 논문이 그래프의 언어가 아니라 (선형) 대수학의 형식 언어로 수학자들을 위해 쓰였기 때문입니다. 보루프카의 후속 논문은 마찬가지로 1927년에 발표되었지만 전기공학 학술지에 실렸으며, 훨씬 더 폭넓은 독자를 위해 평이한 언어로 쓰여 사실상 오늘날의 현대적인 형태를 갖추고 있었습니다. 크루스칼은 보루프카의 두 번째 논문을 알지 못했던 것으로 보입니다. 망할 철의 장막 탓입니다.

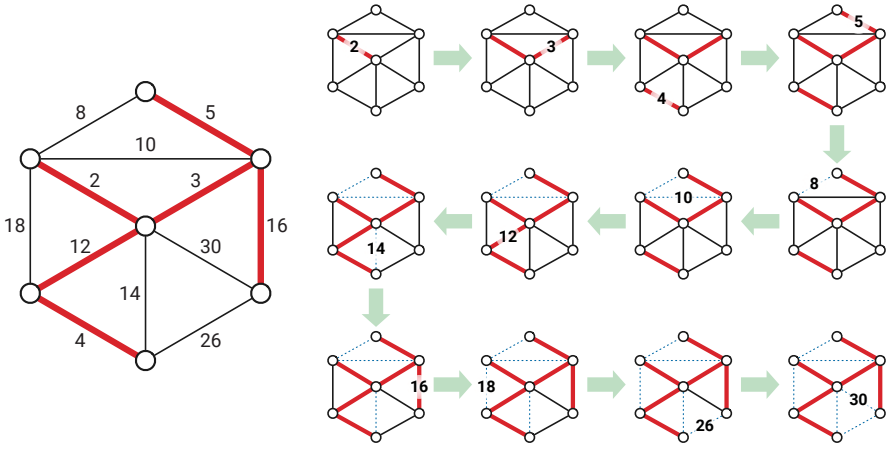


그림 7.6. 예제 그래프에서 크러스컬 알고리즘을 실행한 모습입니다. 굵은 빨간색 간선은 F에 속하며, 가는 점선 간선은 쓸모없는 간선입니다.

간선을 가장 가벼운 것부터 가장 무거운 것 순서로 검사하므로, 검사하는 간선은 양 끝점이 숲 F의 서로 다른 성분에 속할 때, 그리고 그때에만 안전합니다. 두 성분 A와 B를 잇지만 안전하지 않은 간선 e를 만났다고 가정합시다. 그러면 정확히 한 끝점만 A에 속하는 더 가벼운 간선 e'이 반드시 존재해야 합니다. 그러나 이는 불가능합니다. (귀납적으로) 이전에 검사한 모든 간선은 두 끝점이 모두 F의 같은 성분에 속하기 때문입니다.

보루프카 알고리즘에서와 마찬가지로, F의 각 정점은 자신이 F의 어느 컴포넌트에 속하는지 "알아야" 합니다. 그러나 보루프카 알고리즘과 달리, 우리는

간선을 추가할 때마다 모든 성분 레이블을 처음부터 다시 계산하지 않습니다. 대신 두 성분이 간선으로 연결되면, 더 작은 성분이 더 큰 성분의 레이블을 물려받습니다. 즉, (아무거나 우선 탐색을 통해) 더 작은 성분을 순회합니다. 이 순회는 더 작은 성분의 각 정점마다 $O(1)$ 시간이 걸립니다. 정점의 성분 레이블이 바뀔 때마다 그 정점을 포함하는 F의 성분은 적어도 2배로 커집니다. 따라서 각 정점 레이블은 최대 $O(\log V)$번 바뀝니다. 그러므로 정점 레이블을 갱신하는 데 드는 총 시간은 $O(V \log V)$에 불과합니다.

더 일반적으로 말하면, 크루스칼 알고리즘은 다음 연산들을 지원하는 자료구조를 사용하여 G의 정점들을 서로소인 부분집합들(우리의 경우에는 F의 연결 요소들)로 나눈 분할을 유지합니다.

- MakeSet(v) — 정점 v만을 포함하는 집합을 생성합니다.
- Find(v) — v를 포함하는 집합의 고유한 식별자를 반환합니다.
- Union(u, v) — u와 v를 포함하는 두 집합을 그 합집합으로 대체합니다. (이 연산은 집합의 개수를 줄입니다.) 다음은 이 연산들을 이용하여 크러스컬 알고리즘을 완전하게 기술한 것입니다:

```
KRUSKAL(V, E):
  sort E by increasing weight
  F ← (V, ∅)
  for each vertex v ∈ V
      MAKESET(v)
  for i ← 1 to |E|
      uv ← ith lightest edge in E
      if FIND(u) ≠ FIND(v)
          UNION(u, v)
          add uv to F
  return F
```

처음 정렬을 마친 뒤 알고리즘은 정확히 V번의 MakeSet 연산(정점마다 한 번), $2E$번의 Find 연산(간선마다 두 번), $V-1$번의 Union 연산(최소 신장 트리의 간선마다 한 번)을 수행합니다. 방금 MakeSet과 Find가 $O(1)$ 시간에 수행되고 Union이 $O(\log V)$ *분할 상환* 시간에 수행되는 서로소 집합 자료구조를 설명했습니다. 이 구현을 사용하면 집합 분할을 유지하는 데 드는 전체 시간은 $O(E + V \log V)$입니다.[6]

그러나 간선을 정렬하는 데만도 이미 $O(E \log E) = O(E \log V)$ 시간이 필요하다는 점을 기억하십시오. 이는 Union-Find 자료구조를 유지하는 데 드는 시간보다 크므로, 크러스컬 알고리즘의 전체 실행 시간은 $O(E \log V)$입니다,

Borůvka 알고리즘, 또는 일반적인(피보나치 힙이 아닌) 힙을 사용하는 Jarník 알고리즘과 정확히 같습니다.

연습문제

1. $G = (V, E)$를 간선에 가중치가 있는 임의의 연결 그래프라고 합시다.

   (a) G의 임의의 사이클에 대해, G의 최소 신장 트리는 그 사이클에서 가중치가 가장 큰 간선을 *포함하지 않음*을 증명하십시오.

   (b) 다음을 증명하거나 반증하십시오: G의 최소 신장 트리는 G의 모든 사이클 각각에서 가중치가 최소인 간선을 포함합니다.

2. 이 장 전체에 걸쳐 우리는 입력 그래프의 어떤 두 간선도 가중치가 같지 않다고 가정했으며, 이로부터 최소 신장 트리가 유일하다는 사실이 따라 나옵니다. 사실 간선 가중치에 대한 더 약한 조건만으로도 MST의 유일성이 보장됩니다.

   (a) 두 간선의 가중치가 같은데도 최소 신장 트리가 유일한 간선 가중 그래프를 설명하십시오.

[6] *경로 압축을 사용하는 랭크 기반 합집합*이라는 전략을 사용하는 다른 서로소 집합 자료구조는 각 Union 또는 Find를 $O(\alpha(V))$ 분할 상환 시간에 수행합니다. 여기서 α는 거의 상수이지만 완전한 상수는 아닌 *역 아커만 함수*입니다. 위키백과를 찾아보고 싶지 않다면 $\alpha(V)$를 그냥 4라고 생각하십시오. 이 구현을 사용하면 집합 분할을 유지하는 데 드는 전체 시간은 $O(E\alpha(V))$이며, V가 크고 E가 V에 매우 가까울 때 조금 더 빠릅니다.

(b) 간선 가중 그래프 G가 *유일한* 최소 신장 트리를 가질 필요충분조건은 다음 조건들이 성립하는 것임을 증명하십시오.

- G의 정점을 두 부분집합으로 나누는 임의의 분할에 대해, 각 부분집합에 끝점이 하나씩 있는 간선 중 가중치가 최소인 간선은 유일합니다.
- G의 어떤 사이클에서든 가중치가 최대인 간선은 유일합니다.

(c) 그래프의 최소 신장 트리가 유일한지 판정하는 알고리즘을 설명하고 분석하십시오.

3. 대부분의 고전적인 최소 신장 트리 알고리즘은 본문에서 설명한 "안전한" 간선과 "쓸모없는" 간선이라는 개념을 사용하지만, 다른 정식화도 있습니다. G를 간선 가중치가 모두 서로 다른 가중 무방향 그래프라고 합시다. 간선 e가 G의 어떤 사이클에서 가장 긴 간선이면 e를 ***위험하다***고 하고, G의 어떤 사이클에도 속하지 않으면 ***유용하다***고 합니다.

(a) G의 최소 신장 트리가 모든 유용한 간선을 포함함을 증명하십시오.

(b) G의 최소 신장 트리에는 위험한 간선이 하나도 포함되지 않음을 증명하십시오.

(c) 조지프 크루스칼(Joseph Kruskal)이 "크루스칼 알고리즘"을 제안한 바로 그 1956년 논문에서 처음 기술한 다음 알고리즘을 효율적으로 구현하는 방법을 서술하고 분석하십시오. G의 간선들을 *내림차순*으로 살펴보면서, 어떤 간선이 위험하면 그 간선을 G에서 제거합니다. *[힌트: 크루스칼의 일반적인 알고리즘만큼 빠르지는 않을 것입니다.]*

4. (a) 주어진 간선 가중 그래프의 *최대* 가중치 신장 트리를 계산하는 알고리즘을 설명하고 분석하십시오.

(b) 무방향 그래프 G의 *피드백 간선 집합*은 G의 모든 사이클이 F에 속한 간선을 적어도 하나 포함하는 간선의 부분집합 F입니다. 다시 말해, F에 속한 모든 간선을 제거하면 그래프 G는 비순환 그래프가 됩니다. 주어진 간선 가중 그래프에서 가중치가 최소인 피드백 간선 집합을 계산하는 빠른 알고리즘을 설계하고 분석하십시오.

5. 간선에 가중치가 있는 무방향 그래프 G와 G의 최소 신장 트리 T가 모두 주어졌다고 가정합시다.

(a) 간선 e 하나의 가중치가 감소했을 때 최소 신장 트리를 갱신하는 알고리즘을 설명하십시오.

(b) 간선 e 하나의 가중치가 증가했을 때 최소 신장 트리를 갱신하는 알고리즘을 설명하십시오.

두 경우 모두 알고리즘의 입력은 간선 e와 그 간선의 새 가중치입니다. 알고리즘은 T가 여전히 최소 신장 트리가 되도록 T를 수정해야 합니다. *[힌트: $e \in T$인 경우와 $e \notin T$인 경우를 나누어 생각하십시오.]*

6. (a) 주어진 그래프 G의 *두 번째로 작은 신장 트리*, 즉 최소 신장 트리를 제외하고 총 가중치가 가장 작은 G의 신장 트리를 찾는 알고리즘을 설명하고 분석하십시오.

♥(b) 가중치가 있는 무방향 그래프 G와 정수 k가 주어졌을 때, G의 신장 트리 중 가중치가 가장 작은 k개를 구하는 효율적인 알고리즘을 설명하고 분석하십시오.

7. $E = \Theta(V^2)$이면 그래프 $G = (V, E)$는 *조밀*하다고 합니다. 입력 그래프가 조밀할 때 기본적인 자료구조만을 사용하여—특히 피보나치 힙을 사용하지 *않고*—$O(V^2)$ 시간(E와 무관하게)에 실행되도록 야르니크의 최소 신장 트리 알고리즘을 수정하는 방법을 설명하십시오. 야르니크 알고리즘의 이 변형은 1958년에 Edsger Dijkstra가 처음 기술했습니다.

8. 최소 신장 트리 알고리즘은 흔히 *간선 축약*이라는 연산을 사용하여 기술됩니다. 간선 uv를 축약하려면 새 노드를 하나 삽입하고, u 또는 v에 접하는 모든 간선(uv는 제외)을 이 새 노드로 다시 연결한 다음, u와 v를 삭제합니다. 축약 후에는 새 노드와 그래프의 다른 노드들 사이에 여러 개의 평행 간선이 생길 수 있으며, 이 경우 임의의 두 노드 사이에서 가장 가벼운 간선 하나만 남기고 나머지는 모두 제거합니다.

이 장에서 설명하는 세 가지 고전적인 최소 신장 트리 알고리즘은 모두 다음과 같이 축약의 관점에서 깔끔하게 표현할 수 있습니다. 세 알고리즘 모두 입력 그래프 G의 깨끗한 사본 G'를 만드는 것으로 시작한 다음, G'에서 안전한 간선을 반복해서 축약합니다. 최소 신장 트리는 축약된 간선들로 이루어집니다.

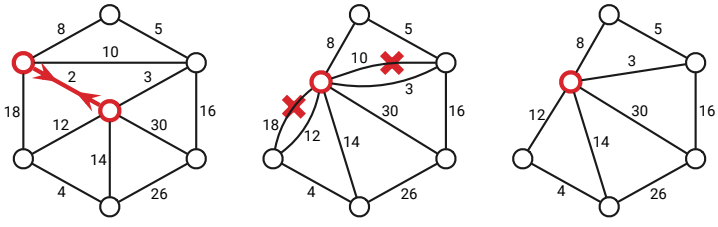


그림 7.7. 간선을 축약하고 중복되는 평행 간선을 제거합니다.

- Borůvka: 각 정점에서 나가는 간선 중 가장 가벼운 간선을 표시하고, 표시된 모든 간선을 축약한 다음 재귀합니다.
- Jarník: 미리 고정한 루트 정점에 접하는 간선 중 가장 가벼운 간선을 반복해서 축약합니다.
- Kruskal: 그래프에서 가장 가벼운 간선을 반복해서 축약합니다.

(a) $O(V + E)$ 시간에 Borůvka 축약 알고리즘의 단일 패스를 실행하는 알고리즘을 설명하십시오. 입력 그래프는 인접 리스트로 표현됩니다.

(b) 먼저 Borůvka의 축약 알고리즘을 k번 수행한 다음, 그 결과로 얻은 축약 그래프에 대해 (피보나치 힙을 *사용하는*) Jarník의 알고리즘을 실행하는 알고리즘을 생각해 봅시다.

i. 이 하이브리드 알고리즘의 실행 시간을 V, E, k에 대한 함수로 나타내면 어떻게 됩니까?

ii. k가 어떤 값일 때 이 실행 시간이 최소가 됩니까? 그때의 실행 시간은 얼마입니까?

(c) 다음 성질을 만족하는 그래프 족을 *좋은* 족이라고 부르겠습니다:

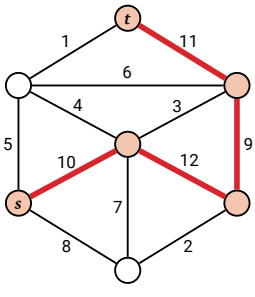
- 좋은 그래프의 간선 하나를 축약하면 또 다른 좋은 그래프가 됩니다.
- 정점이 V개인 모든 좋은 그래프에는 간선이 $O(V)$개밖에 없습니다.

예를 들어, 평면 그래프—간선이 서로 교차하지 않도록 평면에 그릴 수 있는 그래프—는 다루기 좋은 그래프입니다. 평면 그래프의 어떤 간선을 축약하더라도 더 작은 평면 그래프가 남으며, 오일러 공식에 따르면 정점이 V개인 모든 평면 그래프의 간선은 최대 $3V - 6$개입니다.

보루브카 축약 알고리즘이 임의의 좋은 그래프의 최소 신장 트리를 $O(V)$ 시간에 계산함을 증명하십시오.

9. 무방향 가중 그래프 G에서 두 정점 s와 t 사이의 경로를 생각해 봅시다. 이 경로의 *폭*은 경로에 있는 간선의 가중치 중 최솟값입니다. s와 t 사이의 *병목 거리*는 s에서 t로 가는 경로 중 폭이 가장 넓은 경로의 폭입니다. (s에서 t로 가는 경로가 없으면 병목 거리는 $-\infty$이고, 반면에 s에서 자기 자신까지의 병목 거리는 ∞입니다.)

(a) G의 *최대* 신장 트리가 모든 정점 쌍 사이의 가장 넓은 경로를 포함함을 증명하십시오.



s와 t 사이의 병목 거리는 9입니다.

(b) 다음 문제를 $O(V + E)$ 시간에 해결하는 알고리즘을 설명하십시오: 무방향 가중 그래프 G, 두 정점 s와 t, 그리고 가중치 W가 주어질 때, s와 t 사이의 병목 거리가 W 이하입니까?

(c) B가 s와 t 사이의 병목 거리라고 가정합시다.

i. 가중치가 B보다 작은 간선을 어느 것이든 삭제해도 s와 t 사이의 병목 거리가 변하지 않음을 증명하십시오.

ii. 가중치가 B보다 큰 간선을 하나 *축약*해도 s와 t 사이의 병목 거리가 변하지 않음을 증명하십시오. (축약으로 평행 간선이 생기면 각 노드 쌍 사이에서 *가장 무거운* 간선만 남기고 나머지는 모두 삭제하십시오.)

♥(d) s와 t 사이의 최소 병목 경로를 $O(V + E)$ 시간에 계산하는 알고리즘을 설명하십시오. *[힌트: 먼저 G에서 가중치가 중앙값인 간선을 찾으십시오.]*

10. 보루프카 알고리즘은 각 반복에서 변화하는 신장 숲 F의 연결 요소를 명시적으로 세고 레이블을 붙이는 대신, 크러스컬 알고리즘과 마찬가지로 표준 서로소 집합 자료구조를 사용하여 안전한 간선을 식별하도록 재정식화할 수 있습니다.

이 변형에서는 F의 각 성분을 *업트리(up-tree)*로 표현합니다. 각 정점 v는 자신의 부모를 가리키는 포인터 $parent(v)$를 저장하며, v가 자기 업트리의 루트이면 이 포인터는 v 자신을 가리킵니다. 서브루틴 FIND(v)는 v가 속한 업트리의 루트를 반환할 뿐 아니라 *경로 압축(path compression)*도 적용합니다. 즉, 이후의 FIND 연산을 빠르게 하기 위해 v에서 루트까지의 경로에 있는 모든 부모 포인터가 루트를 직접 가리키도록 다시 지정합니다.[7] 서브루틴 UNION은 두 루트 노드 중 하나를 다른 하나의 부모로 만들어 두 업트리를 하나로 합칩니다.[8]

```
FIND(v):
  if parent(v) = v
      return v
  else
      v̄ ← FIND(parent(v))
      parent(v) ← v̄
      return v̄
```

```
UNION(u, v):
  ū ← FIND(u)
  v̄ ← FIND(v)
  either
      parent(ū) ← v̄
  or
      parent(v̄) ← ū
```

수정된 Borůvka 알고리즘에서는 부모 포인터 외에도 F의 각 컴포넌트의 루트 정점 $\bar{v}$가 간선 $safe(\bar{v})$를 유지합니다. 이 간선은 (FINDSAFEEDGES가 끝났을 때) 끝점 하나만 해당 컴포넌트에 있는 간선 중 가장 가벼운 간선입니다.

[7] 경로 압축은 메모이제이션의 한 형태입니다!

[8] 보통은 더 크거나 더 오래된 업트리의 루트가 바뀌지 않도록 UNION을 더 신중하게 구현합니다. 하지만 여기서는 그런 세부 사항이 중요하지 않습니다.

```
FindSafeEdges(V, E):
  for each vertex v ∈ V
      safe(v) ← Null
  found ← False
  for each edge uv ∈ E
      ū ← Find(u)
      v̄ ← Find(v)
      if ū ≠ v̄
          if safe(ū) = Null or w(uv) < w(safe(ū))
              safe(ū) ← uv
          if safe(v̄) = Null or w(uv) < w(safe(v̄))
              safe(v̄) ← uv
          found ← True
  return found
```

```
AddSafeEdges(V, E, F):
  for each vertex v ∈ V
      if safe(v) ≠ Null
          xy ← safe(v)
          if Find(x) ≠ Find(y)
              Union(x, y)
              add xy to F
```

```
Borůvka(V, E):
  F = ∅
  for each vertex v ∈ V
      parent(v) ← v
  while FindSafeEdges(V, E)
      AddSafeEdges(V, E, F)
  return F
```

FindSafeEdges와 AddSafeEdges를 호출할 때마다 $O(E)$ 시간만 걸린다는 것을 증명하십시오. *[힌트: FindSafeEdges가 끝날 때 업트리(up-tree)의 깊이는 얼마입니까?]* 이로부터 이 Borůvka 변형 역시 $O(E \log V)$ 시간에 실행됨을 알 수 있습니다.

저는 사과를 따듯이 성경을 공부합니다. 먼저 가장 잘 익은 것들이 떨어지도록 나무 전체를 흔듭니다. 그다음 나무에 올라가 큰 가지를 하나하나 흔들고, 이어서 작은 가지와 잔가지를 차례로 하나하나 흔든 뒤, 마지막으로 잎사귀 하나하나 밑을 들여다봅니다.

— 마르틴 루터의 말로 전해짐 (c. 1500)

삶은 펼쳐져 가는 과정이며, 멀리 나아갈수록 우리가 이해할 수 있는 진리도 많아집니다. 우리 문 앞에 있는 것들을 이해하는 일이야말로 그 너머에 있는 것들을 이해하기 위한 최선의 준비입니다.

— 엘버트 허버드가 알렉산드리아의 히파티아(약 400년)의 말이라고 전함
위대한 스승들의 집으로 떠나는 작은 여행(1908)에서

긴장을 풀고 답을 기다리는 법을 익히면, 여러분의 마음은 대부분의 질문에 답해 줄 것입니다. 저 생각하는 기계들처럼, 질문을 집어넣고 느긋하게 기대앉아 기다리십시오 …

— William S. Burroughs, *Naked Lunch* (1959)

이 논문에서 제시하는 방법들은 어떠한 선견지명이나 독창성도 필요로 하지 않으므로, 알고리즘이라고 불릴 만합니다.

— Edward R. Moore, "미로를 통과하는 최단 경로(The Shortest Path Through a Maze)" (1959)

8

최단 경로

두 개의 특별한 정점이 있는 가중 *방향* 그래프 $G = (V, E, w)$가 주어졌고, *출발* 정점 s에서 *대상* 정점 t까지의 최단 경로를 찾으려 한다고 가정합시다. 즉, s에서 시작하여 t에서 끝나는 방향 경로 P 중에서 다음 함수를 최소화하는 경로를 찾으려 합니다.

$$w(P) := \sum_{u \to v \in P} w(u \to v).$$

예를 들어 "일리노이주 샴페인에 있던 제 옛 아파트에서 오하이오주 콜럼버스에 있던 아내의 옛 아파트까지 차로 가장 빨리 가는 방법은 무엇인가?"라는 질문에 답하고 싶다면, 정점은 도시, 간선은 도로, 가중치는 주행 시간이고 s는 샴페인, t는 콜럼버스인 그래프를 사용할 수 있습니다.[1] 같은 도로라도 방향에 따라 주행 시간이 다를

[1] 처치 가에서 서쪽, 프로스펙트 가에서 북쪽, I-74에서 동쪽, I-465에서 남쪽, 에어포트 고속도로에서 동쪽, I-65에서 북쪽, I-70에서 동쪽, 그랜드뷰 가에서 북쪽, 5번가에서 동쪽, 올렌탠지 리버 가에서 북쪽, 도드리지 가에서 동쪽, 하이 가에서 북쪽, 켈소 가에서 서쪽, 닐 가에서 남쪽으로 갑니다. 교통 상황에 따라 다릅니다. 지금 저희는 어바나에 삽니다.

수 있으므로 이 그래프는 방향 그래프입니다. (한때 인디애나/오하이오 주 경계 바로 동쪽의 I-70에 과속 단속 구간이 있었는데, 동쪽으로 가는 차량만 단속했습니다.)

8.1 최단 경로 트리

한 정점에서 다른 정점으로 가는 최단 경로를 계산하는 알려진 알고리즘은 거의 모두 실제로는 다음과 같은 더 일반적인 ***단일 출발점 최단 경로***, 즉 ***SSSP*** 문제(의 상당 부분)를 풉니다. 출발 정점 s에서 그래프의 *다른* 모든 정점으로 가는 최단 경로를 찾으십시오. 이 문제는 보통 원하는 최단 경로를 모두 포함하면서 s를 루트로 하는 ***최단 경로 트리***를 찾는 방식으로 풉니다.

최단 경로가 유일하다면 그 경로들이 트리를 이룬다는 것은 어렵지 않게 알 수 있습니다. 최단 경로의 모든 부분 경로는 그 자체로 최단 경로이기 때문입니다. 어떤 정점들로 가는 최단 경로가 여러 개 있다면, 각 정점마다 최단 경로를 하나씩 골라 그 경로들의 합집합이 트리가 되도록 언제나 할 수 있습니다. s에서 두 정점 u와 v로 가는 최단 경로들이 갈라졌다가 다시 만나고 또다시 갈라진다면, 두 경로 중 하나를 길이를 바꾸지 않고 수정하여 두 경로가 한 번만 갈라지게 할 수 있습니다.

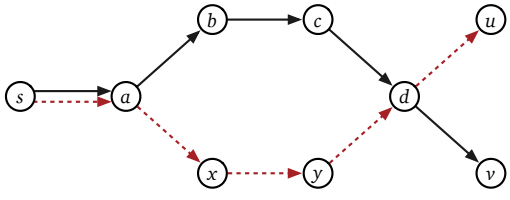


그림 8.1. $s \to a \to b \to c \to d \to v$(실선)와 $s \to a \to x \to y \to d \to u$(점선)가 최단 경로이면, $s \to a \to b \to c \to d \to u$(위쪽을 따라가는 경로)도 최단 경로입니다.

최단 경로 트리와 최소 신장 트리는 둘 다 최적 신장 트리이지만 성격이 매우 다릅니다. 최단 경로 트리에는 루트와 방향이 있고, 최소 신장 트리는 루트 없는 무방향 트리입니다. 최단 경로 트리는 방향 그래프에서 정의하는 것이 가장 자연스럽고, 최소 신장 트리는 무방향 그래프에서 정의하는 것이 더 자연스럽습니다. 간선 가중치가 모두 서로 다르면 최소 신장 트리는 하나뿐이지만, 최단 경로 트리는 출발 정점마다 다르게 정해집니다. 게다가 모든 최단 경로 트리가 최소 신장 트리와 다른 간선 집합을 사용할 수도 있습니다.

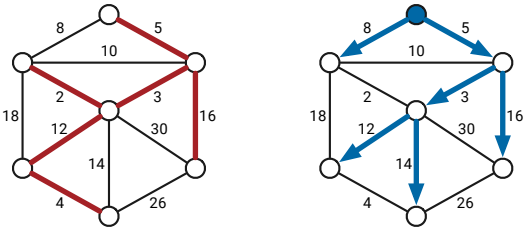


그림 8.2. 같은 무방향 그래프의 최소 신장 트리와 최단 경로 트리입니다.

8.2 ♥음수 간선

간선 가중치가 거리나 길이 또는 시간에 대응하는 대부분의 최단 경로 문제에서는 모든 간선 가중치가 음이 아니거나 더 나아가 양수라고 가정하는 것이 자연스럽습니다. 그러나 최단 경로 알고리즘의 여러 응용에서는 음의 가중치를 가진 간선을 고려하는 것이 자연스럽습니다. 예를 들어 간선의 가중치는 한 정점에서 다른 정점으로 이동하는 *비용*을 나타낼 수 있습니다. 이 경우 음의 가중치를 가진 간선은 비용이 음수인 전이, 다시 말해 이익을 얻는 전이를 나타냅니다.

음수 간선은 대부분의 최단 경로 문제에서 골칫거리입니다. 음수 *사이클*이 있으면 최단 경로가 잘 정의되지 않을 수 있기 때문입니다. 정확히 말하면, s에서 t로 가는 최단 경로는 s에서 t로 가는 경로가 적어도 하나 있으면서 음수 사이클에 닿는 s에서 t로의 경로는 하나도 없을 때, 그리고 오직 그때에만 존재합니다. 음수 사이클에 닿는 s에서 t로의 경로가 *어떤* 것이든, 그 사이클을 한 바퀴 더 도는 더 짧은 s에서 t로의 경로가 존재합니다.[2] 따라서 s에서 t로 가는 경로 중 적어도 하나가 음수 사이클에 닿는다면, s에서 t로 가는 최단 경로는 존재하지 않습니다.

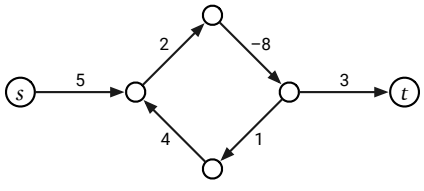


그림 8.3. s에서 t로 가는 최단 보행은 존재하지 않습니다.

음수 간선 가중치를 고려해야 한다는 점이 한 가지 이유가 되어, 이 장에서는 명시적으로 방향 그래프*만*을 다룹니다. 여기서 설명하는 모든 알고리즘은 사실상 사소한 수정만 거치면 무방향 그래프에서도 동작하지만, 이는 음수 간선이 금지된 경우*에 한해서만* 그렇습니다. 무방향 그래프에서 음수 간선을 올바르게 처리하는 것은 훨씬 더 까다로운 문제입니다. 모든 무방향 간선을 한 쌍의 방향 간선으로 단순히 대체할 수는 없는데, 그렇게 하면 모든 음수 간선이 짧은 음수 사이클로 바뀌기 때문입니다. 음수 간선을 포함하는 *무방향* 최단 경로의 부분 경로가 반드시 최단 경로인 것은 *아닙니다*. 따라서 최단 경로가 유일하더라도, 하나의 시작 정점에서 출발하는 모든 무방향 최단 경로의 집합은 트리를 이루지 않을 수 있습니다.

음수 간선이 있는 무방향 그래프를 완전히 다루는 것은 이 책의 범위를 벗어납니다. Google로 더 찾아보고 싶은 분들을 위해, 음수 간선이 있는 무방향 그래프에서 *단일* 최단 경로는 최대 가중치 매칭으로의 환원을 통해 $O(VE + V^2 \log V)$ 시간에 계산할 수 있다는 점만 언급해 두겠습니다.

[2] 엄밀히 말하면 여기서는 최단 *경로*가 아니라 최단 *보행*을 논해야 하지만, 이런 용어 남용은 관례적입니다. s에서 t에 도달할 수 있다면 s에서 t로 가는 최단 단순 경로가 반드시 존재합니다. 다만 해밀턴 경로 문제로부터의 간단한 환원에 의해, (음수 사이클이 있을 때) 그것을 계산하는 것이 NP-난해할 뿐입니다. 반면에 s에서 t로 가는 최단 *보행*이 존재한다면, 그 보행은 반드시 단순 경로이며, 따라서 반드시 s에서 t로 가는 최단 단순 경로입니다. 에휴.

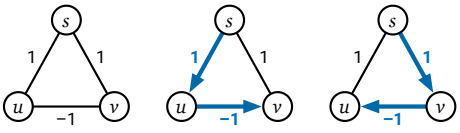


그림 8.4. *s*에서 출발하는 최단 경로가 유일하지만 트리를 이루지는 않는 무방향 그래프입니다.

8.3 유일한 SSSP 알고리즘

그래프 순회나 최소 신장 트리와 마찬가지로, 여러 다양한 SSSP 알고리즘은 하나의 일반적인 알고리즘의 특수한 경우로 기술할 수 있습니다. 이 알고리즘은 1956년 레스터 포드(Lester Ford)가 처음 제안하였고, 1957년 조지 단치그(George Dantzig)가 독립적으로 기술하였으며[3] 1958년 조지 민티(George Minty)가 다시 기술하였습니다. 그래프의 각 정점 v는 두 개의 값을 저장하며, 이 값들은 s에서 v로 가는 *잠정적인* 최단 경로를 (귀납적으로) 나타냅니다.

- $dist(v)$는 잠정적인 최단 $s \rightsquigarrow v$ 경로의 길이이며, 그러한 경로가 없으면 ∞입니다.
- $pred(v)$는 잠정적인 최단 $s \rightsquigarrow v$ 경로에서 v의 직전 정점이며, 그러한 정점이 없으면 NULL입니다.

선행자 포인터는 s를 루트로 하는 잠정적인 최단 경로 *트리*를 자동으로 정의합니다. 이 포인터들은 앞서 다룬 일반적인 그래프 순회 알고리즘의 부모 포인터와 정확히 같습니다. 알고리즘을 시작할 때 거리와 선행자를 다음과 같이 초기화합니다:

```
INITSSSP(s):
  dist(s) ← 0
  pred(s) ← NULL
  for all vertices v ≠ s
      dist(v) ← ∞
      pred(v) ← NULL
```

알고리즘을 실행하는 동안 $dist(u) + w(u \to v) < dist(v)$이면 간선 $u \to v$는 ***긴장*** 상태에 있다고 합니다. $u \to v$가 긴장 상태이면 잠정적인 최단 경로 $s \rightsquigarrow v$는 분명히 올바르지 않습니다. 경로 $s \rightsquigarrow u \to v$가 더 짧기 때문입니다. 다음과 같이 간선을 ***완화*** 하면 이 명백한 과대 추정을 바로잡을(또는 적어도 개선할) 수 있습니다:

```
RELAX(u→v):
  dist(v) ← dist(u) + w(u→v)
  pred(v) ← u
```

이제 모든 준비가 끝났으므로, 포드의 일반 알고리즘은 다음과 같이 한 줄로 간단히 설명할 수 있습니다:

> **긴장된 간선이 더 이상 없을 때까지 긴장된 간선을 반복해서 완화하십시오.**

[3] 구체적으로, 단치그는 최단 경로 문제를 선형 계획법 문제로 표현할 수 있음을 보인 뒤, 자신의 심플렉스 방법을 원래 그래프의 관점에서 해석하여 기술하였습니다. 그의 기술은 포드의 완화(relaxation) 전략과 (본질적으로) 동등합니다.

```
FordSSSP(s):
  InitSSSP(s)
  while there is at least one tense edge
      Relax any tense edge
```

FordSSSP가 (더 이상 긴장된 간선이 없어서) 결국 종료되면, 선행자 포인터는 최단 경로 트리를 올바르게 정의하며 각 값 $dist(v)$는 s에서 v까지의 실제 최단 경로 거리입니다. 특히 s에서 v에 도달할 수 없으면 $dist(v) = \infty$이고, s에서 도달할 수 있는 음수 사이클이 하나라도 있으면 알고리즘은 결코 종료되지 않습니다.

포드의 일반 알고리즘이 올바르다는 사실은 다음과 같은 일련의 더 간단한 주장들로부터 도출됩니다.

1. 알고리즘이 실행되는 동안 어느 시점에서든, 모든 정점 v에 대해 거리 $dist(v)$는 ∞이거나 s에서 v로 가는 보행(walk)의 길이입니다. 이 주장은 완화(relaxation) 횟수에 대한 귀납법으로 증명할 수 있습니다.

2. 그래프에 음수 사이클이 없으면 $dist(v)$는 ∞이거나 s에서 v로 가는 어떤 *단순 경로*의 길이입니다. 구체적으로 말하면, $dist(v)$가 방향 사이클을 포함하는 s에서 v로 가는 보행의 길이라면 그 사이클의 길이는 반드시 음수입니다. 이 주장에 따르면 G에 음수 사이클이 없을 때 G의 단순 경로는 유한 개뿐이므로 완화 알고리즘은 결국 멈춥니다.

3. G의 어떤 간선도 긴장 상태가 아니라면, 모든 정점 v에 대해 거리 $dist(v)$는 선행자 경로 $s \to \cdots pred(pred(v)) \to pred(v) \to v$의 길이입니다. 구체적으로, v는 이 조건을 위반하지만 그 선행자 $pred(v)$는 위반하지 않는다면 간선 $pred(v) \to v$는 긴장 상태입니다.

4. G의 어떤 간선도 긴장 상태가 아니라면, 모든 정점 v에 대해 선행자 간선들로 이루어진 경로 $s \to \cdots \to pred(pred(v)) \to pred(v) \to v$는 실제로 s에서 v로 가는 최단 경로입니다. 구체적으로 말하면, v는 이 조건을 위반하지만 *어떤 최단 경로에서의* v의 선행자 u는 위반하지 않는다면, 간선 $u \to v$는 긴장 상태입니다. 또한 이 주장에 따르면 G에 음수 사이클이 있을 경우 어떤 간선은 *항상* 긴장 상태이므로, 일반 알고리즘은 결코 멈추지 않습니다.

지금까지 긴장된 간선을 어떻게 찾는지, 또는 긴장된 간선이 둘 이상일 때 어느 간선(들)을 완화할지에 대해서는 아무것도 말하지 않았습니다. 아무거나 우선 탐색과 마찬가지로, 포드의 일반적인 완화 알고리즘에도 여러 가지 구체적인 형태가 있습니다. 그러나 아무거나 우선 탐색과 달리, 각 탐색 전략의 효율성과 정확성은 입력 그래프의 구조에 따라 달라집니다.

이 장의 나머지 부분에서는 포드 알고리즘을 구체화한 가장 흔한 네 가지 형태를 살펴봅니다. 각 형태는 서로 다른 부류의 입력 그래프에 가장 적합한 선택입니다. 일반적인 정확성 증명의 나머지 세부 사항은 연습 문제로 남겨 두고, 대신 이 네 가지 구체적인 알고리즘 각각에 대해 (더 많은 것을 알려 주고 그 자체로 완결된) 정확성 증명을 제시하겠습니다.

8.4 가중치 없는 그래프: 너비 우선 탐색

최단 경로 문제의 가장 단순한 특수한 경우에는 모든 간선의 가중치가 1이며, 경로의 길이는 단순히 간선의 개수입니다. 이 특수한 경우는 ***너비 우선 탐색***이라고 불리는, 우리의 일반적인 그래프 순회 알고리즘의 한 변종으로 풀 수 있습니다. 너비 우선 탐색은 흔히 Edward Moore의 업적으로 여겨지는데, 그는 1957년에 이를 미로를 통과하는 최단 경로를 찾는 최초로 발표된 방법으로서 ("알고리즘 A"라는 이름으로) 기술하였습니다.[4] 특히 VLSI 배선과 로봇 경로 계획의 맥락에서는 너비 우선 탐색이 때때로 Chin Yang Lee의 업적으로 여겨지기도 하는데, 그는 1961년에 Moore의 "알고리즘 A"의 여러 응용을 (Moore의 공로를 제대로 밝히며) 기술하였습니다. 그러나 Moore가 미로를 고찰하기 10여 년도 더 전인 1945년에, Konrad Zuse가 연결되지 않은 그래프의 연결 요소를 세고 레이블을 붙이는 방법으로서 너비 우선 탐색의 구현을 기술하였습니다.[6]

너비 우선 탐색은 정점들의 선입선출 큐를 유지하며, 이 큐에는 처음에 시작 정점 s만 들어 있습니다. 각 반복에서 알고리즘은 큐의 앞에서 정점 u를 PULLS하고, u의 나가는 간선 $u \to v$를 각각 검사합니다. 알고리즘은 나가는 긴장된 간선 $u \to v$를 발견할 때마다 그 간선을 완화하고 정점 v를 큐에 PUSHES합니다. 큐가 비면 알고리즘이 끝납니다.

```
BFS(s):
  InitSSSP(s)
  Push(s)
  while the queue is not empty
      u ← Pull()
      for all edges u→v
          if dist(v) > dist(u) + 1      《u→v가 긴장 상태이면》
              dist(v) ← dist(u) + 1     《u→v를 완화합니다》
              pred(v) ← u
              Push(v)
```

[4] Moore는 Claude Shannon이 1950년에 설계하고 제작한 미로 풀이 로봇 "Theseus"의 약점에서 동기를 얻었습니다. (Theseus는 전기기계식 계전기로 구현한, 메모이제이션을 적용한 깊이 우선 탐색을 사용하였습니다. 이는 거의 확실히 그래프에서 깊이 우선 탐색을 처음으로 구현한 사례입니다.) Moore에 따르면, "이 기계를 해가 둘 이상인 미로에 사용했을 때, 한 방문객이 왜 항상 최단 경로를 찾도록 만들지 않았느냐고 물었습니다. Shannon과 저는 각자 이를 기계로 수행하는 경제적인 방법을 찾으려고 시도하였습니다. 그는 아날로그 계산에 적합한 여러 방법을 찾아냈고,[5] 저는 이 알고리즘들을 얻었습니다."

[5] 미로를 통과하는 최단 경로를 계산하는 아날로그 방법으로는 볼 베어링, 유체/플라스마 흐름, 화학 반응파, 주화성, 저항 네트워크, LED가 달린 전기 회로, 멤리스터 네트워크, 미세유체 칩 내의 글로 방전, 자라는 식물, 점균류, 아메바, 개미, 벌, 선충, 그리고 관광객을 이용하는 방법들이 제안되었습니다.

[6] Konrad Zuse는 컴퓨팅의 초기 선구자 중 한 명입니다. 그는 1930년대 후반에 부모님 댁 거실에서 금속 띠와 막대로 자신의 첫 번째 프로그래밍 가능한 컴퓨터(훗날 Z1이라 불림)를 설계하고 제작하였으며, Z1과 그 원본 설계도는 1944년 영국군의 공습으로 파괴되었습니다. Zuse의 1945년 박사 학위 논문은 *Plankalkül*이라는 최초의 고급 프로그래밍 언어를 기술합니다. Zuse의 논문에 실린 최초의 완전한 Plankalkül 프로그램 예제는 연결 요소를 세는 너비 우선 탐색의 구현으로, 의사코드 설명과 함께 정점이 여덟 개인 연결되지 않은 그래프에서 알고리즘이 실행되는 과정을 그림으로 단계별로 추적한 내용이 수록되어 있습니다. 나치 정권의 붕괴로 인해 Zuse는 박사 학위 논문을 제출하지 못하였고, Plankalkül은 1972년까지 출판되지 않은 채로 남아 있었습니다. 최초의 Plankalkül 컴파일러는 마침내 1975년에 Joachim Hohmann에 의해 구현되었습니다.

너비 우선 탐색은 가상의 *토큰*을 도입하여 그 실행을 여러 *단계*로 나누면 분석하기가 다소 쉬워집니다. 어떤 정점도 Pull하기 전에 먼저 토큰을 큐에 Push합니다. 현재 단계는 토큰을 큐에서 Pull할 때 끝나며, 토큰을 다시 큐에 Push할 때 다음 단계가 시작됩니다. 따라서 첫 번째 단계는 전적으로 시작 정점 s를 스캔하는 것으로만 이루어집니다. 알고리즘은 큐에 토큰*만* 남아 있을 때 끝납니다. 수정된 알고리즘은 그림 8.5에 나와 있으며, 그림 8.6은 이 알고리즘이 실제로 동작하는 예를 보여 줍니다. 이러한 수정은 단지 분석의 편의를 위한 것일 뿐임을 강조해 두겠습니다. 토큰이 있든 없든 알고리즘은 같은 순서로 정점을 Push하고 Pull하며, 같은 순서로 간선을 스캔하고, 정확히 같은 거리와 선행 정점을 출력합니다.

```
BFSWithToken(s):
  InitSSSP(s)
  Push(s)
  Push(✠)                              《첫 번째 단계를 시작합니다》
  while the queue contains at least one vertex
      u ← Pull()
      if u = ✠
          Push(✠)                      《다음 단계를 시작하십시오》
      else
          for all edges u→v
              if dist(v) > dist(u) + 1     《u→v가 긴장 상태이면》
                  dist(v) ← dist(u) + 1    《u→v를 완화합니다》
                  pred(v) ← u
                  Push(v)
```

그림 8.5. 단계 종료 토큰(✠)을 사용하는 너비 우선 탐색입니다. 굵은 빨간색 선은 분석을 위해서만 표시한 것입니다.

다음 보조정리에서 $dist(v)$는 알고리즘이 유지하는 변수일 뿐이라는 점을 강조해 둡니다. $dist(v)$는 *직관적*으로는 잠정적인 최단 경로 거리를 나타내지만, $dist(v)$가 s에서 v까지의 진짜 최단 경로 거리와 실제로 같아지는 때가 있다고 (아직은) 가정할 수 없습니다. 걱정하지 마십시오. 곧 그 지점에 도달할 것입니다.

보조정리 8.1. *모든 정수 $i \geq 0$과 모든 정점 v에 대해, i번째 단계가 끝났을 때 $dist(v) = \infty$이거나 $dist(v) \leq i$이며, v가 큐에 있을 필요충분조건은 $dist(v) = i$입니다.*

증명: 증명은 i에 대한 귀납법으로 진행합니다. 기저 사례 $i = 0$은 간단합니다. 첫 번째 단계가 시작될 때("0번째 단계가 끝날 때") 큐에는 시작 정점 s와 토큰 ✠만 들어 있으며, InitSSSP가 방금 $dist(s) \leftarrow 0$으로, 그리고 모든 $v \neq s$에 대해 $dist(v) \leftarrow \infty$로 설정했습니다.

이제 정수 $i > 0$을 고정합시다. 귀납 가정에 따르면 i번째 단계가 *시작*될 때 큐에는 $dist(u) = i - 1$을 만족하는 모든 정점 u가 들어 있고, 그 뒤에 토큰 ✠가 이어집니다. 다시 말해 큐는 다음과 같은 모습입니다:

$$\rightarrow \boxed{\text{✠} \quad i-1 \quad i-1 \quad \cdots \quad i-1} \rightarrow$$

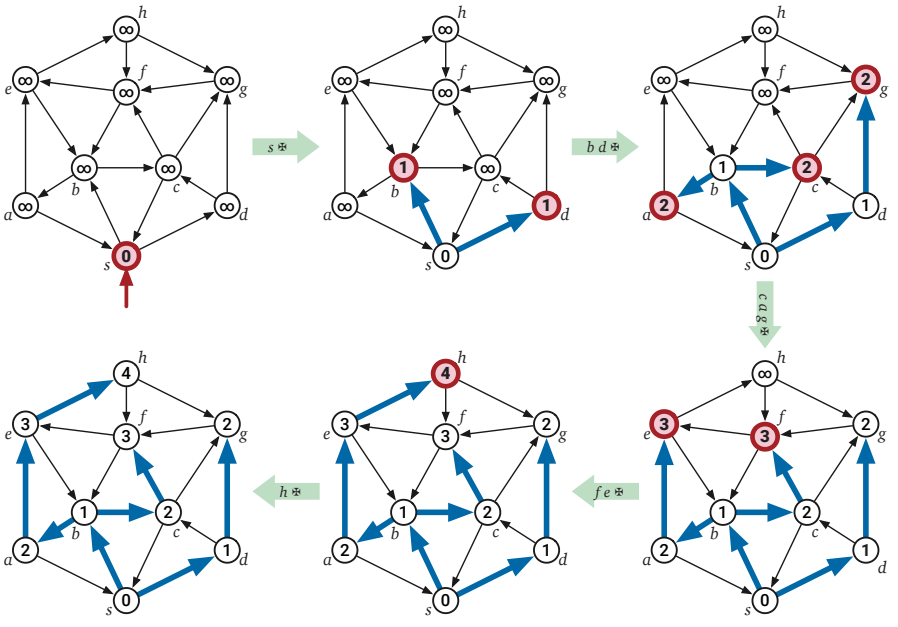


그림 8.6. 방향 그래프에서 너비 우선 탐색을 처음부터 끝까지 실행한 예입니다. 정점은 s ✠ b d ✠ c a g ✠ f e ✠ h ✠✠ 순서로 큐에서 꺼내지며, 여기서 ✠는 단계 종료 토큰입니다. 굵게 표시된 정점은 각 단계가 끝날 때 큐에 들어 있는 정점입니다. 굵게 표시된 간선은 점차 자라나는 최단 경로 트리를 나타냅니다.

따라서 큐에서 토큰 ✠를 Pull하여 i번째 단계를 끝내기 전에, $dist(u) = i-1$인 모든 정점 u를 Pull합니다.

이러한 각 정점 u에 대해 나가는 간선 $u \to v$를 모두 살펴봅니다. $u \to v$가 긴장되어 있으면 $dist(v) \leftarrow dist(u) + 1$로 설정하여 $dist(v) = i$가 되게 하고, 곧바로 v를 큐에 Push합니다. i번째 단계 동안 거리 레이블에 대한 대입은 이것뿐입니다. 따라서 귀납법에 의해, i번째 단계 전체에 걸쳐 큐에는 거리 레이블이 $i-1$인 정점 몇 개, 그 뒤에 토큰, 그 뒤에 거리 레이블이 i인 정점 몇 개가 차례로 들어 있습니다:

$$\to \overline{\; i \;\; \cdots \;\; i \;\; ✠ \;\; i-1 \;\; \cdots \;\; i-1 \;} \to$$

특히 i번째 단계가 끝나기 직전에는 큐에 토큰이 들어 있고, 그 뒤에 거리 레이블이 i인 정점 몇 개가 이어집니다.

$$\to \overline{\; i \;\; i \;\; \cdots \;\; i \;\; ✠ \;} \to$$

또한 정점 v가 이 최종 큐에 나타나는 것은 i번째 단계에서 $dist(v)$가 변경된 경우이며, 오직 그 경우뿐입니다. 따라서 i번째 단계가 끝날 때 큐에는 $dist(v) = i$인 모든 정점 v가 들어 있습니다. □

보조정리 8.1에 따르면 BFS의 본체는 거리 레이블을 감소하지 않는 순서로 할당합니다. 한편 각 정점 v의 거리 레이블 $dist(v)$는 결코 증가하지 않습니다. 따라서 각

정점 v에 대해 "$dist(v) \leftarrow dist(u)+1$" 줄은 단계 $dist(v)$ 동안 *최대 한 번* 실행됩니다. 마찬가지로 다음이 성립합니다:

- 각 선행자 포인터 $pred(v)$는 단계 $dist(v)$ 동안에 최대 한 번만 변경됩니다.
- 각 정점 v는 큐에 최대 한 번, 단계 $dist(v)$ 동안 Push됩니다.
- 각 정점 u는 큐에서 최대 한 번 Pull되며, 이는 단계 $dist(u)+1$ 동안 일어납니다.
- 각 간선 $u \to v$에 대해 "$dist(v) > dist(u)+1$인가"라는 비교는 단계 $dist(u)+1$에서 최대 한 번만 수행됩니다.

종합하면, 이러한 관찰들로부터 너비 우선 탐색이 $\boldsymbol{O(V+E)}$ *시간*에 실행된다는 것을 알 수 있습니다. 직관적으로는 큐 안의 정점들을 시작 정점 s로부터 바깥쪽으로 단조롭게 퍼져 나가면서 그래프의 각 정점과 간선을 최대 한 번씩 지나가는 "파면(wavefront)"으로 생각할 수 있습니다. 이처럼 퍼져 나가는 파면의 비유는 이미 1961년에 Chin Yang Lee가 제안한 것으로, 그는 자신이 구현한 무어(Moore)의 알고리즘 A가 만들어 낸 시각화에서 영감을 얻었습니다.

이러한 관찰은 또한 조건 "if $dist(v) > dist(u)+1$"을 (어쩌면) 더 간단한 검사 "if $dist(v) = \infty$"로 바꿀 수 있음을 의미합니다. 그러면 거리는 다른 그래프 순회 알고리즘이 유지하는 표시와 같은 역할을 하며, 이 표시는 각 정점이 단 한 번만 방문되도록 보장합니다. 구체적으로 말하면, 정점은 거리 레이블이 유한할 때, 그리고 오직 그때에만 "표시된" 상태입니다.

그러나 최종 거리 레이블이 올바르다는 것은 아직 증명해야 합니다!

정리 8.2. *BFS가 끝나면, 모든 정점 v에 대해 $dist(v)$는 G에서 s부터 v까지의 최단 경로의 길이입니다.*

증명: 임의의 정점 v를 고정하고, G에서 $v_0 = s$이고 $v_\ell = v$인 임의의 경로 $v_0 \to v_1 \to \cdots \to v_\ell$을 생각해 봅시다. 저는 각 인덱스 j에 대해 $dist(v_j) \leq j$가 성립한다고 주장합니다. 특히 $dist(v) \leq \ell$입니다. 이 주장은 다음과 같이 j에 대한 귀납법으로 증명할 수 있습니다.

- 자명하게 $dist(v_0) = dist(s) = 0$입니다.
- 임의의 인덱스 $j > 0$에 대해, 귀납법 가정에 의해 $dist(v_{j-1}) \leq j-1$입니다. 큐에서 정점 v_{j-1}을 Pull한 직후에는 이미 $dist(v_j) \leq dist(v_{j-1})+1$이거나, 아니면 $dist(v_j) \leftarrow dist(v_{j-1})+1$로 설정합니다. 어느 경우든 $dist(v_j) \leq dist(v_{j-1})+1 \leq j$가 성립합니다.

방금 우리는 $dist(v)$가 s에서 v까지 가는 *임의의* 경로의 길이 이하임을 증명했습니다. 따라서 $dist(v)$는 s에서 v까지 가는 *최단* 경로의 길이 이하입니다.

비슷한 귀납법 증명을 통해 $dist(v)$가 선행자 경로 $s \to \cdots \to pred(pred(v)) \to pred(v) \to v$의 길이임을 알 수 있으므로, 이 경로는 최단 경로일 수밖에 없습니다. □

8.5 방향 비순환 그래프: 깊이 우선 탐색

방향 비순환 그래프에서도 최단 경로를 쉽게 계산할 수 있습니다. 간선에 가중치가 있는 경우에도, 특히 일부 간선의 가중치가 음수인 경우에도 마찬가지입니다. (음수

사이클은 걱정할 필요가 없습니다. 정의상 dag에는 사이클이 *전혀* 없기 때문입니다!) 실제로 이것은 완전히 표준적인 동적 계획법 알고리즘입니다.

G를 간선에 가중치가 있는 방향 그래프라 하고, s를 고정된 시작 정점이라 합시다. 임의의 정점 v에 대해 $dist(v)$가 G에서 s로부터 v까지의 최단 경로의 길이를 나타낸다고 합시다. 이 함수는 다음과 같은 간단한 점화식을 만족합니다.

$$dist(v) = \begin{cases} 0 & \text{만약} v = s \\ \min_{u \to v} \big(dist(u) + w(u \to v) \big) & \text{그렇지 않으면} \end{cases}$$

사실 이 항등식은 모든 방향 그래프에 대해 성립하지만, *점화식*이 되는 것은 방향 비순환 그래프에 대해서뿐입니다. 입력 그래프 G에 사이클이 있다면 이 함수를 재귀적으로 계산할 때 무한 루프에 빠질 것입니다. 그러나 G는 dag이므로, 각 재귀 호출은 위상 순서상 더 앞에 있는 정점을 방문합니다.

이 점화식의 의존성 그래프는 입력 그래프 G의 역방향 그래프입니다. 즉, 부분 문제 $dist(v)$가 $dist(u)$에 의존할 필요충분조건은 $u \to v$가 G의 간선이라는 것입니다. 따라서 G의 역방향 그래프에서 깊이 우선 탐색을 수행하면서 정점을 후위 순서로 고려하면 모든 정점의 거리를 $O(V+E)$ *시간*에 계산할 수 있습니다. 동등하게, 그림 8.7에 나타낸 것처럼 원래 그래프 G의 정점을 위상 순서로 고려할 수도 있습니다.

이렇게 얻은 동적 계획법 알고리즘은 포드의 일반적인 완화 알고리즘의 또 다른 예입니다! 이 연관성을 더 분명히 드러내기 위해, 그림 8.8에서 보듯이 $dist(v)$의 초기화를 메인 루프 밖으로 옮기고 선행자 포인터를 계산하는 부분을 추가할 수 있습니다. 그림 8.9는 이 알고리즘이 실제로 동작하는 모습을 보여 줍니다.

```
DagSSSP(s):
  for all vertices v in topological order
      if v = s
          dist(v) ← 0
      else
          dist(v) ← ∞
          for all edges u→v
              if dist(v) > dist(u) + w(u→v)      《u→v가 긴장 상태이면》
                  dist(v) ← dist(u) + w(u→v)     《u→v를 완화합니다》
```

그림 8.7. 동적 계획법을 이용한 DAG에서의 최단 경로 계산

```
DagSSSP(s):
  InitSSSP(s)
  for all vertices v in topological order
      for all edges u→v
          if u→v is tense
              Relax(u→v)
```

그림 8.8. 포드 알고리즘을 사용한 dag의 최단 경로 계산. (이 둘은 같은 알고리즘입니다.)

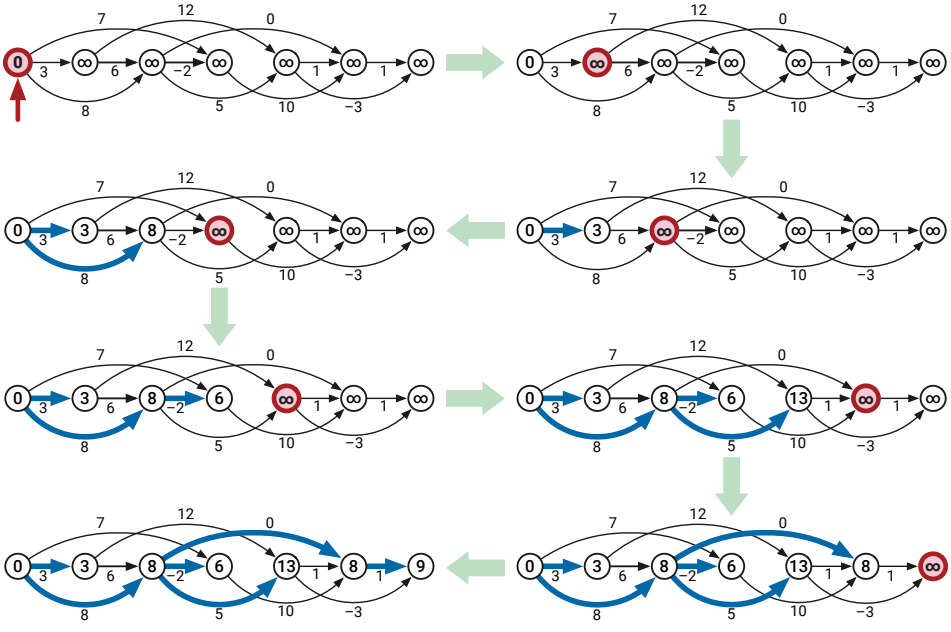


그림 8.9. 위상 순서에 따라 **들어오는** 간선을 완화하여 dag에서 최단 경로를 계산하는 과정입니다. 각 반복에서 굵은 간선은 선행 정점을 나타내며, 굵은 정점은 곧 탐색될 정점입니다. 그림 8.10과 비교해 보십시오.

DagSSSP는 너비 우선 탐색 및 포드의 완화 전략의 다른 사례들과 한 가지 사소한 점에서 다릅니다. 이러한 다른 최단 경로 알고리즘은 정점을 고려할 때마다 그 정점의 *나가는* 간선 각각을 완화하려고 시도하며, 직관적으로 말하면 파면을 출발점에서부터 앞으로 *밀어냅니다*. 반면 DagSSSP는 각 정점의 *들어오는* 간선 각각을 완화하려고 시도하며, 직관적으로 말하면 파면을 앞으로 *끌어당깁니다*.

하지만 들어오는 간선 대신 나가는 간선을 완화하도록 DagSSSP를 수정하면, dag에서 최단 경로를 $O(V + E)$ *시간*에 계산하면서도 우리가 다룬 다른 최단 경로 알고리즘들과 더 비슷한 또 다른 알고리즘을 얻게 됩니다.

```
PushDagSSSP(s):
  InitSSSP(s)
  for all vertices u in topological order
      for all outgoing edges u→v
          if u→v is tense
              Relax(u→v)
```

그림 8.10은 그림 8.9와 같은 그래프에서 이 수정된 알고리즘을 실행하는 과정을 보여 줍니다. PushDagSSSP의 정확성은 포드의 일반적인 완화 전략의 정확성으로부터 곧바로 따라 나오지만, 위상 순서에 따른 정점들에 대한 귀납법으로 정확성을 직접 증명하는 것도 어렵지 않습니다.

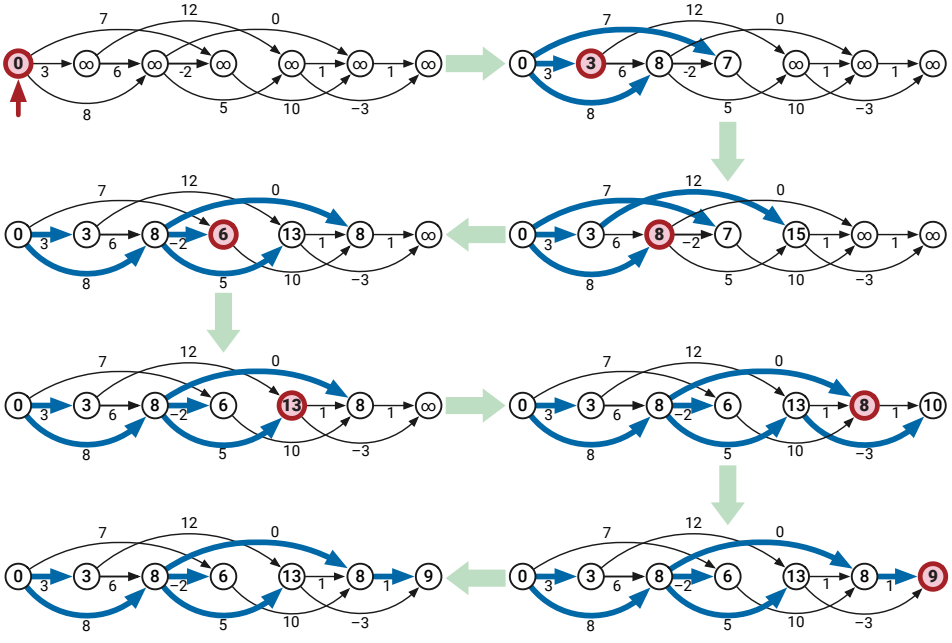


그림 8.10. 위상 순서에 따라 **나가는** 간선을 완화하여 dag에서 최단 경로를 계산하는 과정입니다. 각 반복에서 굵은 간선은 선행자를 나타내고, 굵은 정점은 곧 스캔될 정점입니다. 그림 8.9와 비교하십시오.

8.6 최선 우선 탐색: 데이크스트라 알고리즘

너비 우선 탐색의 FIFO 큐를 우선순위 큐로 바꾸고 정점 v의 키를 그 잠정 거리 $dist(v)$로 두면, 1957년에 마이클 레이조렉(Michael Leyzorek)이 이끈 케이스 공과대학(Case Institute of Technology) 연구팀이 미 육군 전자 시험장(US Army Electronic Proving Ground) 전투 개발부(Combat Development Department)를 위한 연간 프로젝트 보고서에서 처음 "발표"한 알고리즘을 얻습니다. 같은 알고리즘을 1956년에 에츠허르 데이크스트라(Edsger Dijkstra)가 독립적으로 발견했고(다만 출판은 1959년에야 이루어졌습니다), 1960년 이전 어느 시점에 조지 민티(George Minty)가 다시 발견했으며, 1960년에는 피터 휘팅(Peter Whiting)과 존 힐리어(John Hillier)가 또다시 발견했습니다. 1958년에는 조지 댄치그(George Dantzig)도 거의 동일한 알고리즘을 기술했습니다. 초기의 몇몇 문헌에서는 이를 "민티의 알고리즘"이라고 불렀지만, 오늘날 이 방법은 스티글러의 법칙(Stigler's Law)에 완벽히 부합하게도 "데이크스트라 알고리즘"으로 널리 알려져 있습니다.[7] 이 알고리즘의 의사코드는 그림 8.11에 나와 있습니다.

간단한 귀납법 증명으로 다음을 보일 수 있습니다. 이 알고리즘이 실행되는 동안 어느 시점에서든 간선 $u \to v$가 긴장(tense) 상태일 필요충분조건은 정점 u가 우선순위

[7] 역사적으로 부정확하기는 하지만 저는 이 일반적인 관례를 따르겠습니다. 한편으로는 "레이조렉-그레이-존슨-라듀-미커-페트리-세이츠-댄치그-데이크스트라-민티-휘팅-힐리어 알고리즘"에 관해 읽고 싶어 하는 사람은 아무도 없으리라 생각하기 때문이고, 다른 한편으로는 *실제로 출판되지* 않은 논문은 인정되지 않기 때문입니다.

```
DIJKSTRA(s):
  INITSSSP(s)
  INSERT(s, 0)
  while the priority queue is not empty
      u ← EXTRACTMIN( )
      for all edges u→v
          if u→v is tense
              RELAX(u→v)
              if v is in the priority queue
                  DECREASEKEY(v, dist(v))
              else
                  INSERT(v, dist(v))
```

그림 8.11. 데이크스트라 알고리즘.

큐에 있거나 우선순위 큐에서 가장 최근에 EXTRACT된 정점이라는 것입니다. 따라서 다익스트라 알고리즘은 포드의 일반 전략의 한 사례입니다. 그러므로 G에 음수 사이클이 없다면 이 알고리즘은 최단 경로를 올바르게 계산합니다.

음수 간선이 없는 경우

다익스트라 알고리즘은 입력 그래프에 음의 가중치를 갖는 간선이 없을 때 특히 잘 동작합니다. 이 경우 알고리즘은 직관적으로 시작 정점 s에서 바깥쪽으로 파면을 넓혀 가며, 너비 우선 탐색과 비슷하게 s로부터의 거리가 증가하는 순서대로 정점들을 지나갑니다. 그림 8.12는 이 알고리즘이 동작하는 모습을 보여 줍니다.

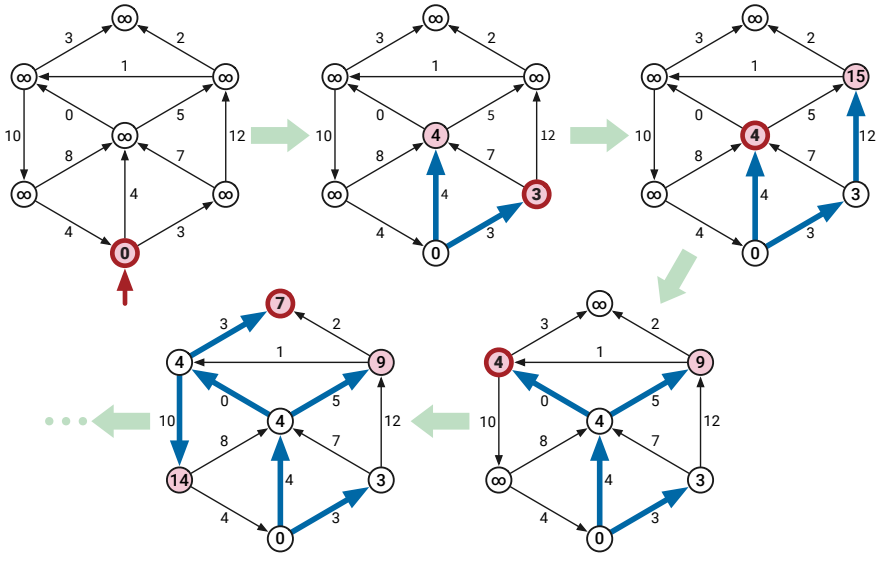


그림 8.12. 음수 간선이 없는 그래프에서 데이크스트라 알고리즘이 처음 네 번 반복하는 과정입니다. 각 반복에서 굵은 간선은 선행 정점을 나타내고, 음영 처리된 정점은 우선순위 큐에 들어 있으며, 굵게 표시된 정점은 곧 스캔될 정점입니다. 나머지 반복에서는 거리나 최단 경로 트리가 바뀌지 않습니다.

이 파면(wavefront) 직관을 형식화하면 이 설정에서 다익스트라 알고리즘의 정확성을 보이는 자기 완결적인 증명을 유도할 수 있습니다. 각 정수 i에 대해 u_i를 ExtractMin의 i번째 호출이 반환하는 정점이라 하고, d_i를 이 Extraction 직후의 $dist(u_i)$ 값이라 합시다. 특히 $u_1 = s$이고 $d_1 = 0$입니다. 이 시점에서는 정점 u_i들이 서로 다르다고 가정할 수 없습니다. 원칙적으로는 같은 정점이 두 번 이상 Extract될 수도 있습니다.

보조정리 8.3. *G에 음의 가중치를 갖는 간선이 없으면, 모든 $i < j$에 대해 $d_i \le d_j$가 성립합니다.*

증명: G에 음수 가중치 간선이 없다고 가정합니다. 임의의 인덱스 i를 고정합니다. 이 보조정리를 증명하려면 $d_{i+1} \ge d_i$임을 증명하는 것으로 충분합니다. 고려해야 할 경우는 두 가지입니다.

- G가 간선 $u_i \to u_{i+1}$을 포함하고 이 간선이 주 반복문의 i번째 반복 중에 완화된다면, 모든 간선 가중치가 음이 아니므로 i번째 반복이 끝날 때 $dist(u_{i+1}) = dist(u_i) + w(u_i \to u_{i+1}) \ge dist(u_i)$가 성립합니다.
- 그렇지 않은 경우, i번째 반복이 시작될 때 u_{i+1}은 이미 우선순위 큐에 들어 있어야 하며, u_i가 ExtractMin이 반환한 정점이므로 u_{i+1}의 우선순위는 $dist(u_{i+1}) \ge dist(u_i)$를 만족해야 합니다. 또한 $dist(u_{i+1})$은 i번째 반복 동안 변하지 않습니다.

두 경우 모두 $d_{i+1} \ge d_i$라는 결론을 얻습니다. 따라서 i에 대한 귀납법에 의해 보조정리가 바로 성립합니다. □

보조정리 8.4. *G에 음수 가중치 간선이 없으면, G의 각 정점은 우선순위 큐에서 최대 한 번만 Extract됩니다.*

증명: v가 두 번 이상 Extract된다고 가정합시다. 구체적으로, 어떤 인덱스 $i < j < k$에 대해 v가 주 루프의 i번째 반복에서 Extract되고, j번째 반복 중에 다시 Insert되며, k번째 반복 중에 다시 Extract된다고 가정합시다. 그러면 앞의 증명에서 사용한 표기법으로 $v = u_i = u_k$가 성립합니다.

거리 레이블 $dist(v)$는 절대 증가하지 않습니다. 또한 $dist(v)$는 j번째 반복 중 v가 다시 Insert되기 직전에 엄격하게 감소합니다. 따라서 $d_i > d_k$가 성립합니다. 그러므로 앞의 보조정리에 의해 G에는 음의 가중치를 갖는 간선이 적어도 하나 존재합니다. □

보조정리 8.4로부터 각 정점은 최대 한 번만 스캔되며, 따라서 각 간선도 최대 한 번만 완화된다는 사실이 곧바로 따라 나옵니다. 그러나 너비 우선 탐색과 달리 각 거리 레이블 $dist(v)$는 여러 번 바뀔 수 있습니다. $dist(v)$가 처음으로 ∞에서 다른 값으로 바뀔 때 v를 우선순위 큐에 Insert하고, 그 이후에는 $dist(v)$가 바뀔 때마다 DecreaseKey를 호출합니다. v가 우선순위 큐에서 Extract된 후에는 그 거리 레이블이 다시는 바뀌지 않습니다.

정확성 증명의 나머지 부분은 너비 우선 탐색의 경우와 거의 같습니다.

정리 8.5. *G에 음의 가중치를 갖는 간선이 없다면, Dijkstra가 종료될 때 모든 정점 v에 대해 $dist(v)$는 G에서 s로부터 v까지의 최단 경로의 길이입니다.*

증명: 임의의 정점 v를 고정하고, G에서 $v_0 = s$이고 $v_\ell = v$인 임의의 경로 $v_0 \to v_1 \to \cdots \to v_\ell$을 생각해 봅시다. 임의의 인덱스 j에 대해 L_j를 부분 경로

$v_0 \to v_1 \to \cdots \to v_j$의 길이라고 합시다. 모든 j에 대해 $dist(v_j) \le L_j$임을 귀납법으로 증명합니다.

- 자명하게 $dist(v_0) = dist(s) = 0 = L_0$입니다.
- 임의의 인덱스 $j > 0$에 대해, 귀납법 가정에 의해 $dist(v_{j-1}) \le L_{j-1}$이 성립합니다. 큐에서 정점 v_{j-1}을 Pull한 직후에는 이미 $dist(v_i) \le dist(v_{j-1}) + w(v_{j-1} \to v_j)$이거나, 아니면 $dist(v_i) \leftarrow dist(v_{j-1}) + w(v_{j-1} \to v_j)$로 설정합니다. 어느 경우든 다음이 성립합니다

$$dist(v_j) \;\le\; dist(v_{j-1}) + w(v_{j-1} \to v_j) \;\le\; L_{j-1} + w(v_{j-1} \to v_j) \;=\; L_j.$$

우리는 방금 $dist(v)$가 s에서 v로 가는 모든 경로의 길이 이하임을 증명했습니다. 따라서 $dist(v)$는 s에서 v로 가는 *최단* 경로의 길이 이하입니다.

한편, 비슷한 귀납법 증명을 통해 $dist(v)$가 선행자 경로 $s \to \cdots \to pred(pred(v)) \to pred(v) \to v$의 길이라는 것도 알 수 있습니다. □

이제 알고리즘의 실행 시간을 한정하는 일만 남았습니다. Dijkstra는 모두 합쳐 최대 E번의 DecreaseKey 연산과 최대 V번의 Insert 및 ExtractMin 연산을 수행합니다. 따라서 각 연산을 $O(\log V)$ 시간에 지원하는 표준 이진 힙으로 내부의 우선순위 큐를 구현하면, Dijkstra는 $\boldsymbol{O(E \log V)}$ *시간*에 실행됩니다.[8]

입력 그래프에 음수 간선이 *절대* 없다는 것을 미리 알고 있다면, 그림 8.13에 나타낸 것처럼 초기화 단계에서 모든 정점을 우선순위 큐에 Insert하고 주 루프에서는 DecreaseKey만 호출하는 방식으로 다익스트라 알고리즘을 약간 단순화할 수 있습니다. 이것은 대부분의 알고리즘 교과서와 위키백과, 심지어 다익스트라의 원 논문에서도 제시하는 버전의 다익스트라 알고리즘이며, 제가 5장에서 "최선 우선 탐색"이라고 설명한 버전의 다익스트라 알고리즘이기도 합니다.

♥음수 간선

하지만 NonnegativeDijkstra는 음수 간선이 있는 그래프에서 최단 경로를 올바르게 계산하지 *못합니다*. 게다가 모든 간선 가중치가 양수일 때에도 NonnegativeDijkstra는 Dijkstra보다 (이론적으로나 실제로나) 조금도 빠르지 않습니다. 이 두 가지 이유로, 저는 NonnegativeDijkstra보다 Dijkstra가 "데이크스트라 알고리즘"이라는 이름에 더 걸맞다고 생각합니다. 에츠허르 데이크스트라 본인조차도 항상 작동하지는 않는 빠른 알고리즘보다 가끔(그리고 실제로는 드물게) 느린 올바른 알고리즘이 더 낫다는 데 동의했을 것입니다!

안타깝게도 입력 그래프에 음수 간선이 있으면 익숙한 "퍼져 나가는 파면" 직관은 더 이상 정확하지 않습니다. 같은 정점이 여러 번 Extract될 수 있고, 같은 간선이

[8] 1950년대의 최단 경로 논문들은 우선순위 큐를 전혀 언급하지 않았습니다. 데이크스트라는 매 반복마다 파면 위의 모든 정점을 무차별적으로 훑는 방법을 제안했습니다. 그의 원래 알고리즘은 $\boldsymbol{O(V^2)}$ *시간*에 실행되며, 이는 $E = \Omega(V^2)$일 때 실제로 이진 힙 구현보다 빠릅니다! Minty는 $dist(u)$는 유한하지만 $dist(v)$는 유한하지 않은 모든 *간선* $u \to v$를 무차별적으로 훑는 방법을 제안했습니다. 따라서 그의 원래 알고리즘은 $O(VE)$ 시간에 실행됩니다. 거의 선형에 가까운 실행 시간을 얻기 위해 이진 힙으로 구현한 우선순위 큐를 사용하는 방법은 1977년 도널드 존슨이 제안했습니다. *피보나치 힙*이라는 더 복잡한 우선순위 큐 자료구조를 사용하면 실행 시간을 $\boldsymbol{O(E + V \log V)}$로 개선할 수 있습니다. 간선 가중치가 정수인 특수한 경우에는 훨씬 더 정교한 우선순위 큐를 사용하는 더욱 빠른 알고리즘도 있습니다.

```
NonnegativeDijkstra(s):
  InitSSSP(s)
  for all vertices v
      Insert(v, dist(v))
  while the priority queue is not empty
      u ← ExtractMin( )
      for all edges u→v
          if u→v is tense
              Relax(u→v)
              DecreaseKey(v, dist(v))
```

그림 8.13. 음수 간선이 없는 그래프를 위해 아주 약간 단순화한 데이크스트라 알고리즘입니다. Dijkstra와 다른 부분은 굵은 빨간색으로 표시했습니다.

여러 번 완화될 수 있으며, 거리가 증가하는 순서대로 발견되지 않을 수도 있습니다. 그림 8.15는 왼쪽 위 정점이 여섯 번 Extracted되고 위쪽의 세 간선이 각각 두 번씩 완화되는 실행 예를 보여 줍니다.

음수 사이클이 없다는 것 외에는 간선 가중치에 아무런 제약이 없는 그래프에서, Dijkstra의 최악의 경우 실행 시간은 사실 지수적입니다. 그림 8.14는 Dijkstra가 $\Theta(2^{V/2})$번의 완화를 수행하도록 강제하는 특히 단순한 그래프족(Douglas Shier와 Christoph Witzgall이 고안함)을 보여 줍니다.[9] 더 복잡한 그래프족(연습 문제로 남겨 두겠습니다)은 $\Theta(2^V)$번의 완화를 강제하며, 이는 가능한 최악의 경우입니다. 그러나 *실제*로는 음수 간선이 있는 그래프에서도 데이크스트라 알고리즘은 대개 빠릅니다.

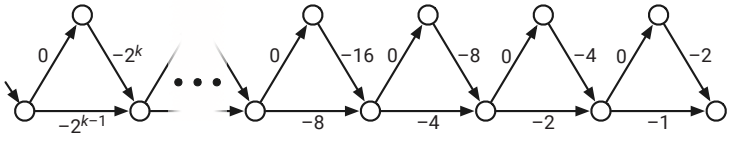


그림 8.14. 음수 간선이 있어서 Dijkstra가 지수 시간 동안 실행될 수밖에 없는 방향 그래프입니다.

8.7 모든 간선을 완화하기: 벨먼-포드

포드의 일반적인 최단 경로 알고리즘의 가장 단순한 구현은 1954년 Alfonso Shimbel이 처음 개략적으로 제시했고, 1957년 Edward Moore가 더 자세히 기술했으며, 1957년 Max Woodbury와 George Dantzig, 1958년 리처드 벨먼, 그리고 1958년 George Minty가 각각 독립적으로 재발견했습니다. (Woodbury와 Dantzig도, Minty도 자신들의 알고리즘을 발표하지는 않았습니다.) 스티글러의 법칙에 완벽하게 부합하게도, 이 알고리즘은 거의 보편적으로 ***벨먼-포드***라고 알려져 있습니다.[10] 벨먼이 간선을

[9] 재미있게도 Shier와 Witzgall의 예제는 간선이 $O(V)$개뿐인 dag입니다. 따라서 위쪽을 따라가는 지그재그 경로가 최단 경로 트리라는 사실을 미리 알아채지 *못했더라도* 최단 경로를 $O(V)$ 시간 만에 계산할 수 있습니다.

[10] 역사적으로 부정확하기는 하지만 저는 이 흔한 관례를 따르겠습니다. 한편으로는 "Shimbel / Moore / Woodbury-Dantzig / 벨먼-포드 / Kalaba / Minty 알고리즘"에 대해 정말로 읽고 싶어 하

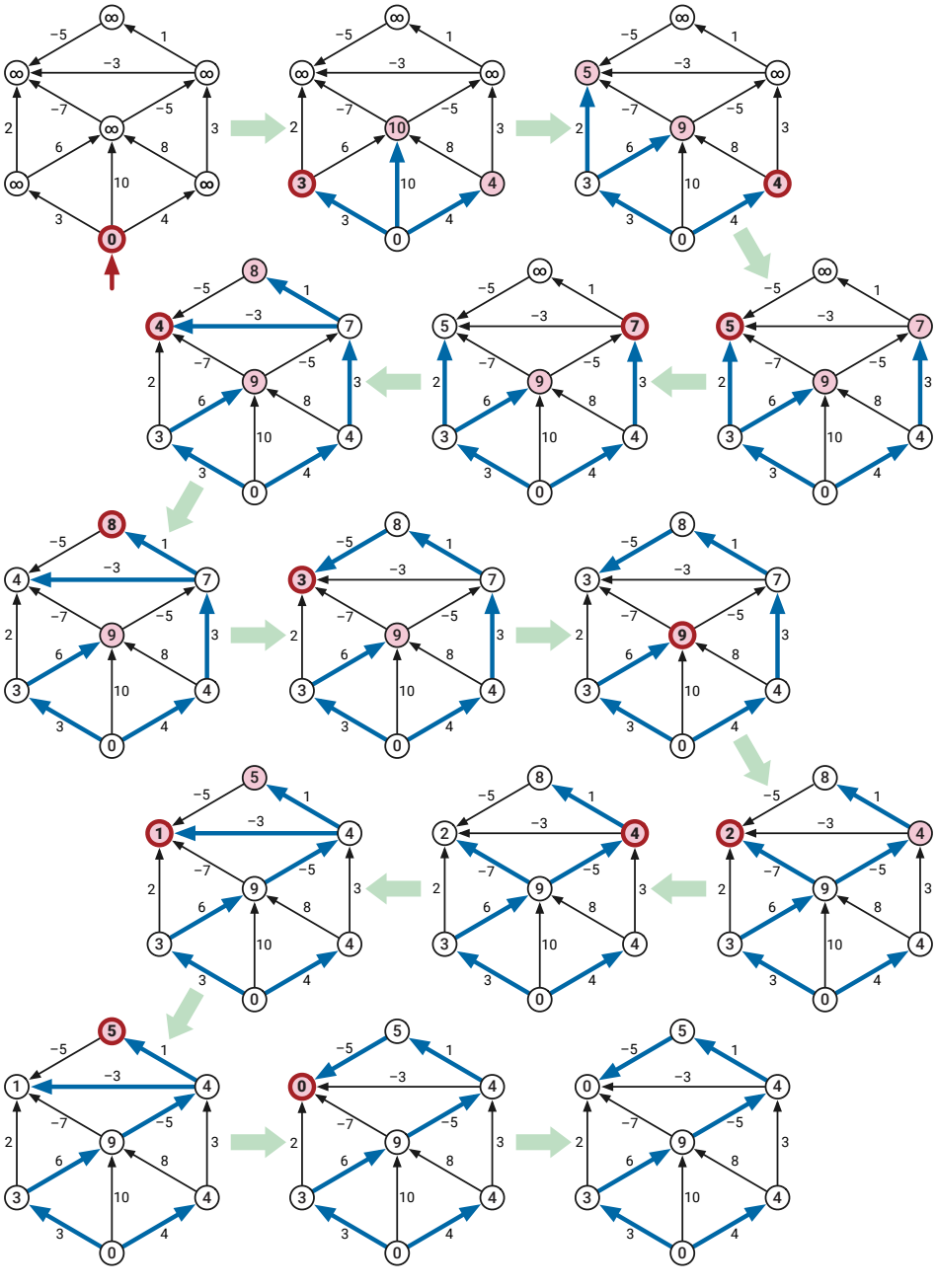


그림 8.15. 음수 간선이 있는 그래프에서 데이크스트라 알고리즘을 끝까지 실행한 모습입니다. 각 반복에서 굵은 간선은 선행자를 나타내고, 음영 처리된 정점은 우선순위 큐에 들어 있는 정점이며, 굵은 정점은 다음에 스캔할 정점입니다. 그림 8.17과 비교해 보십시오.

완화하는 포드의 1956년 정식화를 명시적으로 사용했기 때문입니다. 다만 일부 저자들은 "벨먼-Kalaba"라고 부르며[11] 몇몇 초기 문헌은 "벨먼-Shimbel"이라고 부릅니다.

Shimbel / Moore / Woodbury-Dantzig / 벨먼-포드 / Kalaba / Minty / Brosh[12] 알고리즘은 한 줄로 요약할 수 있습니다:

> BELLMAN-FORD: 긴장된 간선을 **모두** 완화한 다음 재귀합니다. (벨먼-포드)

```
BellmanFord(s)
  InitSSSP(s)
  while there is at least one tense edge
      for every edge u→v
          if u→v is tense
              Relax(u→v)
```

다음 보조정리는 벨먼-포드의 정확성과 효율성을 모두 증명하는 데 핵심입니다. 모든 정점 v와 음이 아닌 정수 i에 대해, $\boldsymbol{dist}_{\le i}(v)$가 G에서 s부터 v까지 *최대* i개의 간선으로 이루어진 최단 *보행*의 길이를 나타낸다고 합시다. 특히 $dist_{\le 0}(s) = 0$이고, $v \neq s$인 모든 v에 대해 $dist_{\le 0}(v) = \infty$입니다.

보조정리 8.6. *모든 정점 v와 음이 아닌 정수 i에 대해, BellmanFord의 주 반복문을 i번 반복한 후에는 $dist(v) \le dist_{\le i}(v)$가 성립합니다.*

증명: i에 대한 귀납법으로 증명합니다. 기저 사례 $i = 0$은 자명하므로 $i > 0$이라고 가정합니다. 정점 v를 하나 고정하고, W를 s에서 v까지 최대 i개의 간선으로 이루어진 최단 보행이라고 합시다(동률은 임의로 정합니다). 정의에 따라 W의 길이는 $dist_{\le i}(v)$입니다. 고려해야 할 경우는 두 가지입니다.

- W에 간선이 없다고 가정합시다. 그러면 W는 s에서 s로 가는 자명한 보행이어야 하므로 $v = s$이고 $dist_{\le i}(s) = 0$입니다. InitSSSP에서 $dist(s) \leftarrow 0$으로 설정하며 $dist(s)$는 결코 증가할 수 없으므로, 항상 $dist(s) \le 0$이 성립합니다.

- 그렇지 않다면 $u \to v$를 W의 마지막 간선이라고 합시다. 귀납법 가정에 의해 $i-1$번의 반복 후에는 $dist(u) \le dist_{\le i-1}(u)$가 성립합니다. 바깥쪽 루프의 i번째 반복 동안 안쪽 루프에서 간선 $u \to v$를 고려할 때, 이미 $dist(v) < dist(u) + w(u \to v)$이거나, 아니면 $dist(v) \leftarrow dist(u) + w(u \to v)$로 설정합니다. 두 경우 모두 $dist(v) \le dist_{\le i-1}(u) + w(u \to v) = dist_{\le i}(v)$가 성립합니다. 늘 그렇듯이 $dist(v)$는 증가할 수 없습니다(다만 바깥쪽 루프의 i번째 반복이 끝나기 전에 $dist(v)$가 더 감소할 수는 있습니다).

두 경우 모두, i번째 반복이 끝날 때 $dist(v) \le dist_{\le i}(v)$가 성립한다고 결론지을 수 있습니다. □

는 사람은 아무도 없으리라 생각하기 때문이고, 다른 한편으로는 제가 "Shimbel의 알고리즘"에 대해 이야기할 때마다 사람들이 저를 이상하게 쳐다보는 데 지쳤기 때문입니다.

[11] 이 이름은 아마도 리처드 벨먼과 Robert Kalaba가 1965년에 펴낸 동적 계획법과 제어 이론에 관한 단행본을 가리키는 것으로 보이며, 이 책에는 벨먼의 알고리즘이 기술되어 있습니다. 벨먼과 Kalaba는 1960년에 임의의 상수 k에 대해 k번째 최단 경로를 계산하는 벨먼 알고리즘의 확장도 발표했습니다.

[12] *Hyperbole and a Half*의 모든 글을 다시 읽으십시오. 그런 다음 고양이를 한 마리 더 입양해서 그 고양이에게도 책을 한 권 더 사 주십시오.

입력 그래프에 음수 사이클이 없다면, s에서 다른 임의의 정점으로 가는 최단 보행은 간선이 최대 $V-1$개인 단순 경로입니다. 따라서 벨먼-포드 알고리즘(BellmanFord)은 최대 $V-1$번의 반복 후에 올바른 최단 경로 거리를 가지고 종료합니다. 달리 말하면, $V-1$번의 반복 후에도 여전히 긴장된 간선이 있다면 입력 그래프에는 반드시 음수 사이클이 있습니다! 그러므로 알고리즘을 다음과 같이 더 구체적으로 다시 쓸 수 있습니다:

```
BellmanFord(s)
  InitSSSP(s)
  repeat V − 1 times
      for every edge u→v
          if u→v is tense
              Relax(u→v)
  for every edge u→v
      if u→v is tense
          return "Negative cycle!"
```

안쪽 루프의 각 반복에는 자명하게 $O(E)$ 시간이 걸리므로, 전체 알고리즘은 ***$O(VE)$ 시간***에 실행됩니다. 따라서 벨먼-포드 알고리즘은 그래프에 음수 간선이 있더라도, 더 나아가 그래프에 음수 *사이클*이 있더라도 *항상* 효율적입니다.

그러나 모든 간선 가중치가 음수가 아니라면, 적어도 최악의 경우에는 데이크스트라 알고리즘이 더 빠릅니다. (실제로는 음수 간선이 있는 그래프에서도 데이크스트라 알고리즘이 벨먼-포드보다 더 빠른 경우가 많습니다.)

Moore의 개선

Moore와 벨먼 중 누구도 벨먼-포드 알고리즘을 제가 여기서 제시한 형태로 기술하지는 않았습니다. Moore는 가중치 없는 그래프에 대한 너비 우선 탐색("Algorithm A")을 제안한 바로 그 논문에서 이 알고리즘의 자신만의 버전("Algorithm D")을 제시하였으며, 실제로 두 알고리즘은 거의 동일합니다. Moore의 알고리즘은 BellmanFord와 같은 $O(VE)$ 최악의 경우 실행 시간을 가지지만, 실제로는 상당히 더 빠른 경우가 많습니다. 직관적으로 말하면, 이는 "명백히" 긴장되지 않은 간선을 검사하지 않기 때문입니다.

Moore는 너비 우선 탐색에 두 가지 수정을 가하여 자신의 가중치 최단 경로 알고리즘을 유도했습니다. 첫째, 간선 가중치를 고려하기 위해 가장 안쪽 루프의 각 "+1"을 "$+w(u\to v)$"로 바꿉니다. 둘째, 정점을 Insert하기 전에 그 정점이 이미 FIFO 큐에 있는지 확인하여, 큐가 항상 각 정점의 사본을 최대 하나만 포함하도록 합니다.[13]

앞서 했던 너비 우선 탐색 분석에서처럼, 알고리즘의 실행을 여러 단계로 나누기 위해 ✠라는 "토큰"을 도입하겠습니다. 너비 우선 탐색과 마찬가지로, 각 단계는 토큰이 큐에 Push될 때 시작되고, 토큰이 다시 큐에서 Pull될 때 끝납니다. BFS와 마찬가지로, 큐에 토큰*만* 남아 있으면 알고리즘이 종료됩니다. 이렇게 만든 알고리즘은 그림 8.16에 나와 있습니다.

[13] 이 확인이 없어도 Moore의 알고리즘은 여전히 *올바르지만*, $O(VE)$ 시간 한계는 성립하지 않습니다.

```
MOORE(s):
  INITSSSP(s)
  PUSH(s)
  PUSH(✠)                《첫 번째 단계를 시작합니다》
  while the queue contains at least one vertex
      u ← PULL( )
      if u = ✠
          PUSH(✠)        《다음 단계를 시작하십시오》
      else
          for all edges u→v
              if u→v is tense
                  RELAX(u→v)
                  if v is not already in the queue
                      PUSH(v)
```

그림 8.16. Moore의 최단 경로 알고리즘입니다. 토큰 ✠가 포함된 굵은 빨간 줄은 분석만을 위한 것입니다.

큐에는 어느 시점에서든 각 정점의 사본이 최대 하나만 들어 있으므로, 각 정점은 각 단계에서 큐로부터 최대 한 번만 PULL되며, 따라서 각 간선 $u \to v$의 긴장 여부도 각 단계에서 최대 한 번만 검사됩니다. 게다가 단계가 시작될 때 긴장된 모든 간선은 그 단계 동안 완화됩니다. (단계 도중에 긴장된 간선 중 일부도 그 단계 동안 완화될 수 있으며, 완화된 간선 중 일부가 같은 단계에서 다시 긴장될 수도 있습니다.) 따라서 MOORE는 모든 간선을 무차별 대입으로 검사하는 대신 큐를 사용하여 긴장된 간선을 관리하는 BELLMANFORD의 개선판으로 볼 수 있습니다. 특히, 비슷한 귀납적 증명으로 보조정리 8.6에 대응하는 다음 결과를 증명할 수 있습니다.

보조정리 8.7. *모든 정점 v와 음이 아닌 정수 i에 대해, MOORE의 단계를 i번 수행한 후에는 $dist(v) \le dist_{\le i}(v)$가 성립합니다.*

따라서 입력 그래프에 음수 사이클이 없으면 MOORE는 최대 $V-1$개의 단계를 거친 후 정지합니다. 각 단계에서 각 정점을 최대 한 번 검사하므로 각 간선을 최대 한 번 완화하며, 따라서 한 단계의 최악의 경우 실행 시간은 $O(E)$입니다. 따라서 MOORE의 전체 실행 시간은 $\boldsymbol{O(VE)}$입니다. 그러나 실제로는 MOORE가 BELLMANFORD보다 최단 경로를 상당히 더 빠르게 계산하는 경우가 많습니다. 이는 MOORE가 이전 단계에서 $dist(u)$가 변경된 경우에만 간선 $u \to v$를 검사하기 때문입니다.

입력 그래프에 음수 사이클이 있으면 MOORE는 절대 멈추지 않습니다. 다행히 BELLMANFORD와 마찬가지로, 음수 사이클이 존재할 때 이를 보고하도록 MOORE의 알고리즘을 수정하기는 쉽습니다. 아마 가장 쉬운 수정 방법은 토큰을 *실제로* 유지하면서 그 토큰이 큐에서 PULL된 횟수를 세는 것입니다. 그러면 토큰이 $(V-1)$번째로 PULL된 직후에 큐가 비어 있지 않은 경우에만, 그리고 그 경우에는 항상 입력 그래프에 음수 사이클이 있습니다.

동적 계획법 정식화

리처드 벨먼은 자신의 이름이 붙은 거의 모든 것과 마찬가지로 "벨먼-포드" 최단 경로 알고리즘도 동적 계획법으로 유도했습니다. 늘 그렇듯이, 최단 경로 거리의 재귀적

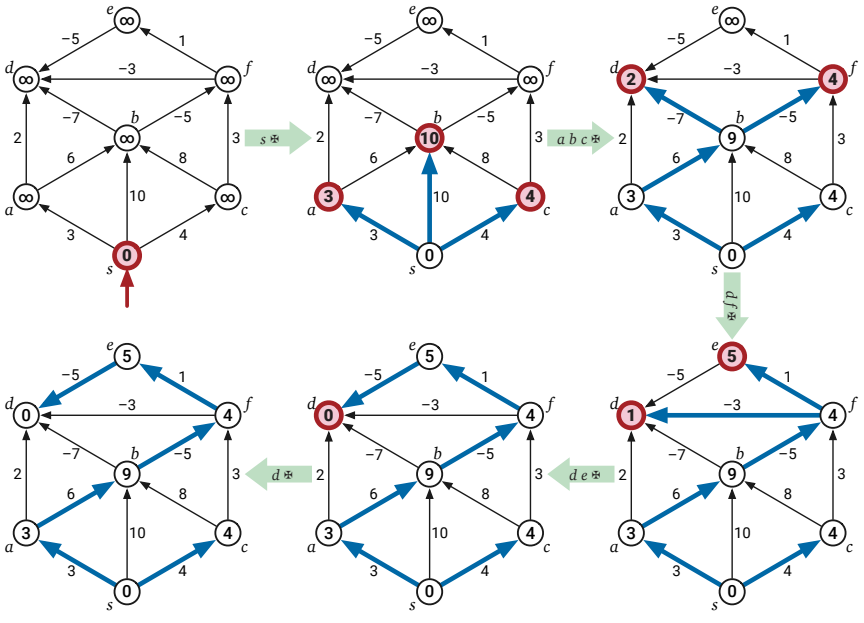


그림 8.17. 음수 간선이 있는 방향 그래프에서 Moore 알고리즘을 처음부터 끝까지 실행한 모습입니다. 노드는 s ✠ a b c ✠ d f ✠ d e ✠ d ✠✠ 순서로 큐에서 꺼내지며, 여기서 ✠는 단계 종료 토큰입니다. 각 단계가 시작될 때 굵은 간선은 선행자를 나타내고, 음영 처리된 정점은 정점 큐에 들어 있는 정점입니다. 그림 8.6 및 그림 8.15와 비교해 보십시오.

정의에서 시작해야 합니다. 방향 비순환 그래프에서 활용했던 것과 같은 항등식을 사용하고 싶어집니다:

$$dist(v) = \begin{cases} 0 & \text{만약} v = s \\ \min\limits_{u \to v}\big(dist(u) + w(u \to v)\big) & \text{그렇지 않으면} \end{cases}$$

안타깝게도 입력 그래프가 dag가 아니면 이 점화식은 작동하지 않습니다! 입력 그래프에 방향 사이클 $u \to v \to w \to u$가 있다고 가정합시다. $dist(w)$를 계산하려면 먼저 $dist(v)$가 필요하고, $dist(v)$를 계산하려면 먼저 $dist(u)$가 필요한데, $dist(u)$를 계산하려면 먼저 $dist(w)$가 필요합니다. 입력 그래프에 방향 사이클이 하나라도 있으면 무한 루프에 빠지고 맙니다!

적절한 점화식을 세우려면 거리 함수에 구조적 매개변수를 하나 더 추가해야 합니다. 이 매개변수는 재귀 호출마다 단조 감소하며, 매개변수가 0에 도달하면 함수를 손쉽게 평가할 수 있도록 정의됩니다. 벨먼은 이 추가 매개변수로 ***간선의 최대 개수***를 선택했습니다.[14]

앞선 분석에서와 같이, $dist_{\le i}(v)$를 최대 i개의 간선으로 이루어진 s에서 v로의 최단 보행의 길이라고 합시다. 벨먼은 이 함수가 다음 벨먼 방정식 점화식을 따른다는 것을

[14] 다음 장에서 보겠지만, 이것이 유일하게 합리적인 선택은 아닙니다.

관찰했습니다:

$$dist_{\le i}(v) = \begin{cases} 0 & \text{만약} i = 0 \text{그리고} v = s \\ \infty & \text{만약} i = 0 \text{그리고} v \neq s \\ \min \left\{ \begin{array}{c} dist_{\le i-1}(v) \\ \min\limits_{u \to v} \left(dist_{\le i-1}(u) + w(u \to v) \right) \end{array} \right\} & \text{그렇지 않으면} \end{cases}$$

그래프에 음수 사이클이 없다고 가정합시다. 그러면 우리의 목표는 모든 정점 v에 대해 $dist_{\le V-1}(v)$를 계산하는 것입니다. 다음은 이 점화식을 동적 계획법으로 직접 평가하는 방법이며, 여기서 $dist[i, v]$는 $dist_{\le i}(v)$의 값을 저장합니다. 최종 최단 경로 거리의 정확성은 점화식의 정확성으로부터 도출되며, $O(VE)$ 실행 시간은 자명합니다. 이것이 본질적으로 벨먼이 자신의 최단 경로 알고리즘을 제시한 방식입니다.

```
BellmanFordDP(s)
  dist[0, s] ← 0
  for every vertex v ≠ s
      dist[0, v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i, v] ← dist[i − 1, v]
          for every edge u→v
              if dist[i, v] > dist[i − 1, u] + w(u→v)
                  dist[i, v] ← dist[i − 1, u] + w(u→v)
```

몇 가지 사소한 최적화를 차례로 적용하면 이 동적 계획법 알고리즘을 BellmanFord의 원래 형태로 바꿀 수 있습니다. 먼저, 가장 바깥쪽 루프의 각 반복은 각 간선 $u \to v$를 정확히 한 번씩 고려하지만, 그 간선들을 고려하는 순서는 사실 중요하지 않습니다. 따라서 마지막 세 줄의 들여쓰기를 한 단계 없애도 안전합니다! 수정된 알고리즘은 간선을 다른 *순서*로 고려할 수도 있지만, 여전히 모든 i와 v에 대해 $dist_{\le i}(v)$를 올바르게 계산합니다.

```
BellmanFordDP2(s)
  dist[0, s] ← 0
  for every vertex v ≠ s
      dist[0, v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i, v] ← dist[i − 1, v]
      for every edge u→v
          if dist[i, v] > dist[i − 1, u] + w(u→v)
              dist[i, v] ← dist[i − 1, u] + w(u→v)
```

다음으로 마지막 두 줄의 인덱스를 $i - 1$에서 i로 바꿉니다. 이렇게 바꾸면 거리 $dist[i, v]$가 이전보다 더 빨리 실제 최단 경로 거리에 가까워질 수 있지만, 알고리즘은 여전히 실제 최단 경로 거리를 올바르게 계산합니다. 이제 모든 i와 v에 대해

$dist[i,v] = dist_{\le i}(v)$ 대신 $dist[i,v] \le dist_{\le i}(v)$가 성립하며, 이는 보조정리 8.6 및 8.7과 같은 방식입니다.

```
BellmanFordDP3(s)
  dist[0, s] ← 0
  for every vertex v ≠ s
      dist[0, v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i, v] ← dist[i − 1, v]
      for every edge u→v
          if dist[i, v] > dist[i, u] + w(u→v)        《i − 1이 아닙니다!》
              dist[i, v] ← dist[i, u] + w(u→v)       《i − 1이 아닙니다!》
```

하지만 이 알고리즘은 조금 어리석습니다. 가장 바깥쪽 루프의 i번째 반복에서는 먼저 배열 $dist[\cdot,\cdot]$의 $(i-1)$번째 행을 i번째 행으로 복사한 다음, i번째 행의 원소들을 수정합니다. 그러니 사실 2차원 배열은 전혀 필요하지 않으며, 반복 인덱스 i는 완전히 불필요합니다! 마지막으로 수정한 버전에서는 잠정 거리를 담은 1차원 배열 하나만 유지합니다.

```
BellmanFordFinal(s)
  dist[s] ← 0
  for every vertex v ≠ s
      dist[v] ← ∞
  for i ← 1 to V − 1
      for every edge u→v
          if dist[v] > dist[u] + w(u→v)
              dist[v] ← dist[u] + w(u→v)
```

이 마지막 동적 계획법 알고리즘은 BellmanFord의 원래 정식화와 거의 동일합니다! 처음 세 줄은 최단 경로 거리를 초기화하고, 마지막 두 줄은 간선 $u\to v$가 긴장되어 있으면 그 간선을 완화합니다. BellmanFordFinal에는 앞선 정식화의 기능 중 두 가지만 빠져 있습니다. 선행자 포인터를 유지하지 않으며 음수 사이클을 감지하지도 않습니다. 다행히 이 기능들을 추가하는 것은 간단합니다.

연습문제

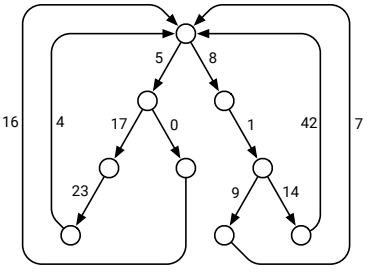
0. G를 간선 가중치가 임의의 값(양수, 음수 또는 0일 수 있습니다)이고 음수 사이클이 있을 수도 있는 방향 그래프라고 하고, s를 G의 임의의 정점이라고 합시다.

   (a) 모든 정점 v에 수 $dist(v)$가 저장되어 있다고 가정합시다(단, 선행자 포인터는 저장되어 있지 않습니다). 모든 정점 v에 대해 $dist(v)$가 s에서 v까지의 최단 경로 거리인지 판별하는 알고리즘을 서술하고 분석하십시오.

   (b) 이번에는 대신 $v \neq s$인 모든 정점 v가 G의 다른 정점을 가리키는 포인터 $pred(v)$를 저장한다고 가정합시다(단, 거리는 저장하지 않습니다). 이 선행

자 포인터들이 s를 루트로 하는 단일 출발점 최단 경로 트리를 정의하는지 판별하는 알고리즘을 설명하고 분석하십시오.

1. *루프 트리*란 이진 트리에 모든 리프에서 루트로 되돌아가는 간선을 추가하여 만든, 가중치가 있는 방향 그래프입니다. 모든 간선의 가중치는 음이 아닙니다.



루프가 있는 트리.

   (a) n개의 노드로 이루어진 루프 트리에서 두 정점 u와 v 사이의 최단 경로를 계산하는 데 다익스트라 알고리즘은 얼마나 많은 시간이 필요합니까?

   (b) 더 빠른 알고리즘을 설명하고 분석하십시오.

2. 가중치가 있는 간선을 가진 방향 그래프 G와 두 정점 s, t가 주어졌다고 가정합시다.

   (a) G에서 정확히 하나의 간선만 음의 가중치를 가질 때, s에서 t까지의 최단 경로를 찾는 알고리즘을 설명하고 분석하십시오. *[힌트: 다익스트라 알고리즘을 수정하십시오. 아니면 수정하지 마십시오.]*

   (b) G에서 정확히 k개의 간선이 음의 가중치를 가질 때, s에서 t까지의 최단 경로를 찾는 알고리즘을 설명하고 분석하십시오. 알고리즘의 실행 시간은 k에 어떻게 의존합니까?

3. 모든 *정점*이 양의 가중치를 갖는 무방향 그래프 G가 주어졌다고 가정합시다.

   (a) 총 가중치가 최소인 G의 *신장 트리*를 찾는 알고리즘을 설명하고 분석하십시오. (신장 트리의 총 가중치는 그 정점들의 가중치의 합입니다.)

   (b) G에서 주어진 한 정점 s부터 또 다른 주어진 정점 t까지 가는 *경로* 중 총 가중치가 최소인 경로를 찾는 알고리즘을 서술하고 분석하십시오. (경로의 총 가중치는 그 경로에 속한 정점들의 가중치의 합입니다.)

   [힌트: 이 문제들 중 하나는 자명합니다.]

4. 임의의 그래프 G의 임의의 간선 e에 대해, $G \setminus e$는 G에서 e를 삭제하여 얻은 그래프를 나타낸다고 합시다. 그래프 G와 두 정점 s, t가 주어졌다고 가정합시다. *대체 경로* 문제는 G의 모든 간선 e에 대해 $G \setminus e$에서 s로부터 t까지의 최단 경로

거리를 계산할 것을 요구합니다. 출력은 G의 각 간선마다 하나씩, 총 E개의 거리로 이루어진 배열입니다.

(a) G가 *방향* 그래프이고, 정점 s에서 정점 t까지의 최단 경로가 G의 모든 정점을 지난다고 가정합시다. 대체 경로 문제의 이 특수한 경우를 $O(E \log V)$ 시간에 해결하는 알고리즘을 설명하십시오.

♥(b) 임의의 *무방향* 그래프에 대해 대체 경로 문제를 $O(E \log V)$ 시간에 해결하는 알고리즘을 설명하십시오.

두 부분 문제 모두에서 모든 간선 가중치가 음이 아니라고 가정해도 됩니다. *[힌트: 원래 최단 경로의 간선 하나를 삭제하면, 기존 최단 경로와 새 최단 경로는 어떻게 겹칩니까?]*

5. $G = (V, E)$를 음수 간선 가중치가 없는 연결 방향 그래프라 하고, s와 t를 G의 정점이라 하며, H를 G에서 일부 간선을 삭제하여 얻은 부분 그래프라 합시다. 결과 그래프에서 s에서 t까지의 최단 경로가 가능한 한 짧아지도록, G의 간선 중 정확히 하나를 H에 다시 삽입하려 한다고 가정합시다. 다시 삽입할 최선의 간선을 $O(E \log V)$ 시간에 선택하는 알고리즘을 설명하고 분석하십시오.

6. (a) s에서 도달 가능한 음수 사이클이 있으면 그러한 사이클을 실제로 반환하고, 그러한 사이클이 없으면 최단 경로 트리를 반환하도록 벨먼-포드를 수정한 알고리즘을 기술하고 분석하십시오. 수정된 알고리즘은 여전히 $O(VE)$ 시간에 실행되어야 합니다.

(b) 입력 그래프에 음수 사이클이 있더라도 s에서 다른 모든 정점까지의 올바른 최단 경로 거리를 계산하도록 벨먼-포드를 수정한 알고리즘을 설명하고 분석하십시오. s에서 v로 가는 보행 중 음수 사이클을 포함하는 것이 하나라도 있으면 알고리즘은 $dist(v) = -\infty$인 상태로 끝나야 합니다. 그렇지 않으면 $dist(v)$에는 s에서 v까지의 최단 경로 길이가 담겨 있어야 합니다. 수정한 알고리즘의 실행 시간은 여전히 $O(VE)$여야 합니다.

♥(c) 포드(Ford)의 일반 완화 알고리즘에 대해 (a)와 (b)를 다시 푸십시오. 음수 사이클이 없으면 수정하지 않은 알고리즘이 $O(2^V)$ 단계 안에 종료한다고 가정해도 됩니다. 수정한 알고리즘도 $O(2^V)$ 시간에 실행되어야 합니다.

7. 포드의 일반적인 완화 알고리즘을 훨씬 더 느슨하게 변형한 다음 알고리즘을 생각해 보십시오.

```
FellmanBored(s):
  InitSSSP(s)
  for i ← 1 to whatever, man, I don't care
      e_i ← any edge in G
      if e_i is tense
          Relax(e_i)
```

FellmanBored가 s에서 시작하는 임의의 보행 W의 간선들을 W를 따라 순서대로 검사한다면, W의 마지막 거리 레이블은 W의 길이 이하임을 증명하십시오. 더 형식적으로 말하면 다음과 같습니다. $v_0 = s$인 임의의 보행 $v_0 \to v_1 \to \cdots \to v_\ell$의 간선들이 FellmanBored가 검사하는 간선들의 *부분 수열* $e_1, e_2, e_3, \ldots$을 이룬다면, $dist(v_\ell) \le \sum_{i=1}^{\ell} w(v_{i-1} \to v_i)$가 성립합니다. *[힌트: 이 성질은 올바르게 서술하는 것보다 증명하는 것이 오히려 더 쉽다고 할 정도입니다.]*

8. 이 문제에서는 포드의 일반적인 완화 알고리즘을 사용하여 음수 사이클을 검출하는 여러 가지 방법을 살펴봅니다.

   (a) InitSSSP 이후에 $pred(s)$가 한 번이라도 바뀐다면 입력 그래프에 s를 지나는 음수 사이클이 존재함을 증명하십시오.

   (b) 입력 그래프에 s를 지나는 음수 사이클이 있더라도 InitSSSP 이후 $pred(s)$가 한 번도 바뀌지 않을 수 있음을 보이십시오.

   (c) P를 선행자 간선 $pred(v) \to v$로 이루어진 현재 그래프라고 하고, X를 현재 *긴장*된 모든 간선의 집합이라고 합시다. 두 집합은 모두 알고리즘이 실행되는 동안 변합니다. 입력 그래프에 음수 사이클이 없을 필요충분조건은 $P \cup X$가 항상 dag인 것임을 증명하십시오.

   (d) 지금까지 완화된 모든 간선의 집합을 R이라고 합시다. 이 집합은 알고리즘이 실행됨에 따라 커집니다. 입력 그래프에 음수 사이클이 없을 필요충분조건은 R이 항상 dag인 것임을 증명하십시오.

♥9. 간선에 음수 가중치가 허용되면, 기저 그래프가 비순환이더라도 데이크스트라 알고리즘이 최악의 경우 $\Omega(2^V)$번의 완화를 수행함을 증명하십시오. 구체적으로, 모든 양의 정수 n에 대해 G_n이 입력 그래프일 때 데이크스트라 알고리즘이 Relax를 $\Omega(2^n)$번 호출하도록, 간선에 가중치가 있는 n-정점 dag G_n을 구성하십시오. *[힌트: 이진 카운터.]*

♥10. 입력 그래프에 음수 사이클이 없다면, 포드의 일반적인 완화 알고리즘(따라서 데이크스트라 알고리즘도)이 최대 $O(2^V)$번의 완화 후에 멈춘다는 것을 증명하십시오. *[힌트: 문제 8(d)를 참고하십시오.]*

11. ***모든 간선의 가중치가 음수인*** 방향 그래프 G와 시작 정점 s가 주어졌다고 가정합시다. s에서 G의 다른 모든 정점까지의 최단 경로 거리를 계산하는 효율적인 알고리즘을 설명하고 분석하십시오. 구체적으로는 모든 정점 t에 대해 다음과 같습니다:

   - s에서 t에 도달할 수 없다면, 알고리즘은 $dist(t) = \infty$를 보고해야 합니다.
   - G에 s에서 도달할 수 있는 사이클이 있고 그 사이클에서 t에 도달할 수 있다면, s에서 t까지의 최단 경로 거리는 잘 정의되지 않습니다. s에서 t까지 길이가 음수이면서 그 절댓값이 임의로 큰 경로(엄밀히 말하면 보행)가 존재하기 때문입니다. 이 경우 여러분의 알고리즘은 $dist(t) = -\infty$를 보고해야 합니다.

- 앞의 두 조건 중 어느 것에도 해당하지 않으면, 여러분의 알고리즘은 s에서 t까지의 올바른 최단 경로 거리를 보고해야 합니다.

12. 우리는 보통 두 노드 사이의 "그" 최단 경로라고 말하지만, 하나의 그래프에는 끝점이 같은 최소 길이 경로가 여러 개 있을 수 있습니다. 가중 그래프에서도 최소 가중치 경로 가운데 간선 수가 가장 적은 경로를 고르는 것이 바람직한 경우가 많습니다. 이러한 경로를 s에서 t로 가는 ***최선 경로***라고 부릅시다. 간선 가중치가 양수인 방향 그래프 G와 G의 시작 정점 s가 주어졌다고 가정합시다. G에서 s로부터 다른 모든 정점까지의 *최선* 경로를 계산하는 알고리즘을 설명하고 분석하십시오.

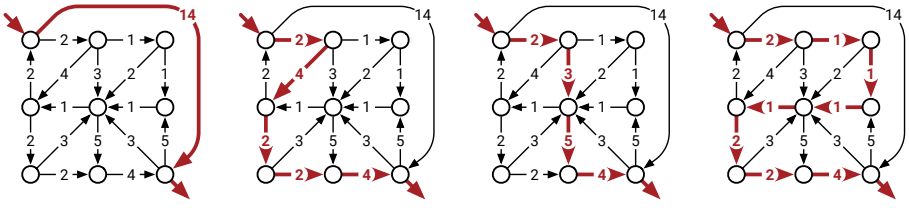


그림 8.18. 길이가 같은 (수많은) 최단 경로 중 네 개입니다. 첫 번째 경로가 "가장 좋은" 최단 경로입니다.

13. 간선에 가중치가 있는 임의의 방향 그래프 G에서 출발 정점 s부터 대상 정점 t까지의 최단 경로의 *개수*를 구하는 알고리즘을 설명하고 분석하십시오. 모든 간선 가중치는 양수이며, 필요한 모든 산술 연산은 $O(1)$ 시간에 수행할 수 있다고 가정해도 됩니다. *[힌트: s부터 다른 모든 정점까지의 최단 경로 거리를 계산하십시오. s부터 다른 정점까지의 최단 경로에 포함될 수 없는 간선을 모두 버리십시오. 무엇이 남습니까?]*

14. 방금 Twitbook에서 초등학교 시절 가장 친한 친구를 찾았습니다. 두 사람 모두 되도록 빨리 만나고 싶지만, 서로 멀리 떨어진 두 도시에 살고 있습니다. 이동 시간을 최소화하기 위해 두 사람은 중간에 있는 한 도시에서 만나기로 하고, 동시에 각자 차에 올라 서로를 향해 운전하기 시작합니다. 그런데 *정확히* 어디에서 만나야 하겠습니까?

    가중 그래프 $G = (V, E)$가 주어집니다. 여기서 정점 V는 도시를 나타내고, 간선 E는 도시를 직접 잇는 도로를 나타냅니다. 각 간선 e에는 두 도시 사이를 이동하는 데 걸리는 시간과 같은 가중치 $w(e)$가 있습니다. 또한 여러분의 출발 위치를 나타내는 정점 p와 친구의 출발 위치를 나타내는 정점 q도 주어집니다.

    여러분과 친구가 가능한 한 빨리 만날 수 있게 해 주는 대상 정점 t를 찾는 알고리즘을 설명하고 분석하십시오.

15. 여러분은 미국 국가 고속도로망 전체를 따라 거리 수준의 이미지를 제공할 Giggle Highway View 프로젝트의 자전거 라이더로 고용되었습니다. 시범 프로젝트로서, 여러분은 Giggle Highway-View Fixed-Gear Carbon-Fiber Bicycle을 타

고 오리건주 포틀랜드의 "the Giggleplex"에서 뉴욕 브루클린 윌리엄스버그의 "Gigglesburg"까지 달려 달라는 요청을 받았습니다.

당신은 구제 불능의 카페인 중독자이지만, 대부분의 Giggle 직원과 마찬가지로 커피에 까다로운 사람이기도 합니다. 당신은 독립 로스터리에서 로스팅하고, 손으로 추출하고, 직거래로 들여오고, 유기농으로 그늘에서 재배한 싱글 오리진 에스프레소만 마시며, 우유나 설탕은 절대 섞지 않습니다. 이 점, *대단히* 감사하게 생각합니다. 에스프레소 한 잔을 마실 때마다 카페인 금단 편두통이 오기 전까지 최대 L마일을 자전거로 달릴 수 있습니다.

Giggle은 친절하게도 무방향 그래프 G 형태의 미국 지도를 제공합니다. 이 그래프의 정점은 독립 로스터리에서 로스팅하고 수동으로 추출한 직거래 유기농 그늘 재배 싱글 오리진 에스프레소를 판매하는 커피숍을 나타내고, 간선은 커피숍 사이를 잇는 고속도로를 나타냅니다. 각 간선 e에는 해당 고속도로 구간의 길이 $\ell(e)$가 표시되어 있습니다. 당연히 Giggle의 두 사무실에는 모두 그럭저럭 괜찮은 에스프레소 가판대가 있으며, 이들은 그래프 G의 두 특정 정점 s와 t로 표현됩니다.

(a) 카페인 금단 편두통에 시달리지 않고 Giggleplex에서 Gigglesburg까지 자전거로 갈 수 있는지 판별하는 알고리즘을 설명하고 분석하십시오.

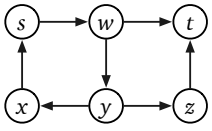
(b) 더 비싼 페도라를 쓰면 에스프레소 한 잔과 다음 잔 사이에 자전거로 이동할 수 있는 거리 L을 늘릴 수 있다는 사실을 알게 되었습니다. 카페인 금단으로 인한 편두통을 겪지 않고 Giggleplex에서 Gigglesburg까지 자전거로 이동할 수 있게 해 주는 L의 최솟값을 구하는 알고리즘을 설명하고 분석하십시오.

(c) 그 승차가 불가능하다고 상사(Giggle이 최근 경쟁사 Ünter에서 스카우트해 온 사람입니다)에게 보고하자, 상사는 여러분의 지도를 보여 달라고 요구합니다. "아, 문제가 뭔지 알겠습니다. 이 지도에는 *스타벅스*가 하나도 없습니다!" 여러분이 경악하며 지켜보는 가운데, 상사는 미국 내 모든 스타벅스 매장마다 정점을 하나씩 포함한 갱신된 그래프 G'을 건네줍니다. 친절하게도 각 정점은 스타벅스 그린(Pantone® 3425 C)으로 표시되어 있습니다.

카페인 금단으로 인한 편두통을 겪지 않고 기글플렉스에서 기글스버그까지 자전거로 가려면 반드시 들러야 하는 스타벅스 매장의 최소 개수를 구하는 알고리즘을 설명하고 분석하십시오. 좀 더 형식적으로 말하면, 여러분의 알고리즘은 G'에서 길이가 L 이하인 간선만 사용하여 s에서 t로 가는 경로들 가운데 초록색 정점이 가장 적은 경로의 초록색 정점 개수를 구해야 합니다.

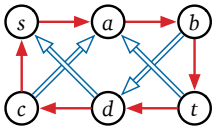
16. 음이 아닌 가중치를 갖는 간선들로 이루어진 방향 그래프 $G = (V, E)$와 두 정점 s와 t가 주어졌다고 가정합시다. 간선의 개수가 3으로 나누어떨어지는, G에서 s로부터 t까지의 최단 보행(정점 및/또는 간선의 반복이 허용될 수 있음)을 찾는 알고리즘을 설명하고 분석하십시오.

예를 들어, 아래 그래프에서 정점 s와 t가 표시된 대로 주어지고 모든 간선의 가중치가 1이라면, 여러분의 알고리즘은 6을 반환해야 합니다. 이는 보행 $s \to w \to y \to x \to s \to w \to t$의 길이이며, 이 보행의 길이는 6입니다.



17. 음이 아닌 가중치가 매겨진 간선들로 이루어진 방향 그래프 G가 주어지며, 그중 일부 간선은 빨간색이고 나머지 간선은 파란색이라고 가정합시다. G에서 한 정점 s로부터 다른 정점 t까지 가는 보행(walk) 가운데, 연속된 세 간선이 모두 같은 색인 경우가 없는 최단 보행을 찾는 알고리즘을 설명하십시오. 즉, 보행에 빨간 간선이 두 개 연속으로 나타나면 다음 간선은 반드시 파란색이어야 하고, 파란 간선이 두 개 연속으로 나타나면 다음 간선은 반드시 빨간색이어야 합니다.

    예를 들어, 모든 빨간 간선의 가중치가 1이고 모든 파란 간선의 가중치가 2인 다음 그래프가 입력으로 주어지면, s에서 t로 가는 가장 짧은 합법적 보행이 $s \to a \to b \Rightarrow d \to c \Rightarrow a \to b \to c$이므로 알고리즘은 정수 9를 반환해야 합니다.



18. 각 간선에 음이 아닌 가중치가 있고 각 간선이 빨간색, 흰색, 파란색 중 하나로 칠해진 방향 그래프 G를 생각해 봅시다. G의 보행에서 간선 색의 순서가 빨간색, 흰색, 파란색, 빨간색, 흰색, 파란색, ⋯ 과 같이 이어지면 그 보행을 *프랑스 국기 보행*이라고 부릅니다. 더 엄밀하게 말하면, 보행 $v_0 \to v_1 \to \cdots \to v_k$는 모든 정수 i에 대해 간선 $v_i \to v_{i+1}$이 $i \bmod 3 = 0$이면 빨간색, $i \bmod 3 = 1$이면 흰색, $i \bmod 3 = 2$이면 파란색일 때 프랑스 국기 보행입니다.

    G의 한 시작 정점 s에서 다른 모든 정점까지 가는 *최단* 프랑스 국기 보행을 찾는 알고리즘을 설명하십시오.

19. n개의 은하가 m개의 은하 간 순간이동 통로로 연결되어 있습니다. 각 순간이동 통로는 두 은하를 잇고 양방향으로 지나갈 수 있습니다. 또한 각 순간이동 통로 e에는 $c(e)$달러의 비용이 매겨져 있으며, 여기서 $c(e)$는 양의 정수입니다. 순간이동 통로는 여러 번 사용할 수 있지만, 사용할 때마다 통행료를 내야 합니다.

    주디는 은하 s에서 은하 t까지 가능한 한 적은 비용으로 이동하려고 합니다. 하지만 잔돈을 들고 다니는 것도 달갑지 않으므로 총비용이 5달러의 배수가 되기를 원합니다.

    (a) 총비용이 5달러의 배수여야 한다는 제약 조건 아래에서, 은하 s에서 은하 t까지 이동하는 데 드는 최소 총비용을 계산하는 알고리즘을 설명하고 분석하십시오.

    (b) (a) 부분을 풀되, 이번에는 주디가 순간이동 통로 하나를 정확히 한 번 무료로 사용할 수 있는 쿠폰을 가지고 있다고 가정하십시오.

20. 새 도시로 이사한 뒤, 여러분은 집에서 새 사무실까지 걸어갈 경로를 정하기로 합니다. 매일 운동을 제대로 하기 위해, 경로는 오르막길(운동용) 뒤에 내리막길(몸을 식히기 위한 용도)이 이어지는 형태이거나, 오르막길로만 이루어지거나, 내리막길로만 이루어져야 합니다. (집에 돌아올 때도 같은 길로 걸을 것이므로, 어느 쪽으로든 운동은 하게 됩니다.) 하지만 실제로 제시간에 출근하려면, 이 조건을 만족하는 경로 중 *가장 짧은* 경로를 원하기도 합니다.

    입력은 무방향 그래프 G와 시작 정점 s, 대상 정점 t로 이루어집니다. G의 정점은 교차로를, 간선은 도로 구간을 나타냅니다. 모든 정점 v에는 그 교차로의 해발 높이인 값 $h(v)$가 연관되어 있고, 각 간선 uv에는 그 도로 구간의 길이인 값 $\ell(uv)$가 연관되어 있습니다.

    (a) s에서 t로 가는 가장 짧은 오르막–내리막 보행을 찾는 알고리즘을 설명하고 분석하십시오. 모든 정점의 높이는 서로 다르다고 가정합니다.

    (b) 이제 일부 또는 모든 정점의 높이가 같을 수 있다고 가정합시다. s에서 t로 가는 최단 "오르막 후 내리막" 보행을 찾는 알고리즘을 설명하고 분석하십시오. 보행의 "오르막" 부분과 "내리막" 부분 모두에서 평평한 간선을 사용해도 됩니다.

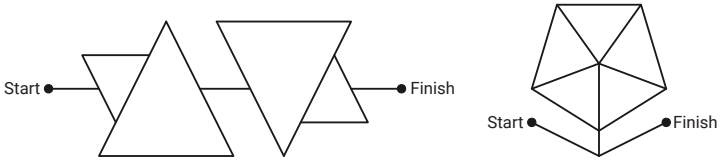
    (c) 마지막으로, s에서 t로 가는 경로 중 원하는 구조를 갖는 경로가 없다는 사실을 알게 되었다고 가정합시다. "오르막" 부분 경로와 "내리막" 부분 경로가 번갈아 바뀌는 횟수가 가능한 한 적고, 그러한 경로 중에서 길이가 최소인 s에서 t로의 경로를 찾는 알고리즘을 설명하십시오.

21. Sham-Poobanana 대학교를 졸업한 뒤 여러분은 비행을 싫어하는 사람들을 위한 최고의 여행사인 Aerophobes-Я-Us에 취직합니다. 여러분이 맡은 일은 고객이 한 도시에서 다른 도시로 가는 비행기 여행을 계획하도록 돕는 시스템을 만드는 것입니다. 여러분의 고객은 모두 비행을(더 나아가 공항도) 두려워하므로, 여러분이 계획하는 여행은 모두 가능한 한 짧아야 합니다. 여러분은 지구상에 있는 모든 항공편의 출발 시각과 도착 시각을 전부 알고 있습니다.

    고객 중 한 명이 도시 X에서 도시 Y로 비행기를 타고 가려 한다고 가정합시다. *총 이동 시간*, 즉 최초 출발부터 최종 도착까지 걸리는 시간을 최소화하는 항공편 순서를 찾는 알고리즘을 설명하십시오. 이 시간에는 중간 공항에서 연결 항공편을 기다리는 시간도 포함됩니다.

22. 5장의 연습문제 20에서 여러분은 주어진 *예각 미로*를 풀 수 있는지 판정하는 알고리즘을 설계했습니다. 이 문제에서는 "길이"에 대한 서로 다른 두 가지 정의 각각에 대해, 주어진 예각 미로를 통과하는 *최단* 보행을 찾는 알고리즘을 설계합니다.

출발점에서 도착점까지 예각으로만 꺾이는 경로를 그려 아래의 각 각도 미로를 완성하십시오.



입력은 연결된 무방향 그래프 G입니다. 이 그래프의 정점은 평면 위의 점이고 간선은 선분입니다. 간선끼리는 끝점 말고는 서로 교차하지 않습니다. 예를 들어 문자 X를 그린 그림에는 정점이 다섯 개, 간선이 네 개 있으며, 위의 첫 번째 미로에는 정점이 14개, 간선이 21개 있습니다. 또한 두 정점 Start와 Finish가 주어집니다.

G에서 Start부터 Finish까지의 보행이 예각만을 포함하면, 더 형식적으로 말해 연속한 임의의 두 간선 $u \to v \to w$에 대해 $\angle uvw = \pi$이거나 $0 < \angle uvw < \pi/2$이면, 그 보행은 *유효*합니다. 주어진 두 선분 사이의 각이 평각인지, 둔각인지, 직각인지, 예각인지를 $O(1)$ 시간에 판별할 수 있다고 가정합니다.

(a) Start에서 Finish까지 가는 유효한 보행 중 지나는 구간의 수가 가장 적은 보행을 계산하는 알고리즘을 설명하십시오. (보행이 같은 구간을 두 번 지나면 두 번으로 셉니다.)

(b) Start에서 Finish까지 가는 유효한 보행 중 방향 전환 횟수가 가장 적은 보행을 계산하는 알고리즘을 설명하십시오. *[힌트: 이것은 (a)와 같은 문제가 아닙니다.]*

(c) 전체 유클리드 길이가 가능한 한 짧은, Start에서 Finish까지의 유효한 보행(walk)을 계산하는 알고리즘을 설명하십시오. (임의의 선분의 길이도 $O(1)$ 시간에 계산할 수 있다고 가정하십시오.)

23. See-Bull 가짜 뉴스 탐지 센터에서 고된 중간고사를 치른 뒤, 여러분은 버스를 타고 집에 가기로 합니다. 미리 계획을 세워 두었으므로, 여러분에게는 Sham-Poobanana의 모든 버스가 서는 모든 정류장의 시각과 위치가 적힌 시간표가 있습니다. 안타깝게도 See-Bull 센터와 여러분의 집을 모두 지나는 버스는 하나도 없으므로, 적어도 한 번은 버스를 갈아타야 합니다. 서로 다른 버스는 정확히 b대 있습니다. 각 버스는 오전 0시 00분 01초에 운행을 시작하여 정확히 n개의 정류장에 정차하고, 마지막으로 오후 11시 59분 59초에 운행을 마칩니다. 버스는 언제나 시간표대로 정확히 운행하며, 여러분에게는 정확한 시계가 있습니다. 마지막으로, 여러분은 너무 지쳐서 버스 정류장 사이를 걸어서 이동할 수 없습니다.

(a) 최대한 일찍 집에 도착할 수 있는 버스 탑승 순서를 구하는 알고리즘을 설명하고 분석하십시오. 목표는 이동하는 데 걸리는 시간이 아니라 *도착 시각*을 최소화하는 것입니다.

(b) 이런! 중간고사가 핼러윈에 치러졌는데, 거리에는 좀비가 들끓고 있습니다! Sham-Poobanana 대중교통 지구에는 버스를 추가로 투입하거나 좀비를

막아 주는 버스 정류장을 설치할 예산이 없습니다. 1년에 단 하룻밤을 위해서라면 더욱 그렇습니다. *버스 정류장에서 기다리는 총 시간*을 최소화하는 버스 탑승 순서를 구하는 알고리즘을 서술하고 분석하십시오. 집에 얼마나 늦게 도착하는지, 버스 안에서 얼마나 오래 있는지는 상관하지 않습니다. (첫 버스가 막 출발하기 직전까지는 See-Bull Center 안에서 기다릴 수 있다고 가정합니다.)

24. 화려한 봄방학을 보내고 돌아온 첫날 아침, 앨리스는 잠에서 깨어 자신의 차에 시동이 걸리지 않는다는 사실을 알게 됩니다. 그래서 앨리스는 대중교통을 이용해 Sham-Poobanana 대학교의 수업에 가야 합니다. 앨리스에게는 Poobanana 카운티의 전체 대중교통 시간표가 있습니다. 시간표에서 버스 노선은 방향 그래프 G로 표현되며, 이 그래프의 정점은 버스 정류장을 나타내고 간선은 그 정류장들 사이의 버스 노선을 나타냅니다. 각 간선 $u \to v$마다 시간표에는 세 개의 양의 실수가 기록되어 있습니다.

- $\ell(u \to v)$는 정류장 u에서 정류장 v까지 버스로 이동하는 데 걸리는 시간(분 단위)입니다
- $f(u \to v)$는 버스가 정류장 u를 떠나 정류장 v로 출발하는 가장 이른 시각(자정부터 경과한 분 단위)입니다.
- $\Delta(u \to v)$는 정류장 u에서 정류장 v로 가는 연속된 두 출발 사이의 시간 간격(분 단위)입니다.

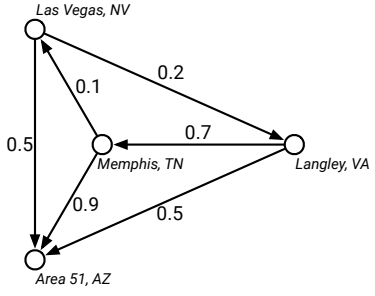
따라서 이 노선의 첫 번째 버스는 시각 $f(u \to v)$에 u를 출발하여 시각 $f(u \to v) + \ell(u \to v)$에 v에 도착하고, 두 번째 버스는 시각 $f(u \to v) + \Delta(u \to v)$에 u를 출발하여 시각 $f(u \to v) + \Delta(u \to v) + \ell(u \to v)$에 v에 도착하며, 세 번째 버스는 시각 $f(u \to v) + 2 \cdot \Delta(u \to v)$에 u를 출발하여 시각 $f(u \to v) + 2 \cdot \Delta(u \to v) + \ell(u \to v)$에 v에 도착하는 식으로 계속됩니다.

앨리스는 특정 시각에 정류장 s(자신의 집)에서 출발하여 정류장 t(See-Bull 센터)에 최대한 빨리 도착하려고 합니다. 앨리스가 한 버스를 타고 어떤 정류장에 도착한 시각이 다른 버스의 출발 예정 시각과 정확히 같다면, 앨리스는 그 두 번째 버스로 갈아탈 수 있습니다. 앨리스는 SPU 학생이라 버스를 무료로 탈 수 있으므로, 버스를 몇 번 갈아타야 하는지는 신경 쓰지 않습니다.

앨리스가 목적지에 도착할 수 있는 가장 이른 시각을 구하는 알고리즘을 설명하고 분석하십시오. 입력은 방향 그래프 $G = (V, E)$, 정점 s와 t, 각 간선 $e \in E$에 대한 값 $\ell(e), f(e), \Delta(e)$, 그리고 앨리스의 출발 시각(자정 이후 경과한 분 단위)으로 이루어집니다. *[힌트: 이번처럼 드문 경우에는 입력 그래프를 수정하는 것보다 알고리즘을 수정하는 편이 더 쉬울 수 있습니다.]*

25. 멀더와 스컬리는 미국의 모든 도로에 대해, 그 도로를 운전하는 사람이 외계인에게 납치되지 *않을* 정확한 확률을 계산해 두었습니다. 멀더 요원은 버지니아주 랭글리에서 네바다주 51구역까지 차를 몰고 가야 합니다. 납치될 확률을 최소로 하려면 그는 어떤 경로를 택해야 합니까?

좀 더 형식적으로 말하면, 모든 간선 e가 독립적인 안전 확률 $p(e)$를 갖는 방향 그래프 $G = (V, E)$가 주어집니다. 경로의 *안전도*는 그 경로에 속한 간선들의 안전 확률을 모두 곱한 값입니다. 주어진 시작 정점 s에서 주어진 대상 정점 t까지 가장 안전한 경로를 구하는 알고리즘을 설계하고 분석하십시오. 필요한 산술 연산은 모두 $O(1)$ 시간에 수행할 수 있다고 가정해도 됩니다.



예를 들어 위에 제시된 확률에 따르면, 멀더가 랭글리에서 51구역까지 곧장 차를 몰고 가려 할 경우 납치되지 않고 그곳에 도착할 확률은 50%입니다. 멤피스에 들를 경우 무사히 도착할 확률은 $0.7 \times 0.9 = 63\%$입니다. 먼저 멤피스에 들렀다가 라스베이거스에 들를 경우에는 납치될 확률이 $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$나 됩니다! (아시다시피, 엘비스도 그렇게 잡혀간 것입니다.)

♣26. 서니데일 국립공원에서 하룻밤 캠핑을 하던 중, 여러분은 뒤척이며 선잠을 자다가 비명 소리에 잠을 깹니다. 무슨 일인지 알아보려고 텐트에서 기어 나오자, 겁에 질린 공원 관리인이 피투성이가 된 채 구겨진 종이 한 장을 가슴에 꼭 움켜쥐고 숲에서 뛰쳐나옵니다. 관리인은 여러분의 텐트에 다다르자 숨을 헐떡이며 "도망치시오. . . 아직. . . 당신이. . . "라고 말하고는, 종이를 여러분의 손에 쥐여 주고 땅에 쓰러집니다. 맥박을 확인해 보니 관리인은 이미 완전히 숨이 끊어져 있습니다. 종이를 내려다보니 공원 지도라는 것을 알 수 있습니다. 이 지도는 무방향 그래프로 그려져 있으며, 정점은 공원의 명소를, 간선은 명소 사이의 산책로를 나타냅니다. (산책로는 명소에서 시작하여 명소에서 끝나며, 서로 교차하지 않습니다.) 정점 중 하나가 여러분의 현재 위치라는 것을 알아볼 수 있고, 지도 가장자리에 있는 정점 몇 개에는 EXIT라고 적혀 있습니다. 더 자세히 살펴보니, 누군가가 (아마도 가엾게 죽은 저 공원 관리인이) 각 정점과 각 간선 옆에 0과 1 사이의 실수를 적어 두었습니다. 지도 뒷면에 휘갈겨 쓴 메모에 따르면, 간선 옆의 숫자는 해당 산책로에서 뱀파이어와 마주칠 확률이고, 정점 옆의 숫자는 해당 명소에서 뱀파이어와 마주칠 확률입니다. (뱀파이어들은 서로 함께 있는 것을 견디지 못하므로, 같은 산책로나 같은 명소에서 뱀파이어를 두 마리 이상 보는 일은 결코 없습니다.) 메모는 표시된 산책로를 벗어나면 느리고 고통스러운 죽음을 맞게 된다고 경고합니다. 여러분은 발치에 놓인 시체를 흘끗 내려다봅니다. 그렇습니다, 그의 죽음은 분명 고통스러워 보였습니다. 잠깐, 방금 움찔하지 않았습니까? 이빨이 점점 길어지고 있지 않습니까? 되살아난 관리인의 심장에 텐트 말뚝을 박아 넣은 뒤, 여러분은 현명하게도 *즉시* 최대한 빨리 공원을 떠나기로

결심합니다. 현재 위치에서 임의의 EXIT 노드까지 가는 경로 중에서, 그 경로를 따라가며 마주치는 뱀파이어의 *기대 수*의 총합이 가능한 한 작은 경로를 찾는 효율적인 알고리즘을 설명하고 분석하십시오. *정점 확률과 간선 확률을 모두* 반드시 고려하십시오. *[힌트: 정점 확률이 없더라도 이 문제는 앞의 문제와 같지 않습니다!]*

아름드리 나무도 털끝만 한 싹에서 자라났고,
구 층 누대도 (작은) 흙더미에서 솟아올랐으며,
천 리 길도 한 걸음에서 시작되었습니다.

— 노자, *도덕경*, 제64장 (기원전 6세기),
제임스 레그 영역 (1891)

저는 오백 마일을 걷겠습니다,
그리고 오백 마일을 더 걷겠습니다,
오직 천 마일을 걸어와
당신의 문 앞에 쓰러지는 남자가 되기 위해서입니다.

— The Proclaimers, "I'm Gonna Be (500 Miles)",
Sunshine on Leith (2001)

거의 다 왔습니다…거의 다 왔습니다…

— 레드 리더 [드루 헨리], *스타워즈* (1977)

9

모든 쌍 최단 경로

9.1 서론

이전 장에서는 s를 루트로 하는 최단 경로 트리를 구성하여, 하나의 시작 정점 s에서 그래프의 다른 모든 정점까지의 최단 경로를 찾는 여러 알고리즘을 살펴보았습니다. 최단 경로 트리는 그래프의 각 노드 v에 대해 다음 두 가지 정보를 지정합니다.

- $dist(v)$는 s에서 v까지의 최단 경로의 길이입니다;
- $pred(v)$는 s에서 v까지의 최단 경로에서 v 바로 앞의 정점입니다.

이 장에서는 더 일반적인 *모든 쌍 최단 경로* 문제를 살펴봅니다. 이 문제는 가능한 모든 출발점에서 가능한 모든 도착점까지의 최단 경로를 구하는 문제입니다. 모든 정점 쌍 u와 v에 대해 다음 정보를 계산하고자 합니다.

- $dist(u, v)$는 u에서 v까지의 최단 경로의 길이입니다;
- $pred(u, v)$는 u에서 v까지의 최단 경로에서 v 바로 앞에 있는 정점입니다.

이러한 직관적인 정의는 몇 가지 경계 사례를 배제하는데, 이들은 모두 이전 장에서 이미 살펴본 것입니다.

- u에서 v로 가는 경로가 없으면 u에서 v로 가는 *최단* 경로도 존재하지 않습니다. 이 경우에는 $dist(u, v) = \infty$, $pred(u, v) = \text{NULL}$로 정의합니다.

- u와 v 사이에 음수 사이클이 있으면, u에서 v로 가는 경로[1] 중에는 길이가 임의로 작은 음수인 것이 존재합니다. 이 경우 $dist(u, v) = -\infty$ 및 $pred(u, v) = \text{NULL}$로 정의합니다.
- 마지막으로, u가 음수 사이클 위에 있지 않다면 u에서 자기 자신으로 가는 최단 경로에는 간선이 없으므로 마지막 간선도 없습니다. 이 경우 $dist(u, u) = 0$, $pred(u, u) = \text{NULL}$로 정의합니다.

모든 쌍 최단 경로 문제에서 원하는 출력은 $V \times V$ 크기의 배열 두 개입니다. 하나는 V^2개의 최단 경로 거리를 모두 저장하고,[2] 다른 하나는 V^2개의 선행자를 모두 저장합니다. 이 장에서는 거의 전적으로 거리 배열을 계산하는 데 집중하겠습니다. 실제 최단 경로를 계산하는 데 쓰이는 선행자 배열은 약간만 수정하면 계산할 수 있습니다(힌트, 힌트).

9.2 수많은 단일 출발점

모든 쌍 최단 경로 문제의 가장 자명한 해법은 가능한 출발 정점마다 한 번씩, 단일 출발점 최단 경로 알고리즘을 총 V번 실행하는 것입니다. 구체적으로 말하면, 1차원 부분 배열 $dist[s, \cdot]$을 채우기 위해 출발 정점 s에서 시작하는 단일 출발점 알고리즘을 호출합니다.

```
OBVIOUSAPSP(V, E, w):
  for every vertex s
    dist[s, ·] ← SSSP(V, E, w, s)
```

이 알고리즘의 실행 시간은 당연히 어떤 단일 출발점 최단 경로 알고리즘을 사용하느냐에 따라 달라집니다. 단일 출발점 설정에서와 마찬가지로, 그래프의 구조와 간선 가중치에 따라 자연스러운 선택지가 네 가지 있습니다:

- 그래프의 간선에 가중치가 없다면, 너비 우선 탐색을 사용하여 전체 실행 시간 $\boldsymbol{O(VE)} = O(V^3)$을 얻습니다.
- 그래프에 사이클이 없으면 정점들을 위상 순서로 훑는 방법으로도 전체 실행 시간 $\boldsymbol{O(VE)} = O(V^3)$을 얻을 수 있습니다.
- 모든 간선 가중치가 음수가 아니면, 데이크스트라 알고리즘을 사용하여 $\boldsymbol{O(VE \log V)} = O(V^3 \log V)$의 실행 시간을 얻습니다.[3]
- 마지막으로, 가장 일반적인 설정에서는 벨먼-포드 알고리즘이 전체 실행 시간 $\boldsymbol{O(V^2E)} = O(V^4)$를 제공합니다.

[1] 엄밀히 말하면 보행

[2] 도로 지도를 종이에 인쇄해서 직접 손으로 찾아봐야 했던 시절에는 지도에 삼각형 모양의 "거리표"가 실려 있는 경우가 꽤 흔했습니다. 예를 들어 샘페인에서 콜럼버스까지의 거리를 찾으려면 "샘페인"이라고 적힌 행과 "콜럼버스"라고 적힌 열을 보면 되었습니다.

[3] 이번에도 데이크스트라 알고리즘 구현에서 이진 힙을 정렬되지 않은 배열로 바꾸면 (그래프에 간선이 얼마나 많든 상관없이) 전체 실행 시간은 $O(V^3)$이 되고, 이진 힙을 피보나치 힙으로 바꾸면 실행 시간은 $O(V(E + V \log V)) = O(VE + V^2 \log V) = O(V^3)$으로 줄어듭니다.

9.3 가중치 재조정

음수 간선은 속도를 크게 떨어뜨립니다. 음수 간선을 없앨 수 있을까요? 많은 사람이 쉽게 떠올리는 간단한 아이디어는 모든 간선의 가중치를 같은 양만큼 늘려서 모든 가중치를 양수로 만든 다음, 벨먼-포드 대신 데이크스트라 알고리즘을 사용하는 것입니다. 안타깝게도 이 간단한 아이디어는 통하지 않습니다. 직관적으로 말하면, "길이"에 대한 우리의 두 가지 자연스러운 개념이 서로 양립하지 않기 때문입니다. 간선이 더 많은 경로가 간선이 더 적은 경로보다 전체 가중치가 더 작을 수 있습니다. 모든 간선의 가중치를 같은 비율로 늘리면 간선이 더 많은 경로는 간선이 더 적은 경로보다 더 빨리 길어집니다. 그 결과 두 정점 사이의 최단 경로가 바뀔 수 있습니다.

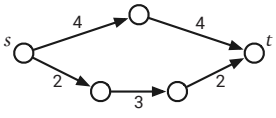


그림 9.1. 모든 간선의 가중치를 2씩 늘리면 s에서 t로 가는 최단 경로가 바뀝니다.

그러나 최단 경로를 실제로 보존하면서 간선의 가중치를 다시 매기는 더 교묘한 방법이 있습니다. 이 가중치 재조정 방법은 흔히 도널드 존슨(Donald Johnson)의 공으로 알려져 있으며, 존슨은 1973년에 이 방법을 최단 경로 알고리즘에 적용하는 방법을 설명했습니다. 그러나 사실 존슨은 이 방법의 공을 잭 에드먼즈(Jack Edmonds)와 리처드 카프(Richard Karp)의 1972년 논문으로 돌렸습니다. 같은 방법을 1971년에 도미자와 노부아키(Nobuaki Tomizawa)도 설명했으며, 1961년에는 델버트 풀커슨(Delbert Fulkerson)이 조금 다른 형태로 설명했습니다.

각 정점 v에 양수, 음수 또는 0일 수 있는 *가격* $\pi(v)$가 연관되어 있다고 가정합시다. 새로운 가중치 함수 w'를 다음과 같이 정의할 수 있습니다:

$$w'(u \to v) = \pi(u) + w(u \to v) - \pi(v)$$

직관적으로 이해하기 위해, 정점 u를 떠날 때는 $\pi(u)$만큼의 출국세를 내야 하고, v에 들어갈 때는 입국 선물로 $\pi(v)$를 받는다고 상상해 보십시오.

새로운 가중치 함수 w'에 대한 최단 경로가 원래 가중치 함수 w에 대한 최단 경로와 정확히 같다는 것을 보이는 것은 어렵지 않습니다. 실제로 한 정점 u에서 다른 정점 v로 가는 *임의의* 경로 $u \rightsquigarrow v$에 대해 다음이 성립합니다.

$$w'(u \rightsquigarrow v) = \pi(u) + w(u \rightsquigarrow v) - \pi(v).$$

출구 요금으로 $\pi(u)$를 내고, 여기에 경로의 원래 가중치를 더한 다음, 입구 선물로 받는 $\pi(v)$를 뺍니다. 경로 위의 모든 중간 정점 x에서는 입구 선물로 $\pi(x)$를 받지만, 곧바로 출구세로 그만큼을 다시 냅니다! u에서 v로 가는 모든 경로의 길이는 정확히 같은 양만큼 바뀌므로, u에서 v로 가는 최단 경로는 바뀌지 않습니다. (서로 다른 정점 쌍 사이의 경로들은 길이가 서로 다른 양만큼 바뀔 수 있으므로, 그 순서는 바뀔 수 있습니다.)

9.4 존슨 알고리즘

존슨의 모든 쌍 최단 경로 알고리즘은 모든 간선의 새로운 가중치가 음이 아니도록 각 정점에 대해 비용 $\pi(v)$를 계산한 다음, 데이크스트라 알고리즘을 사용하여 새로운 가중치에 대한 최단 경로를 계산합니다.

먼저, 입력 그래프에 다른 모든 정점에 도달할 수 있는 정점 s가 있다고 가정합시다. 존슨 알고리즘은 벨먼-포드 알고리즘(간선 가중치가 음수여도 상관하지 않습니다)으로 s에서 다른 정점들까지의 최단 경로를 계산한 다음, 가격 함수 $\pi(v) = dist(s, v)$를 사용하여 그래프의 가중치를 다시 매깁니다. 모든 간선의 새 가중치는 다음과 같습니다.

$$w'(u \to v) = dist(s, u) + w(u \to v) - dist(s, v).$$

이 새로운 가중치들은 벨먼-포드 알고리즘이 정지했기 *때문에* 음이 아닙니다! $dist(s, u) + w(u \to v) < dist(s, v)$이면 간선 $u \to v$가 *긴장된* 간선이라는 것과, 단일 출발점 최단 경로 알고리즘은 모든 긴장된 간선을 제거한다는 것을 떠올려 보십시오. (벨먼-포드 알고리즘이 음수 사이클을 발견하면 최단 경로가 잘 정의되지 않으므로 존슨 알고리즘은 중단됩니다.)

모든 정점에 도달할 수 있는 적절한 정점 s가 없다면, 벨먼-포드를 어디에서 시작하든 결과로 얻는 정점 가격 중 일부는 무한대가 됩니다. 이 문제를 피하기 위해, 우리는 *항상* 그래프에 새로운 정점 s를 추가하고, s에서 다른 정점들로 가는 가중치 0인 간선들을 넣되, s로 돌아 들어오는 간선은 *전혀* 넣지 않습니다. s로 들어오는 경로가 없으므로, 이렇게 추가하더라도 원래 정점들의 어떤 쌍 사이의 최단 경로도 바뀌지 않습니다.

존슨 알고리즘의 전체 의사코드는 그림 9.2에 나와 있습니다. 이 알고리즘의 실행 시간은 다익스트라 알고리즘 호출이 좌우합니다. 구체적으로, BellmanFord를 한 번 실행하는 데 $O(VE)$ 시간, Dijkstra를 V번 실행하는 데 $O(VE \log V)$ 시간, 그 밖의 부수적인 처리에 $O(V + E)$ 시간이 듭니다. 따라서 전체 실행 시간은 $\boldsymbol{O(VE \log V)} = O(V^3 \log V)$입니다.[4] 결국 음수 간선이 있어도 느려지지 않습니다!

9.5 동적 계획법

단일 출발점 알고리즘을 호출하는 대신, 동적 계획법을 사용하여 모든 쌍 최단 경로 문제를 직접 풀 수도 있습니다. $E = \Omega(V^2)$인 *조밀한* 그래프의 경우, 동적 계획법 접근 방식은 결국 Johnson 알고리즘보다 더 간단하면서도 (약간) 더 빠른 알고리즘을 만들어 냅니다.

이 장의 나머지 부분에서는 입력 그래프에 음수 사이클이 없다고 가정하겠습니다.

동적 계획법 알고리즘에서 늘 그렇듯이, 먼저 점화식이 필요합니다.

단일 출발점 설정에서와 마찬가지로, "당연해 보이는" 재귀적 정의는

$$dist(u, v) = \begin{cases} 0 & \text{if} u = v \\ \min\limits_{x \to v} \big(dist(u, x) + w(x \to v) \big) & \text{그렇지 않으면} \end{cases}$$

[4] . . . 기본 이진 힙 구현을 가정한 경우입니다. 앞의 각주를 참고하십시오.

```
JohnsonAPSP(V, E, w) :
《인공 소스를 추가합니다》
  add a new vertex s
  for every vertex v
      add a new edge s→v
      w(s→v) ← 0
《정점 가격을 계산합니다》
  dist[s, ·] ← BellmanFord(V, E, w, s)
  if BellmanFord found a negative cycle
      fail gracefully
《간선의 가중치 재조정》
  for every edge u→v ∈ E
      w'(u→v) ← dist[s, u] + w(u→v) − dist[s, v]
《가중치를 재조정한 최단 경로 거리를 계산합니다》
  for every vertex u
      dist'[u, ·] ← Dijkstra(V, E, w', u)
《원래 최단 경로 거리를 계산합니다》
  for every vertex u
      for every vertex v
          dist[u, v] ← dist'[u, v] − dist[s, u] + dist[s, v]
```

그림 9.2. 존슨의 모든 쌍 최단 경로 알고리즘

입력 그래프가 dag일 때만 동작하며, 방향 사이클이 하나라도 있으면 점화식이 무한 루프에 빠집니다.

벨먼-포드에서 했던 것과 똑같이 추가 매개변수를 도입하여 이 무한 루프를 끊을 수 있습니다. $dist(u, v, \ell)$이 u에서 v로 가는 경로 중 *최대 ℓ개의 간선*을 사용하는 최단 경로의 길이를 나타낸다고 합시다. 임의의 두 정점 사이의 최단 경로는 최대 $V-1$개의 간선을 지나므로, 실제 최단 경로 거리는 $dist(u, v, V-1)$입니다. 벨먼의 단일 출발점 점화식은 이 설정에 곧바로 적용됩니다:

$$dist(u,v,\ell) = \begin{cases} 0 & \text{if } \ell = 0 \text{ 그리고 } u = v \\ \infty & \text{if } \ell = 0 \text{ 그리고 } u \neq v \\ \min \left\{ \begin{array}{c} dist(u,v,\ell-1) \\ \min\limits_{x \to v}(dist(u,x,\ell-1) + w(x \to v)) \end{array} \right\} & \text{그렇지 않으면} \end{cases}$$

이 점화식을 동적 계획법 알고리즘으로 바꾸는 것은 간단합니다. 그렇게 얻은 알고리즘은 $O(V^2E) = O(V^4)$ 시간에 실행됩니다.

```
ShimbelAPSP(V, E, w):
  for all vertices u
      for all vertices v
          if u = v
              dist[u, v, 0] ← 0
          else
              dist[u, v, 0] ← ∞
  for ℓ ← 1 to V − 1
      for all vertices u
          for all vertices v ≠ u
              dist[u, v, ℓ] ← dist[u, v, ℓ − 1]
              for all edges x→v
                  if dist[u, v, ℓ] > dist[u, x, ℓ − 1] + w(x→v)
                      dist[u, v, ℓ] ← dist[u, x, ℓ − 1] + w(x→v)
```

이 알고리즘은 1954년 Alfonso Shimbel이 처음 개략적으로 제시했습니다.[5] 벨먼이 정식화한 벨먼-포드와 마찬가지로, 정점 v에 대한 안쪽 루프나 반복 인덱스 ℓ은 필요하지 않습니다. 수정된 알고리즘은 아래와 같습니다.

```
AllPairsBellmanFord(V, E, w):
  for all vertices u
      for all vertices v
          if u = v
              dist[u, v] ← 0
          else
              dist[u, v] ← ∞
  for ℓ ← 1 to V − 1
      for all vertices u
          for all edges x→v
              if dist[u, v] > dist[u, x] + w(x→v)
                  dist[u, v] ← dist[u, x] + w(x→v)
```

우리가 이 알고리즘을 유도한 방식을 생각하면, 결과로 얻은 알고리즘이 서로 다른 출발 정점을 갖는 V개의 벨먼-포드 실행을 번갈아 수행하는 것과 정확히 같다는 사실은 그리 놀랍지 않습니다. 특히 모든 정점 u와 v에 대해, 주 for-루프의 ℓ번째 반복이 끝난 후 $dist[u, v]$는 u에서 v로 가는 경로 가운데 간선을 최대 ℓ개 포함하는 최단 경로의 길이 *이하*입니다.

9.6 분할 정복

하지만 1971년에 Michael Fischer와 Albert Meyer가 제안한 더 중요한 개선을 할 수 있습니다. 벨먼의 점화식은 대상 정점의 가능한 모든 선행자를 고려하여 최단 경로를 조금 더 짧은 경로와 하나의 간선으로 나눕니다. 그 대신 최단 경로를 *중간* 정점에서 두

[5] Shimbel은 입력이 $V \times V$ 크기의 완전한 거리 행렬이라고 가정했으므로, 그의 원래 알고리즘은 그래프의 간선 수와 관계없이 실제로 $O(V^4)$ 시간에 실행됩니다.

개의 더 짧은 최단 경로로 나누어 봅시다. 이 아이디어로부터 같은 함수 $dist(u, v, \ell)$에 대한 다른 점화식을 얻습니다. 여기서는 $\ell = 0$ 대신 기저 사례 $\ell = 1$에서 멈추어야 합니다. 간선이 최대 하나인 경로에는 "중간" 정점이 없기 때문입니다.

점화식을 조금 더 간단하게 하기 위해, 모든 정점 v에 대해 $w(v \to v) = 0$으로 정의합시다.

$$dist(u, v, \ell) = \begin{cases} w(u \to v) & \text{if } i = 1 \\ \min_x \big(dist(u, x, \ell/2) + dist(x, v, \ell/2) \big) & \text{그렇지 않으면} \end{cases}$$

위에 쓴 형태 그대로라면 이 점화식은 ℓ이 2의 거듭제곱일 때만 성립합니다. 그렇지 않으면 간선이 (최대) 분수 개인 최단 경로를 찾으려 하게 될 수도 있기 때문입니다! 하지만 이것은 실제로 문제가 되지 않습니다. $dist(u, v, \ell)$은 $\ell \geq V - 1$인 모든 ℓ에 대해 u에서 v까지의 실제 최단 경로 거리입니다. 특히 $\ell = 2^{\lceil \lg V \rceil} < 2V$를 사용할 수 있습니다.

이번에도 동적 계획법 풀이는 간단합니다. 알고리즘을 적어 보기도 전에 실행 시간이 $O(V^3 \log V)$임을 알 수 있습니다—u, v, x에 대해서는 각각 V개의 가능한 값을 고려해야 하지만, ℓ에 대해서는 $\lceil \lg V \rceil$개의 가능한 값만 고려하면 되기 때문입니다. 다음에 제시하는 Fischer와 Meyer의 알고리즘 의사코드에서 배열 항목 $dist[u, v, i]$는 $dist(u, v, 2^i)$의 값을 저장합니다.

```
FischerMeyerAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)
  for i ← 1 to ⌈lg V⌉                                   ⟪ℓ = 2^i⟫
      for all vertices u
          for all vertices v
              dist[u, v, i] ← ∞
              for all vertices x
                  if dist[u, v, i] > dist[u, x, i − 1] + dist[x, v, i − 1]
                      dist[u, v, i] ← dist[u, x, i − 1] + dist[x, v, i − 1]
```

앞서 다룬 알고리즘들과 달리, FischerMeyerAPSP는 어떤 단일 출발점 최단 경로 알고리즘을 V번 호출하는 것과도 같지 *않습니다*. 특히 가장 안쪽 루프는 단순히 긴장된 간선을 완화하는 것이 *아닙니다*. 그럼에도 벨먼-포드 알고리즘과 앞서 다룬 동적 계획법 알고리즘에서 했던 것처럼, 모든 곳에서 $dist[u, v, i]$ 대신 $dist[u, v]$를 사용하여 표의 마지막 차원을 제거할 수 있습니다. 이렇게 하면 공간이 $O(V^3)$에서 $O(V^2)$로 줄어듭니다. 이렇게 더 다듬어진 알고리즘은 1957년에 Leyzorek *et al.*이 데이크스트라 알고리즘을 설명한 바로 그 논문에서 기술하였습니다.

```
LeyzorekAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)
  for i ← 1 to ⌈lg V⌉                                  ⟪ℓ = 2^i⟫
      for all vertices u
          for all vertices v
              for all vertices x
                  if dist[u, v] > dist[u, x] + dist[x, v]
                      dist[u, v] ← dist[u, x] + dist[x, v]
```

9.7 기묘한 행렬 곱셈

방향 그래프에서 최단 경로를 계산하는 것과 정사각 행렬의 거듭제곱을 계산하는 것 사이에는 매우 밀접한 관련이 있습니다(이는 Shimbel이 처음 관찰했고, 이후 벨먼이 독립적으로 관찰했습니다). $n \times n$ 행렬 A를 제곱하는 다음 알고리즘을 FischerMeyerAPSP의 안쪽 루프와 비교해 보십시오. (유사성을 더 분명히 보여 주기 위해 두 번째 알고리즘의 표기법을 약간 수정했습니다.)

```
MatrixSquare(A):
  for i ← 1 to n
      for j ← 1 to n
          A'[i, j] ← 0
          for k ← 1 to n
              A'[i, j] ← A'[i, j] + A[i, k] · A[k, j]
```

```
FischerMeyerInnerLoop(D):
  for all vertices u
      for all vertices v
          D'[u, v] ← ∞
          for all vertices x
              D'[u, v] ← min{D'[u, v], D[u, x] + D[x, v]}
```

이 두 알고리즘의 *유일한* 차이점은 두 번째 알고리즘이 곱셈 대신 덧셈을, 덧셈 대신 최솟값 연산을 사용한다는 것입니다. 이러한 이유로 최단 경로 안쪽 루프는 때때로 "min-plus" 행렬 곱셈, "거리" 행렬 곱셈, 또는 "funny" 행렬 곱셈이라고 불립니다.

앞에서 본 더 느린 알고리즘 ShimbelAPSP는 가중치 행렬 w의 $(V-1)$번째 "min-plus 거듭제곱"을 계산하는 표준적인 반복 알고리즘입니다. 첫 번째 루프 묶음은 주대각선에 0이 있고 나머지 모든 곳에 ∞가 있는 min-plus 항등 행렬을 설정하며, 두 번째 주 루프는 반복할 때마다 다음 "min-plus 거듭제곱"을 계산합니다. FischerMeyerAPSP는 1장의 끝에서 보았던 것과 똑같이, 거듭제곱을 계산하는 이 반복적 방법을 반복 제곱법으로 대체합니다. 여기서도 다시 한번 고대 이집트 *ἀρπεδονάπται*의 영향을 볼 수 있습니다!

(표준) 행렬 곱셈에는 정수를 곱하는 카라추바 분할 정복 알고리즘과 비슷한, 더 빠른 분할 정복 알고리즘이 있습니다. 이러한 알고리즘 가운데 최초의 것은 1969년에

폴커 슈트라센(Volker Strassen)이 제시했으며, 두 $n \times n$ 행렬을 곱하는 문제를 두 $n/2 \times n/2$ 행렬을 곱하는 *일곱* 개의 인스턴스로 환원합니다. 슈트라센 알고리즘은 $O(n^{\lg 7}) = O(n^{2.807355})$ 시간에 실행됩니다. 슈트라센 알고리즘은 지난 50년 동안 여러 차례 개선되었으며, 2018년 기준으로 알려진 가장 빠른 행렬 곱셈 알고리즘은 $O(n^{2.372864})$ 시간에 실행됩니다.[6] 안타깝게도 이처럼 더 빠른 알고리즘은 모두 뺄셈을 사용하며, 뺄셈에 해당하는 "이상한" 연산은 존재하지 않습니다. (min의 역연산은 무엇이겠습니까?) 따라서 적어도 일반적인 그래프에서는 우리 동적 계획법 알고리즘의 안쪽 루프를 빠르게 만드는 뻔한 방법이 없습니다.

하지만 "명백하지 않다"는 것이 "불가능하다"는 뜻은 아닙니다! 실제로 모든 쌍 최단 경로 문제의 *특수한 경우*에 대해서는 훨씬 더 빠른 알고리즘이 여러 개 있습니다. 그중 가장 멋진 것 하나는 1991년에 Zvi Galil과 Oded Margalit이 발견하고 1992년에 Raimund Seidel이 더 단순화한 간단한 무작위 알고리즘입니다. 이 알고리즘은 *가중치 없는 무방향 그래프*에서 모든 쌍 최단 경로 *거리*를 *기대* 시간 $O(M(V)\log V)$에 계산합니다. 여기서 $M(n) = O(n^{2.372864})$는 두 $n \times n$ 정수 행렬을 (진지하게) 곱하는 데 필요한 시간입니다.[7] 이후 Galil, Margalit, Seidel의 접근법은 작은 정수 간선 가중치를 갖는 방향 그래프에서 실제 최단 경로를 결정론적으로, 강한 부분 세제곱 시간에 계산하도록 확장되었습니다.

반면, 작은 정수 가중치 환경에서는 상당한 진전이 있었음에도 불구하고, 더 일반적인 간선 가중치에 대해 모든 쌍 최단 경로를 $O(V^{2.999999})$ 시간에 계산하는 방법은 9를 몇 개 붙이든 아무도 알지 못합니다. 게다가 그러한 알고리즘이 불가능하다는 증거도 어느 정도 있습니다! 그러니 결국 "자명하지 않다"는 말이 정말로 "불가능하다"는 뜻일지도 모릅니다.

9.8 (클리니-로이-)플로이드-워셜(-잉거먼)

우리의 빠른 동적 계획법 알고리즘은 최악의 경우 여전히 존슨 알고리즘의 표준 구현보다 $O(\log V)$배 느립니다. 이 로그 인수를 없애는 최단 경로의 다른 정식화는 1962년에 두 번 제안되었습니다. 처음에는 로버트 플로이드(Robert Floyd)가, 나중에는 피터 잉거먼(Peter Ingerman)이 독립적으로 제안했으며, 두 사람 모두 같은 해 앞서 발표된 스티븐 워셜(Stephen Warshall)의 알고리즘을 약간 일반화했습니다. 사실 워셜의 알고리즘은 그 이전인 1959년에 버나드 로이(Bernard Roy)가 이미 발견했으며, 그 바탕에 있는 재귀 패턴은 1951년에 스티븐 클레이니(Stephen Kleene)[8]가 사용했습니다.

[6] 임의의 두 $n \times n$ 행렬을 곱하는 데 필요한 최소 시간을 결정하는 것은 오랫동안 풀리지 않은 문제입니다. 많은 사람은 임의의 $\varepsilon > 0$에 대해 $O(n^{2+\varepsilon})$ 시간에 실행되는, 어쩌면 $O(n^2)$ 시간에 실행되는 알고리즘이 아직 발견되지 않은 채 존재한다고 믿습니다.

[7] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400-403, 1995. 이 논문은 (적어도 1992년 학회 버전에서는) 알고리즘이 논문의 *초록 안에서* 완전히 설명되고 분석된 몇 안 되는 알고리즘 논문 중 하나입니다. 다음도 참고하십시오: Noga Alon, Zvi Galil, Oded Margalit*. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences* 54(2):255–262, 1997.

[8] 발음은 "clay knee"이며, "clean"이나 "clean-ee", "clay-nuh", "dimaggio"가 아닙니다. 구체적으로 클레이니는 모든 유한 오토마타에 동등한 정규 표현식이 존재한다는 귀납적 증명을 기술했습니다. 클레이니의 귀납법 패턴은 플로이드-워셜 점화식과 본질적으로 동일합니다.

Warshall(그리고 Roy와 Kleene)의 통찰은 동적 계획법 점화식에서 세 번째 매개변수를 다른 것으로 사용하는 것이었습니다. 이들은 간선 개수가 제한된 경로를 고려하는 대신, 특정 정점들만 거쳐 갈 수 있는 경로를 고려했습니다. 여기서 "거쳐 간다"는 것은 "들어갔다가 나온다"는 뜻입니다. 예를 들어 경로 $w \to x \to y \to z$는 w에서 시작하여 x와 y를 *거쳐 가고* z에서 끝납니다.

정점들에 1부터 V까지 임의로 번호를 매깁니다. 모든 정점 쌍 u와 v, 그리고 모든 정수 r에 대해 경로 $\pi(u, v, r)$를 다음과 같이 정의합니다.

> $\pi(u, v, r)$는 u에서 v로 가는 경로 중 번호가 r 이하인 정점만을 거치는 최단 경로(존재한다면)입니다.

특히 $\pi(u, v, V)$는 u에서 v로 가는 실제 최단 경로입니다. Kleene, Roy, Warshall은 모두 이 경로들이 단순한 재귀적 구조를 가진다는 사실을 관찰했습니다.

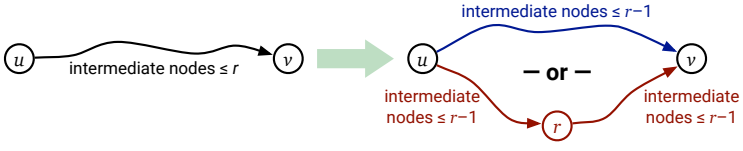


그림 9.3. 제한된 최단 경로 $\pi(u, v, r)$의 재귀적 구조입니다.

- 경로 $\pi(u, v, 0)$은 어떤 중간 정점도 지날 수 없으므로, u에서 v로 가는 간선(존재한다면)이어야 합니다.
- 임의의 정수 $r > 0$에 대해, $\pi(u, v, r)$는 정점 r을 지나거나 지나지 않습니다.
  - $\pi(u, v, r)$가 정점 r을 지난다면, 이 경로는 u에서 r까지 가는 부분 경로와 그 뒤에 이어지는 r에서 v까지 가는 부분 경로로 이루어집니다. 두 부분 경로는 모두 번호가 $r-1$ 이하인 정점만을 지나며, 더욱이 이 제약 아래에서 가능한 한 가장 짧습니다. 따라서 두 부분 경로는 반드시 $\pi(u, r, r-1)$과 $\pi(r, v, r-1)$이어야 합니다.
  - 반면에 $\pi(u, v, r)$가 정점 r을 지나지 않는다면, 이 경로는 번호가 $r-1$ 이하인 정점들만 지나며, 이러한 제약 아래에서 *최단* 경로여야 합니다. 따라서 이 경우에는 반드시 $\pi(u, v, r) = \pi(u, v, r-1)$이 성립합니다.

이제 $dist(u, v, r)$이 경로 $\pi(u, v, r)$의 *길이*를 나타낸다고 합시다. $\pi(u, v, r)$의 재귀적 구조로부터 다음 점화식이 곧바로 도출됩니다:

$$dist(u, v, r) = \begin{cases} w(u \to v) & \text{if } r = 0 \\ \min \left\{ \begin{array}{c} dist(u, v, r-1) \\ dist(u, r, r-1) + dist(r, v, r-1) \end{array} \right\} & \text{그렇지 않으면} \end{cases}$$

우리의 목표는 모든 정점 u와 v에 대해 $dist(u, v, V)$를 계산하는 것입니다. 이번에도 이 점화식은 간단한 동적 계획법 알고리즘으로 $O(V^3)$ *시간*에 계산할 수 있습니다.

```
KleeneAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)
  for r ← 1 to V
      for all vertices u
          for all vertices v
              if dist[u, v, r − 1] < dist[u, r, r − 1] + dist[r, v, r − 1]
                  dist[u, v, r] ← dist[u, v, r − 1]
              else
                  dist[u, v, r] ← dist[u, r, r − 1] + dist[r, v, r − 1]
```

최단 경로를 구하는 앞선 모든 동적 계획법 알고리즘과 마찬가지로, 메모이제이션 테이블의 세 번째 차원을 제거하여 KleeneAPSP를 단순화할 수 있습니다. 또한 정점 번호는 임의로 정한 것이므로 의사코드에서 이를 명시적으로 언급할 이유가 없습니다. 이로써 마침내 워셜(Warshall) 알고리즘을 플로이드(Floyd)가 개선한 알고리즘에 도달합니다:

```
FloydWarshall(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)
  for all vertices r
      for all vertices u
          for all vertices v
              if dist[u, v] > dist[u, r] + dist[r, v]
                  dist[u, v] ← dist[u, r] + dist[r, v]
```

FloydWarshall을 앞서 살펴본, 조금 더 느린 동적 계획법 알고리즘인 LeyzorekAPSP와 비교해 보면 흥미롭습니다. FloydWarshall은 모든 정점 삼중쌍을 $O(\log V)$번 훑는 대신 단 한 번만 훑으면 되는데, 이는 단지 세 반복문의 중첩 순서를 달리했기 때문일 뿐입니다!

연습문제

1. (a) 최단 경로 거리 배열뿐 아니라 선행자 포인터 배열도 함께 반환하면서 여전히 $O(V^3 \log V)$ 시간에 실행되도록 LeyzorekAPSP를 수정하는 방법을 설명하십시오.

   (b) FloydWarshall을 수정하여 최단 경로 거리 배열뿐만 아니라 선행자 포인터 배열도 반환하면서, 여전히 $O(V^3)$ 시간에 동작하도록 하는 방법을 설명하십시오.

2. 그래프에 음수 사이클이 있으면 이 장에서 다룬 알고리즘은 모두 실패합니다. 존슨 알고리즘은 초기화 단계에서 (벨먼-포드를 통해) 음수 사이클을 검출하고 중단하며, 동적 계획법 알고리즘은 그저 잘못된 결과를 반환합니다. 그러나 이 알고리

즘들은 모두 음수 사이클이 있는 경우에도 올바른 최단 경로 거리를 반환하도록 수정할 수 있습니다. 구체적으로, 모든 정점 u와 v에 대해 다음이 성립합니다:

- u에서 v에 도달할 수 없으면 알고리즘은 $dist[u, v] = \infty$를 반환해야 합니다.
- u에서 v에 도달할 수 있는 음수 사이클에 u가 도달할 수 있다면, 알고리즘은 $dist[u, v] = -\infty$를 반환해야 합니다.
- 그렇지 않다면 u에서 v로 가는 최단 경로가 존재하므로, 알고리즘은 그 길이를 반환해야 합니다.

(a) 그래프에 음수 사이클이 있더라도 올바른 최단 경로 거리를 반환하도록 존슨 알고리즘을 수정하는 방법을 설명하십시오.

(b) 그래프에 음수 사이클이 있더라도 올바른 최단 경로 거리를 반환하도록 LeyzorekAPSP를 수정하는 방법을 설명하십시오.

(c) 그래프에 음수 사이클이 있더라도 올바른 최단 경로 거리를 반환하도록 플로이드-워셜을 수정하는 방법을 설명하십시오.

3. 이 장에서 설명한 알고리즘은 G에 음수 사이클이 있는지 여부만 보고하는 대신, 음수 사이클이 존재하면 입력 그래프 G에 있는 음수 사이클 하나를 명시적으로 기술하여 반환하도록 수정할 수도 있습니다.

(a) 모든 최단 경로 거리의 배열 또는 음수 사이클 중 하나를 반환하도록 존슨 알고리즘을 수정하는 방법을 설명하십시오.

(b) LeyzorekAPSP가 모든 최단 경로 거리의 배열 또는 음수 사이클 중 하나를 반환하도록 수정하는 방법을 설명하십시오.

(c) 플로이드-워셜을 수정하여 모든 최단 경로 거리의 배열 또는 음수 사이클 중 하나를 반환하도록 하는 방법을 설명하십시오.

어떤 경우에도 입력 그래프에 음수 사이클이 둘 이상 있으면, 여러분의 알고리즘은 그중 하나를 임의로 선택해도 됩니다.

4. $G = (V, E)$를 간선에 가중치가 있는 방향 그래프라고 합시다. 간선 가중치는 양수, 음수, 0일 수 있지만, 음수 사이클은 없습니다.

(a) G에서 길이가 0인 사이클을 찾거나, 그러한 사이클이 존재하지 않음을 올바르게 보고하는 효율적인 알고리즘을 설명하십시오.

(b) 다음 성질을 만족하는 G의 부분 그래프 H를 구성하는 효율적인 알고리즘을 설명하십시오.

- G의 모든 정점은 H의 정점입니다.
- H의 모든 방향 사이클은 길이가 0입니다.
- G에 있는 길이 0인 모든 방향 사이클은 H에서도 사이클입니다.

특히, G에 영 사이클이 없으면 H에는 간선이 없습니다.

5. $G = (V, E)$를 간선에 가중치가 있는 방향 그래프라고 합시다. 간선 가중치는 양수, 음수, 0 중 어느 것이든 될 수 있습니다. G의 정점들이 k개의 서로소인 부분집합 $V_1, V_2, \ldots, V_k$로 분할되어 있다고 가정합시다. 즉, G의 모든 정점은 정확히 하나의 부분집합 V_i에 속합니다. 각 i와 j에 대해, $\delta(i, j)$를 V_i의 정점과 V_j의 정점 사이의 최단 경로 거리 중 최솟값이라고 합시다:

$$\delta(i, j) = \min\{dist(v_i, v_j) \mid v_i \in V_i \text{그리고} v_j \in V_j\}.$$

   모든 i와 j에 대해 $\delta(i, j)$를 계산하는 알고리즘을 설명하십시오. 만점을 받으려면 알고리즘이 $O(VE + kV \log V)$ 시간에 실행되어야 합니다.

6. 이 문제에서는 여러분, 그렇습니다 바로 ***여러분***이 어떻게 월스트리트에 고용되어 거대한 경제 붕괴를 일으킬 수 있는지 알아보겠습니다! *차익 거래* 사업은 통화 환율의 차이를 이용해 돈을 버는 수법입니다. 구체적으로, 1 미국 달러로 120 일본 엔을 살 수 있고, 1 엔으로 0.01 유로를 살 수 있으며, 1 유로로 1.2 미국 달러를 살 수 있다고 가정해 봅시다. 그러면 $1로 시작하는 거래자는 돈을 달러에서 엔으로, 다시 엔에서 유로로, 마지막으로 유로에서 다시 달러로 바꾸어 $1.44를 손에 쥘 수 있습니다! 통화의 사이클 $\$ \to ¥ \to € \to \$$를 ***차익 거래 사이클***이라고 합니다. 물론 가격이 바로잡히기 전에 차익 거래 사이클을 찾아내어 이용하려면 매우 빠른 알고리즘이 필요합니다.

   여러분의 외환 시장에서 n개의 서로 다른 통화가 거래된다고 가정합시다. 모든 통화 쌍 사이의 환율을 담은 행렬 $Exch[1..n, 1..n]$이 주어지며, 각 i와 j에 대해 통화 i 한 단위를 통화 j $Exch[i, j]$단위로 교환할 수 있습니다. ($Exch[i, j] \cdot Exch[j, i] = 1$이라고 가정해서는 *안 됩니다.*)

   (a) 차익 거래 사이클이 없다고 가정하고, 통화 1 한 단위로 시작하여 거래를 통해 얻을 수 있는 통화 i의 최대량이 $MaxAmt[i]$인 배열 $MaxAmt[1..n]$을 반환하는 알고리즘을 설명하십시오.

   (b) 주어진 통화 환율 행렬에 차익 거래 사이클이 존재하는지 판별하는 알고리즘을 설명하십시오.

   (c) (b)에서 만든 알고리즘을 수정하여, 차익 거래 사이클이 존재하는 경우 실제로 그 사이클을 반환하도록 하십시오.

7. 모티는 클랙스파이어 미궁에서 안정화된 플럼버스를 가져와야 합니다. 모티는 릭의 차원 간 포털 건을 사용해 미궁에 들어간 다음, 미궁을 지나 플럼버스가 있는 곳까지 가야 합니다. 그 후 그 플럼버스를 안정화하기 위해 미궁을 지나 플리브가 있는 곳까지 가져가야 하며, 마지막으로 안정화된 플럼버스를 처음의 포털까지 다시 가져가 집으로 돌아가야 합니다. 플럼버스는 플리브 즙으로 안정화되는데, 어떤 플리브든 자신의 플리브 구멍에서 꺼내지는 즉시 이 즙을 내놓습니다. 안정화되지 않은 플럼버스는 원래의 보관 장치에서 137플링크보다 멀리 옮겨지면 폭발합니다. 클랙스파이어 미궁에서는 방귀 냄새가 나기 때문에 모티는 그곳에서 보내는 시간을 최대한 줄이고 싶어 합니다.

릭은 모티에게 클랙스파이어 미궁의 상세한 지도를 건넸습니다. 이 지도는 간선 가중치(플링크 단위의 거리를 나타냄)가 음이 아닌(즉, 음수 간선이 없는) 방향 그래프 $G = (V, E)$와 서로소인 두 부분집합 $P \subset V$ 및 $F \subset V$로 이루어져 있으며, 이 두 부분집합은 각각 플럼버스 보관소와 플리브홀을 나타냅니다. 모티는 p에서 f까지의 최단 경로 거리가 최대 137플링크이고 최단 보행 $s \rightsquigarrow p \rightsquigarrow f \rightsquigarrow s$의 길이가 가능한 한 짧아지도록 시작 정점 s, 플럼버스 보관소 $p \in P$, 플리브홀 $f \in F$를 찾아야 합니다.

모티의 문제를 해(꺼억)결하는 알고(꺼억)리즘을 설명하고 분석하십시오. 모티가 실제로 성공할 수 있다고 가정해도 됩니다.

8. $G = (V, E)$를 간선에 가중치가 있는 방향 그래프라고 합시다. 간선의 가중치는 양수, 음수 또는 0일 수 있습니다.

   (a) 이 그래프에서 임의의 정점 v를 삭제하되, 다른 어떤 정점 쌍 사이의 최단 경로 거리도 바뀌지 않게 하려면 어떻게 해야 하겠습니까? 가중치가 있는 간선을 갖는 방향 그래프 $G' = (V \setminus \{v\}, E')$를 $O(V^2)$ 시간에 구성하는 알고리즘을 설명하십시오. 이때 G'의 임의의 두 정점 사이의 최단 경로 거리는 G에서 같은 두 정점 사이의 최단 경로 거리와 같아야 합니다.

   (b) 이제 G'에서의 모든 최단 경로 거리를 이미 계산해 두었다고 가정합시다. 원래 그래프 G에서 v로부터 다른 모든 정점까지의 최단 경로 거리와 다른 모든 정점으로부터 v까지의 최단 경로 거리를 모두 $O(V^2)$ 시간에 계산하는 알고리즘을 설명하십시오.

   (c) (a)와 (b)를 결합하여 $O(V^3)$ 시간에 실행되는 또 다른 모든 쌍 최단 경로 알고리즘을 만드십시오. (그 결과로 얻는 알고리즘은 플로이드-워셜과 *거의* 같습니다!)

9. A와 B가 *불리언* $n \times n$ 행렬이라고 가정합시다. A와 B의 *불리언* 곱 또는 *and-or* 곱은 다음과 같이 정의되는 $n \times n$ 행렬 C입니다:

$$C[i, j] := \bigvee_k \big(A[i, k] \wedge B[k, j]\big)$$

   (a) 불리언 행렬 곱셈을 min-plus 행렬 곱셈으로 환원하십시오. 즉, 두 $n \times n$ 행렬의 min-plus 곱을 $T(n)$ 시간에 계산하는 서브루틴 MinPlusMultiply가 주어졌을 때, 두 불리언 행렬을 $O(T(n))$ 시간에 곱하는 알고리즘 BooleanMatrixMultiply를 설명하고 분석하십시오.

   (b) 불리언 행렬 곱셈을 표준 행렬 곱셈으로 환원하십시오. 즉, 두 $n \times n$ 행렬의 표준 곱을 $T(n)$ 시간에 계산하는 서브루틴 MatrixMultiply가 주어졌을 때, 두 불리언 행렬을 $O(T(n))$ 시간에 곱하는 알고리즘 BooleanMatrixMultiply를 설명하고 분석하십시오.

10. 방향 그래프 G의 *추이적 폐포*는 G에 u에서 v로 가는 방향 경로가 있을 때, 그리고 오직 그때에만 간선 $u \rightarrow v$를 포함합니다. 이 문제에서는 어떤 상수 $2 \le \omega < 3$에

대해 두 $n \times n$ 불리언 행렬을 $O(n^\omega)$ 시간에 곱할 수 있다고 가정합니다. (문제 9(b)로부터 $\omega \le 2.372864$임을 알 수 있습니다.)

(a) n개의 정점을 가진 방향 그래프의 추이적 폐포를 $O(n^\omega \log n)$ 시간에 계산하는 알고리즘을 설명하십시오.

(b) 이제 G가 방향 *비순환* 그래프라고 가정합시다. G의 추이적 폐포를 $O(n^\omega)$ 시간에 계산하는 알고리즘을 설명하십시오. *[힌트: dag를 다룰 때 늘 하던 일을 먼저 한 다음, 분할 정복을 적용하십시오. $\omega \ge 2$라는 사실을 이용하십시오.]*

(c) 마지막으로, *임의의* 방향 그래프의 추이적 폐포를 $O(n^\omega)$ 시간에 계산하는 알고리즘을 설명하십시오. *[힌트: 임의의 방향 그래프를 dag로 바꿀 때 늘 하던 방법을 그대로 쓰십시오.]*

(d) 이제 앞의 환원을 거꾸로 뒤집어 봅시다. 어떤 상수 $2 \le \alpha < 3$에 대해 정점이 n개인 방향 그래프의 추이적 폐포를 $O(n^\alpha)$ 시간에 계산하는 서브루틴 TransitiveClosure가 주어졌을 때, $O(n^\alpha)$ 시간에 실행되는 불리언 행렬 곱셈 알고리즘을 기술하고 분석하십시오.

11. 다음 재귀 알고리즘이 모든 쌍 최단 경로 거리를 $O(n^3)$ 시간에 올바르게 계산함을 증명하십시오. 편의상 n이 2의 거듭제곱이라고 가정해도 됩니다. 평소와 같이 배열 D는 보조 함수 RecAPSP에 *참조로* 전달됩니다. *[힌트: 이것은 플로이드-워셜을 뒤섞은 버전으로, 캐시 동작이 훨씬 더 우수합니다.[9]]*

```
RecursiveAPSP(V, E, w):
  n ← |V|
  for i ← 1 to n
      for j ← 1 to n
          if i = j
              D[i,j] ← 0
          if i→j ∈ E
              D[i,j] ← w(i→j)
          else
              D[i,j] ← ∞
  RecAPSP(D, n, 1, 1, 1)
  return D[1..n, 1..n]
```

```
RecAPSP(D, n, i, j, k):
  if n = 1
      D[i,j] ← min{D[i,j], D[i,k] + D[j,k]}
  else
      m ← n/2
      RecAPSP(D, n/2, i,   j,   k)
      RecAPSP(D, n/2, i,   j,   k+m)
      RecAPSP(D, n/2, i,   j+m, k)
      RecAPSP(D, n/2, i,   j+m, k+m)
      RecAPSP(D, n/2, i+m, j,   k)
      RecAPSP(D, n/2, i+m, j,   k+m)
      RecAPSP(D, n/2, i+m, j+m, k)
      RecAPSP(D, n/2, i+m, j+m, k+m)
```

♥12. $G = (V, E)$를 인접 행렬 $A[1..n, 1..n]$로 표현되는, 방향이 없고 가중치가 없으며 연결된, 정점이 n개인 그래프라고 합시다. 이 문제에서는 빠른 행렬 곱셈을

[9] Joon-Sang Park, Michael Penner, Viktor K. Prasanna. 캐시 성능 향상을 위한 그래프 알고리즘 최적화. *IEEE Trans. Parallel and Distributed Systems* 15(9):769–782, 2004. 더 넓은 부류의 동적 계획법 문제로 크게 일반화한 결과는 Rezaul Alam Chowdhury와 Vijaya Ramachandran의 논문을 참고하십시오. 캐시 비인지 동적 계획법. *Proc. 17th SODA* 591–600, 2006.

사용하여 G의 최단 경로 거리로 이루어진 $n \times n$ 행렬 $D[1..n, 1..n]$을 계산하는 Seidel의 삼차 미만(sub-cubic) 알고리즘을 유도하겠습니다. 어떤 미지의 상수 $\omega \geq 2$에 대해, 두 $n \times n$ 행렬의 표준 곱을 $O(n^\omega)$ 시간에 계산하는 서브루틴 MatrixMultiply가 있다고 가정합니다.

(a) G^2를 G와 같은 정점들을 가지며, 두 정점이 G에서 길이가 최대 2인 경로로 연결되어 있을 때, 그리고 그럴 때에만 간선으로 연결되는 그래프라고 합시다. MatrixMultiply를 한 번만 호출하고 $O(n^2)$의 추가 시간을 사용하여 G^2의 인접 행렬을 계산하는 알고리즘을 설명하십시오.

(b) G^2가 완전 그래프라는 사실을 알게 되었다고 가정합시다. $O(n^2)$의 추가 시간 안에 G의 최단 경로 거리 행렬 D를 계산하는 알고리즘을 설명하십시오.

(c) G^2에서의 최단 경로 거리 행렬 D^2를 재귀적으로 계산한다고 가정합시다. G에서 노드 i로부터 노드 j까지의 최단 경로 거리가 $2 \cdot D^2[i,j]$ 또는 $2 \cdot D^2[i,j] - 1$임을 증명하십시오.

(d) 이제 G^2가 완전 그래프가 *아니라고* 가정합시다. $X = D^2 \cdot A$라 하고, $\deg(i)$는 원래 그래프 G에서 정점 i의 차수를 나타낸다고 합시다. G에서 노드 i부터 노드 j까지의 최단 경로 거리가 $2 \cdot D^2[i,j]$일 필요충분조건은 $X[i,j] \geq D^2[i,j] \cdot \deg(i)$임을 증명하십시오.

(e) G의 최단 경로 거리 행렬 D를 $O(n^\omega \log n)$ 시간에 계산하는 알고리즘을 설명하십시오.

13. Gideon Yuval은 1976년에 최소-덧셈(min-plus) 행렬 곱셈을 표준 행렬 곱셈으로 바꾸는 다음 환원을 제안했습니다. 모든 원소가 0 이상 M 이하의 정수인 두 $n \times n$ 정수 행렬 A와 B가 주어졌고, 이들의 최소-덧셈 곱 행렬 C를 계산하려 한다고 가정합시다. 여기서 C는 다음과 같이 정의합니다.

$$C[i,k] = \min_j (A[i,j] + B[j,k])$$

모든 인덱스 i와 k에 대해 성립합니다. 새로운 $n \times n$ 행렬 두 개 A'과 B'을 다음과 같이 정의합니다.

$$A'[i,j] = n^{M - A[i,j]} \qquad \text{그리고} \qquad B'[i,j] = n^{M - B[i,j]}.$$

마지막으로, $C'[i,k] = \sum_j A'[i,j] \cdot B'[j,k]$로 정의되는 A'와 B'의 (표준) 곱을 C'라고 합시다.

(a) 표준 정수 산술 연산($+$, $-$, $\times$)만을 사용하여 A로부터 A'을 구성하는 알고리즘을 설명하십시오.

(b) 표준 정수 산술 연산($+$, $-$, $\times$)만 사용하여 C'에서 min-plus 곱 C를 추출하는 알고리즘을 설명하십시오.[10]

[10] 특히 로그나 나눗셈, 바닥 함수 $\lfloor x \rfloor$는 *절대* 사용하지 마십시오. 제 말을 믿으십시오—이것은 여러분이 결코 건드리고 싶지 않을 골칫거리입니다.

(c) 어떤 상수 $2 \le \omega < 3$에 대해 두 $n \times n$ 정수 행렬의 표준 곱을 $O(n^{\omega})$번의 산술 연산으로 계산할 수 있다고 가정합시다. 유발(Yuval)의 알고리즘이 최소-합(min-plus) 곱 C를 계산하려면 산술 연산이 몇 번 필요합니까?

(d) $n \times n$ 정수 행렬 A 하나가 주어졌을 때, Yuval의 알고리즘을 사용하여 A의 "이상한" n제곱을 계산하려면 몇 번의 산술 연산이 필요합니까? (A가 그래프의 가중치 인접 행렬이면 A의 "이상한" n제곱은 최단 경로 거리의 행렬이라는 점을 상기하십시오.)

(e) Yuval의 알고리즘이 *임의의* 간선 가중치에 대해 플로이드-워셜보다 빠른 모든 쌍 최단 경로 알고리즘을 의미하지 않는 이유는 무엇입니까? 우리는 어떻게 속이고 있는 것입니까?

과정을 멈추어서는 그 과정을 이해할 수 없습니다. 이해는 과정의 흐름과 함께 움직여야 하며, 그 흐름에 합류하여 함께 흘러가야 합니다.

— 프랭크 허버트의 *듄*(1965)에 나오는 멘타트 제1법칙

예상과 달리 몰입은 대개 여가와 오락을 즐기며 느긋하게 쉬는 순간이 아니라, 어려운 일에 적극적으로 뛰어들어 우리의 정신적·신체적 능력을 한계까지 끌어올리는 과제에 몰두하고 있을 때 일어납니다.... 몰입은 노력 없이 이루기 어렵습니다. 몰입은 "시간 낭비"가 아닙니다.

— Mihaly Csíkszentmihályi, *Flow: The Psychology of Optimal Experience* (1990)

경로를 아는 것과 그 경로를 걷는 것은 다릅니다.

— 모피어스 [로런스 피시번], *매트릭스* (1999)

10

최대 흐름 & 최소 컷

1950년대 중반, 미 공군 연구원 Theodore E. Harris와 퇴역한 미 육군 장군 Frank S. Ross는 소련과 동유럽 위성국들을 연결하는 철도망을 연구한 기밀 보고서를 작성했습니다. 이 철도망은 지리적 지역을 나타내는 44개의 정점과 철도망에서 그 지역들 사이의 연결을 나타내는 105개의 간선으로 이루어진 그래프로 모델링되었습니다. 각 간선에는 한 지역에서 다음 지역으로 물자를 수송할 수 있는 속도를 나타내는 가중치가 주어졌습니다. 그들은 사실상 시행착오를 통해 러시아에서 유럽으로 옮길 수 있는 물자의 최대량과, 연결을 제거하여(덜 추상적으로 말하면 철로를 폭파하여) 철도망을 가장 적은 비용으로 마비시키는 방법을 모두 알아냈으며, 후자를 "병목(the bottleneck)"이라고 불렀습니다. 그림 10.1의 철도망 그림이 실린 그들의 보고서는 1999년에야 기밀 해제되었습니다.[1]

이것은 ***최대 흐름*** 문제와 ***최소 컷*** 문제가 처음으로 적용된 것으로 기록된 사례 중 하나입니다. 두 문제 모두 입력은 방향 그래프 $G = (V, E)$이며, 이 그래프에는 각각 *소스*와 *대상*이라고 부르는 두 개의 특별한 정점 s와 t가 있습니다. 앞 장들에서와 마찬가지로, 정점 u에서 정점 v로 가는 방향 간선을 $u \to v$로 표기하겠습니다. 직관적으로 말하면, 최대 흐름 문제는 어떤 자원을 s에서 t로 옮길 수 있는 최대 속도를 구하는 문제이고, 최소 컷 문제는 s를 t로부터 분리하는 데 필요한 최소 피해를 구하는

[1] 저는 이 이야기를 Alexander Schrijver의 흥미로운 조사 논문 "On the history of combinatorial optimization (till 1960)"(조합 최적화의 역사(1960년까지))에서 알게 되었습니다. Harris-Ross 보고서는 Schrijver의 요청으로 기밀 해제되었습니다. 포드와 풀커슨(곧 만나게 될 것입니다)은 최대 흐름 문제를 정식화한 공로를 Harris에게 돌리지만, 정확한 연대는 다소 불분명합니다. Harris와 Ross는 George Dantzig에게 "문제를 정식화하는 데 도움을 준 것에 대해" 감사를 표합니다.

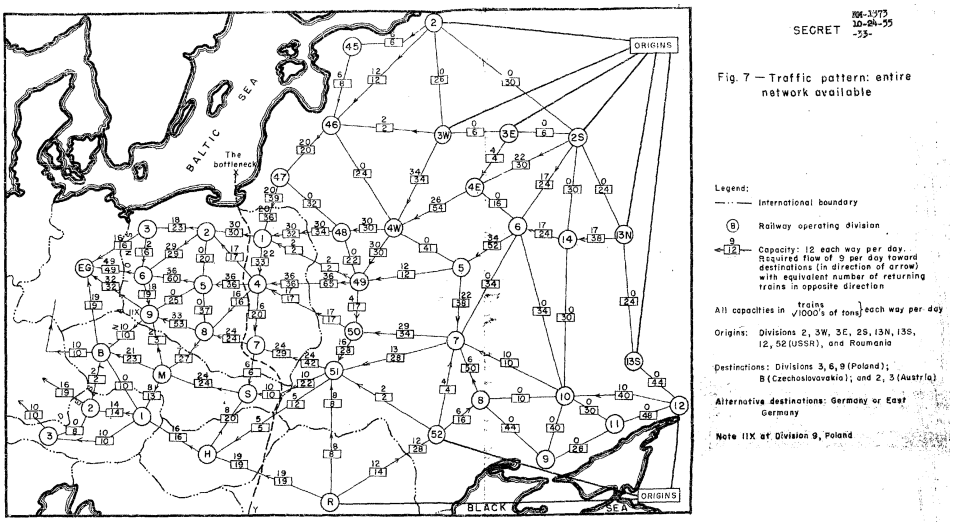


그림 10.1. 해리스와 로스가 작성한 바르샤바 조약기구 철도망 지도입니다. (책 끝의 이미지 출처를 참고하십시오.)

문제입니다.

10.1 흐름

(s,t)-흐름(또는 소스와 대상 정점이 문맥상 명확하다면 그냥 **흐름**)은 s와 t를 제외할 수도 있는 모든 정점 v에서 다음의 ***보존 제약 조건***을 만족하는 함수 $f : E \to \mathbb{R}$입니다:

$$\sum_u f(u{\to}v) = \sum_w f(v{\to}w).$$

말로 풀어 쓰면, v로 들어오는 흐름의 총량은 v에서 나가는 흐름의 총량과 같습니다. 표기를 간단히 하기 위해, 그래프에 간선 $u{\to}v$가 없으면 $f(u{\to}v) = 0$으로 정의합니다.

흐름 f의 ***값***은 $|f|$로 표기하며, 소스 정점 s에서 나가는 순 흐름의 총량입니다:

$$|f| := \sum_w f(s{\to}w) - \sum_u f(u{\to}s).$$

다음과 같이 $|f|$가 대상 정점 t로 들어오는 총 순 흐름과도 같다는 것을 증명하는 것은 어렵지 않습니다. 표기를 간단히 하기 위해, $\partial f(v)$가 임의의 정점 v에서 나가는 총 순 흐름을 나타낸다고 합시다:

$$\partial f(v) := \sum_u f(u{\to}v) - \sum_w f(v{\to}w).$$

보존 제약 조건에 따르면 s와 t를 제외한 모든 정점 v에 대해 $\partial f(v) = 0$이므로,

$$\sum_v \partial f(v) = \partial f(s) + \partial f(t).$$

반면에 한 정점을 떠나는 흐름은 모두 다른 정점으로 들어가야 하므로 $\sum_v \partial f(v) = 0$이어야 합니다. 이로부터 곧바로 $|f| = \partial f(s) = -\partial f(t)$가 성립합니다.

이제 각 간선 e에 음이 아닌 ***용량*** $c(e)$를 할당하는 또 다른 함수 $c : E \to \mathbb{R}_{\geq 0}$이 있다고 가정해 봅시다. 모든 간선 e에 대해 $0 \leq f(e) \leq c(e)$이면 흐름 f가 (c에 대해) ***실현 가능하다***고 합니다. 대부분의 경우 우리는 어떤 고정된 용량 함수 c에 대해 실현 가능한 흐름만을 고려합니다. $f(e) = c(e)$이면 흐름 f가 간선 e를 ***포화시킨다***고 하고, $f(e) = 0$이면 간선 e를 ***회피한다***고 합니다. ***최대 흐름 문제***는 용량 함수가 주어진 방향 그래프에서 값이 가능한 한 큰 실현 가능한 (s,t)-흐름을 계산하는 문제입니다.

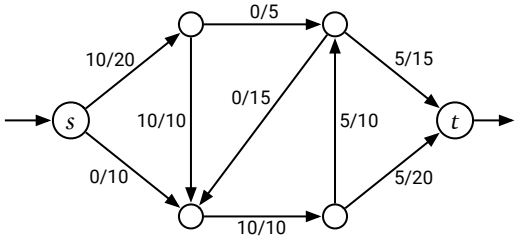


그림 10.2. 값이 10인 실현 가능한 (s,t)-흐름입니다. 각 간선에는 흐름/용량이 표시되어 있습니다.

10.2 컷

(s,t)-컷(문맥상 소스 정점과 대상 정점이 분명한 경우에는 간단히 ***컷***)은 정점들을 서로소인 부분집합 S와 T로 분할한 것으로, 여기서 $s \in S$이고 $t \in T$입니다. 서로소인 부분집합으로 분할한다는 것은 $S \cup T = V$이고 $S \cap T = \varnothing$이라는 뜻입니다.

용량 함수 $c : E \to \mathbb{R}_{\geq 0}$가 주어졌을 때, 컷의 ***용량***은 S에서 시작하여 T에서 끝나는 간선들의 용량의 합입니다:

$$\|S,T\| := \sum_{v \in S} \sum_{w \in T} c(v \to w).$$

(앞에서와 마찬가지로, $v \to w$가 그래프의 간선이 아니면 $c(v \to w) = 0$이라고 가정합니다.) 이 정의가 비대칭적이라는 점에 주목하십시오. T에서 시작하여 S에서 끝나는 간선은 중요하지 않습니다. ***최소 컷 문제***는 용량이 가능한 한 작은 (s,t)-컷을 계산하는 문제입니다.

직관적으로 말하면, 최소 컷은 s에서 t로 가는 모든 흐름을 가장 적은 비용으로 차단하는 방법입니다. 실제로 흐름과 컷 사이에 다음과 같은 관계가 성립함을 보이는 것은 어렵지 않습니다.

보조정리 10.1. *f를 **임의의** 실현 가능한 (s,t)-흐름이라 하고, (S,T)를 **임의의** (s,t)-컷이라 합시다. f의 값은 (S,T)의 용량 이하입니다. 또한 $|f| = \|S,T\|$일 필요충분조건은 f가 S에서 T로 가는 모든 간선을 포화시키고 T에서 S로 가는 모든 간선을 피하는 것입니다.*

증명: 마음에 드는 흐름 f와 마음에 드는 컷 (S,T)를 아무거나 하나씩 고른 다음, 통통 튀어 가는 부등식들을 따라가 보십시오:

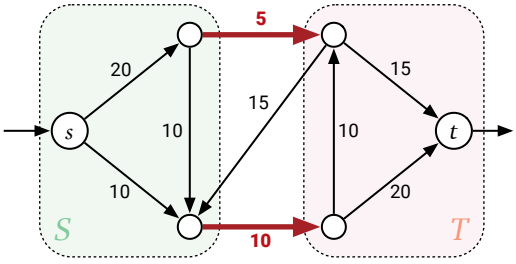


그림 10.3. 용량이 15인 (s,t)-컷입니다. 각 간선에는 그 간선의 용량이 표시되어 있습니다.

$$
\begin{aligned}
|f| &= \partial f(s) && \text{[정의에 의해]}\\
&= \sum_{v\in S} \partial f(v) && \text{[보존 제약 조건]}\\
&= \sum_{v\in S}\sum_{w} f(v\to w) \;-\; \sum_{v\in S}\sum_{u} f(u\to v) && \text{[계산, } \partial\text{의 정의]}\\
&= \sum_{v\in S}\sum_{w\notin S} f(v\to w) \;-\; \sum_{v\in S}\sum_{u\notin S} f(u\to v) && \text{[}S\text{에서 }S\text{로 가는 간선 제거]}\\
&= \sum_{v\in S}\sum_{w\in T} f(v\to w) \;-\; \sum_{v\in S}\sum_{u\in T} f(u\to v) && \text{[컷의 정의]}\\
&\le \sum_{v\in S}\sum_{w\in T} f(v\to w) && \text{[}f(u\to v)\ge 0\text{이므로]}\\
&\le \sum_{v\in S}\sum_{w\in T} c(v\to w) && \text{[}f(v\to w)\le c(v\to w)\text{이므로]}\\
&= \|S,T\| && \text{[정의에 의해]}
\end{aligned}
$$

두 번째 단계에서는 모든 정점 $v \in S \setminus \{s\}$에 대해 $\partial f(v) = 0$이므로 0을 더하는 것에 불과합니다. 네 번째 단계에서는 x와 y가 모두 S에 속하는 흐름 값 $f(x\to y)$를 제거합니다. 이 값들은 두 합 모두에 나타나기 때문입니다. $v = x$이고 $w = y$일 때는 양의 부호로, $v = y$이고 $u = x$일 때는 음의 부호로 나타납니다.

이 유도 과정의 첫 번째 부등식은 f가 T에서 S로 가는 모든 간선을 피할 때, 그리고 오직 그때에만 실제로 등식이 됩니다. 마찬가지로, 두 번째 부등식은 f가 S에서 T로 가는 모든 간선을 포화시킬 때, 그리고 오직 그때에만 실제로 등식이 됩니다. □

이 보조정리로부터 $|f| = \|S,T\|$이면 f는 반드시 최대 흐름이어야 하고 (S,T)는 반드시 최소 컷이어야 한다는 사실이 곧바로 도출됩니다.

10.3 최대 흐름-최소 컷 정리

놀랍게도, 모든 흐름 네트워크에는 $|f| = \|S,T\|$를 만족하는 실현 가능한 (s,t)-흐름 f와 (s,t)-컷 (S,T)가 존재합니다. 이것이 바로 유명한 *최대 흐름-최소 컷 정리*로, 1954년에 (최단 경로로 유명한) 레스터 포드와 델버트 풀커슨이 처음 증명했고, 1956년에 피터 일라이어스, 아미엘 파인스타인, 그리고 (정보 이론과 미로 찾기 로봇으로 유명한) 클로드 섀넌이 독립적으로 증명했습니다.

최대 흐름-최소 컷 정리. *소스가 s이고 대상이 t인 모든 흐름 네트워크에서 최대 (s,t)-흐름의 값은 최소 (s,t)-컷의 용량과 같습니다.*

포드와 풀커슨은 이 정리를 다음과 같이 증명했습니다. 그래프G, 정점s와 t, 그리고 용량 함수 $c : E \to \mathbb{R}_{\geq 0}$가 주어집니다. G가 ***환원되어*** 있다고, 즉 임의의 두 정점 u와 v 사이에 간선이 최대 하나만 있다고 가정하면 증명이 더 쉬워집니다. 특히 $c(u \to v) = 0$이거나 $c(v \to u) = 0$입니다. 이 가정은 쉽게 만족시킬 수 있습니다. G의 각 간선 $u \to v$를 새 정점 x로 세분하여 $u \to v$를 경로 $u \to x \to v$로 대체하고, $c(u \to x) = c(x \to v) = c(u \to v)$로 정의하면 됩니다. 이렇게 수정된 그래프는 원래 그래프와 같은 최대 흐름 값과 최소 컷 용량을 가집니다.

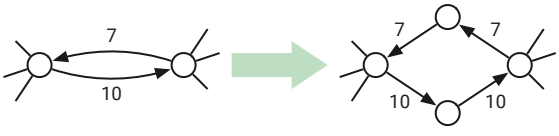


그림 10.4. 단방향 가정 강제하기

f를 G의 임의의 실행 가능한 (s,t)-흐름이라고 합시다. ***잔여 용량***이라고 부르는 새로운 용량 함수 $c_f : V \times V \to \mathbb{R}$을 다음과 같이 정의합니다:

$$c_f(u \to v) = \begin{cases} c(u \to v) - f(u \to v) & u \to v \in E\text{이면} \\ f(v \to u) & v \to u \in E\text{인 경우} \\ 0 & \text{그렇지 않으면} \end{cases}$$

직관적으로, 간선의 잔여 용량은 그 간선을 통해 흐름을 *얼마나 더* 보낼 수 있는지를 나타냅니다. $f \geq 0$이고 $f \leq c$이므로 이러한 잔여 용량은 항상 음이 아닙니다. $u \to v$가 원래 그래프 G의 간선이 아니더라도 $c_f(u \to v) > 0$일 수 있습니다. 따라서 ***잔여 그래프*** $G_f = (V, E_f)$를 정의합니다. 여기서 E_f는 잔여 용량이 양수인 간선들의 집합입니다. 대부분의 잔여 그래프는 축약되어 있지 *않습니다*. 특히 $0 < f(u \to v) < c(u \to v)$이면 잔여 그래프 G_f는 $u \to v$와 그 역방향 간선 $v \to u$를 모두 포함합니다.

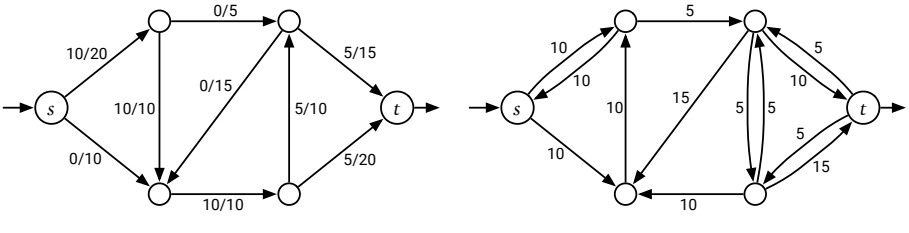


그림 10.5. 가중 그래프 G에서의 흐름 f와 이에 대응하는 잔여 그래프 G_f.

이제 고려할 경우는 두 가지입니다. 잔여 그래프 G^f에 소스 정점 s에서 대상 정점 t로 가는 방향 경로가 존재하거나, 존재하지 않습니다.

먼저 잔여 그래프 G_f에 s에서 t로 가는 방향 경로 P가 있다고 가정합시다. 이러한 P를 ***증가 경로***라고 부릅니다. P를 통해 보낼 수 있는 최대 흐름의 양을 $F =$

$\min_{u \to v \in P} c_f(u \to v)$라고 합시다. (원래 그래프에서) 새로운 흐름 $f' : E \to \mathbb{R}$을 다음과 같이 정의합니다.

$$f'(u \to v) = \begin{cases} f(u \to v) + F & u \to v \in P\text{이면} \\ f(u \to v) - F & v \to u \in P\text{이면} \\ f(u \to v) & \text{그렇지 않으면} \end{cases}$$

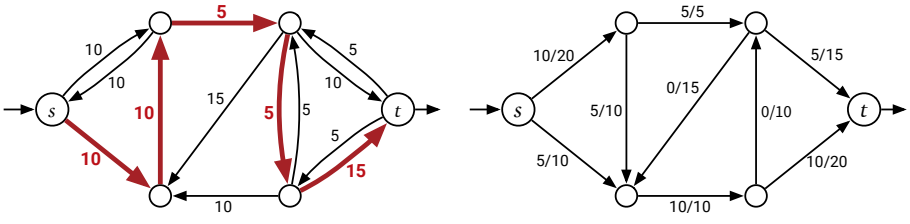


그림 10.6. 값이 $F = 5$인 증가 경로와 그 결과로 얻은 증가된 흐름 f'입니다.

이 새로운 흐름 f'이 원래 용량 c에 대해 실현 가능하다고, 즉 모든 곳에서 $f' \geq 0$이고 $f' \leq c$라고 주장합니다. 원래 그래프 G의 간선 $u \to v$를 생각해 봅시다. 고려해야 할 경우는 세 가지입니다.

- 증가 경로 P가 $u \to v$를 포함하면,

$$f'(u \to v) \;=\; f(u \to v) + F \;>\; f(u \to v) \;\geq\; 0$$

f가 실현 가능하기 때문이며, 또한

$$\begin{aligned} f'(u \to v) &= f(u \to v) + F && f'\text{의 정의에 의해} \\ &\leq f(u \to v) + c_f(u \to v) && F\text{의 정의에 의해} \\ &= f(u \to v) + c(u \to v) - f(u \to v) && c_f\text{의 정의에 의해} \\ &= c(u \to v) && \text{당연합니다.} \end{aligned}$$

- 증가 경로 P가 역방향 간선 $v \to u$를 포함한다면,

$$f'(u \to v) \;=\; f(u \to v) - F \;<\; f(u \to v) \;\leq\; c(u \to v),$$

이번에도 f가 실현 가능하기 때문이며,

$$\begin{aligned} f'(u \to v) &= f(u \to v) - F && f'\text{의 정의에 의해} \\ &\geq f(u \to v) - c_f(v \to u) && F\text{의 정의에 의해} \\ &= f(u \to v) - f(u \to v) && c_f\text{의 정의에 의해} \\ &= 0 && \text{당연합니다.} \end{aligned}$$

- 마지막으로, $u \to v$와 $v \to u$ 중 어느 것도 증가 경로에 없다면 $f'(u \to v) = f(u \to v)$ 이며, f가 실행 가능하므로 $0 \le f'(u \to v) \le c(u \to v)$입니다.

따라서 f는 실제로 실현 가능합니다.

마지막으로, 증가 경로에서 s를 떠나는 간선은 첫 번째 간선뿐이므로 $|f'| = |f| + F > |f|$입니다. 따라서 f'은 f보다 큰 값을 갖는 실현 가능한 흐름입니다. 결론적으로, 잔여 그래프 G_f에 s에서 t로 가는 경로가 있으면 f는 최대 흐름이 *아닙니다*.

반면, 잔여 그래프 G_f에 s에서 t로 가는 방향 경로가 *없다고* 가정합시다. G_f에서 s로부터 도달 가능한 정점들의 집합을 S라 하고, $T = V \setminus S$라 합시다. 분할 (S, T)는 명백히 (s, t)-컷입니다. 모든 정점 $u \in S$와 $v \in T$에 대해 다음이 성립합니다

$$c_f(u \to v) \;=\; (c(u \to v) - f(u \to v)) + f(v \to u) \;=\; 0.$$

f가 실현 가능하므로 $c(u \to v) - f(u \to v) \ge 0$이고 $f(v \to u) \ge 0$입니다. 따라서 실제로는 $c(u \to v) - f(u \to v) = 0$이고 $f(v \to u) = 0$이어야 합니다. 다시 말해, 흐름 f는 S에서 T로 가는 모든 간선을 포화시키고 T에서 S로 가는 모든 간선은 피합니다. 이제 보조정리 10.1에 의해 $|f| = \|S, T\|$이며, 이는 f가 최대 흐름이고 (S, T)가 최소 컷임을 의미합니다.

이로써 증명이 끝났습니다! □

10.4 포드와 풀커슨의 증가 경로 알고리즘

포드와 풀커슨의 최대 흐름-최소 컷 정리 증명은 최대 흐름을 계산하는 알고리즘을 곧바로 제시합니다. 영 흐름에서 시작하여, 잔여 그래프에서 s에서 t로 가는 **아무** 경로를 따라 흐름을 증가시키는 일을 그러한 경로가 더 이상 없을 때까지 반복합니다.

이 알고리즘에서 중요하면서도 간단한 따름정리를 하나 얻을 수 있습니다:

정수성 정리. *흐름 네트워크의 모든 용량이 정수이면, 모든 간선을 지나는 흐름이 정수인 최대 흐름이 존재합니다.*

증명: 증가 경로 알고리즘을 한 번 반복할 때마다, 반복이 끝난 뒤의 모든 흐름 값과 모든 잔여 용량이 정수라는 것을 귀납법으로 증명합니다.

- 첫 번째 반복 전에는 모든 흐름 값이 0이며 이는 정수입니다. 또한 모든 잔여 용량은 원래 용량과 같고, 원래 용량은 정의상 정수입니다.

- 이후의 각 반복에서는 귀납법 가정에 따라 증가 경로의 용량 F가 정수이므로, 증가 연산은 각 간선의 흐름을 정수만큼 바꾸며, 따라서 각 간선의 잔여 용량도 정수만큼 바뀝니다.

특히 증가 경로 알고리즘은 반복할 때마다 흐름의 값을 양의 정수만큼 증가시킵니다. 따라서 이 알고리즘은 결국 종료되며 최대 흐름을 반환합니다. □

모든 간선 용량이 정수라면, 보수적으로 보아 포드-풀커슨 알고리즘은 최대 $|f^*|$번의 반복 후에 멈춥니다. 여기서 f^*는 실제 최대 흐름입니다. 각 반복에서 잔여 그래프 G_f를 구성하고 아무거나 우선 탐색을 수행하여 $O(E)$ 시간에 증가 경로를 찾을 수 있습니다. 따라서 이 설정에서 알고리즘은 최악의 경우 $\boldsymbol{O(E|f^*|)}$ 시간에 실행됩니다.

잭 에드먼즈(Jack Edmonds)와 리처드 카프는 이 실행 시간 분석이 본질적으로 빈틈이 없다는 것을 관찰했습니다. 그림 10.7의 4-노드 네트워크를 생각해 봅시다. 여기서 X는 어떤 큰 정수입니다. 이 네트워크의 최대 흐름은 분명히 $2X$입니다. 그러나 포드-풀커슨은 증가 경로 $s \to u \to v \to t$를 따라 흐름 한 단위를 보내는 것과 증가 경로 $s \to v \to u \to t$를 따라 흐름 한 단위를 보내는 것을 번갈아 할 수 있으며, 그 결과 실행 시간은 $\Theta(X) = \Omega(|f^*|)$가 됩니다.

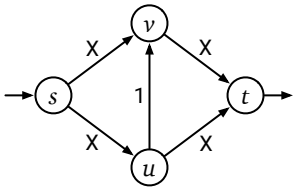


그림 10.7. 포드-풀커슨 알고리즘에 대한 에드먼즈와 카프의 나쁜 예입니다.

포드와 풀커슨의 알고리즘은 실제로는 대개 빠르며, 최대 흐름 값 $|f^*|$가 작을 때는 항상 빠르지만, 증가 경로에 추가적인 제약이 없으면 최악의 경우 효율적인 알고리즘이 *아닙니다*. Edmonds와 카프의 나쁜 예시 네트워크는 $O(\log X)$ 비트만으로 기술할 수 있습니다. 따라서 포드-풀커슨의 실행 시간은 사실 입력 크기에 대해 *지수적*입니다.

♥무리수 용량

그런데 용량이 정수가 *아니라면* 어떻게 될까요? 모든 용량에 같은 (양의) 상수를 곱하면 최대 흐름도 모든 곳에서 같은 상수 배만큼 커집니다. 따라서 모든 간선 용량이 *유리수*이면 포드-풀커슨 알고리즘은 결국 멈춥니다. 다만 여전히 (입력을 나타내는 데 쓰이는 비트 수에 대해) 지수 시간이 걸립니다.

그러나 *무리수* 용량을 허용하면 이 알고리즘은 점점 더 작은 증가 경로를 계속 찾으면서 실제로 영원히 반복할 수 있습니다. 더 나쁜 것은, 이 무한한 증가 수열이 최대 흐름으로 수렴하지 않을 수도 있고, 심지어 최대 흐름의 상당한 비율에도 미치지 못할 수 있다는 점입니다! 이러한 나쁜 동작을 보이는 가장 작은 네트워크는 1993년에 Uri Zwick이 발견했습니다.[2]

그림 10.8에 나타낸 6개 노드로 이루어진 네트워크를 생각해 봅시다. 9개의 간선 중 6개는 어떤 큰 정수 용량 X를 가지고, 2개는 용량 1을 가지며, 나머지 1개는 $1 - \phi = \phi^2$가 되도록 선택한 용량 $\phi = (\sqrt{5} - 1)/2 \approx 0.618034$를 가집니다. 포드-풀커슨 알고리즘이 멈추지 못할 수 있음을 증명하기 위해, 알고리즘이 진행되는 동안 세 수평 간선의 잔여 용량을 관찰할 수 있습니다. (나머지 6개 간선의 잔여 용량은 항상 $X - 3$ 이상입니다.)

포드-풀커슨 알고리즘이 그림 10.8의 맨 위에 보인 가운데 증가 경로를 먼저 선택한다고 가정합시다. 이제 세 수평 간선의 잔여 용량은 왼쪽에서 오른쪽 순서로 1, 0, ϕ입니다. 어떤 음이 아닌 정수 k에 대해 수평 잔여 용량이 ϕ^{k-1}, 0, ϕ^k라고 귀납적으로 가정합시다.

[2] 1962년에 포드와 풀커슨은 정점 10개와 간선 48개로 이루어진 더 복잡한 네트워크를 제시했는데, 이 네트워크도 같은 나쁜 동작을 보입니다.

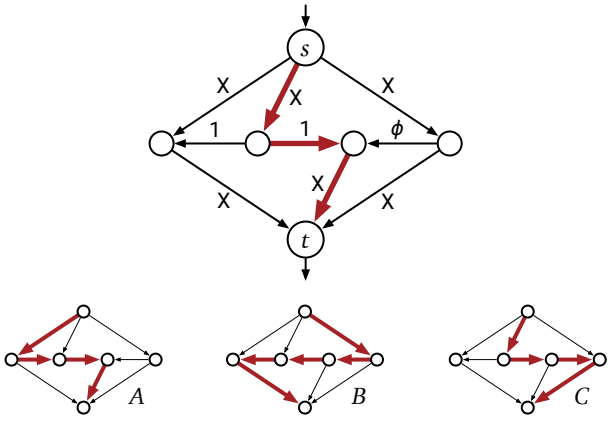


그림 10.8. Uri Zwick의 종료되지 않는 흐름 예제와 세 개의 증가 경로입니다.

1. 경로 B를 따라 흐름을 증가시켜 흐름에 ϕ^k를 더합니다. 이제 잔여 용량은 ϕ^{k+1}, ϕ^k, 0입니다.
2. 경로 C를 따라 흐름을 증강하여 흐름에 ϕ^k를 더합니다. 이제 잔여 용량은 ϕ^{k+1}, 0, ϕ^k입니다.
3. 경로 B를 따라 흐름을 증가시켜 ϕ^{k+1}을 더합니다. 이제 잔여 용량은 0, ϕ^{k+1}, ϕ^{k+2}입니다.
4. 경로 A를 따라 증강하여 흐름에 ϕ^{k+1}을 더합니다. 이제 잔여 용량은 ϕ^{k+1}, 0, ϕ^{k+2}입니다.

귀납법에 의해 $4n+1$번의 증가 단계를 거친 후 수평 간선들의 잔여 용량은 ϕ^{2n-2}, 0, ϕ^{2n-1}이 됩니다. 증가 횟수가 무한히 커지면 흐름의 값은 다음 값으로 수렴합니다.

$$1 + 2\sum_{i=1}^{\infty} \phi^i = 1 + \frac{2}{1-\phi} = 4 + \sqrt{5} < 7,$$

최대 흐름 값은 분명히 $2X + 1 \gg 7$인데도 말입니다.

실용적인 성향의 독자라면 왜 굳이 무리수 용량에 신경 써야 하는지 궁금할 수 있습니다. 어차피 컴퓨터는 (작은) 정수나 (작은 이진) 유리수가 아니면 정확하게 표현할 수 없기 때문입니다. 좋은 질문입니다! 수학자의 답은 용량을 정수로 제한하는 것이 말 그대로 *인위적*이라는 것입니다. 이 제한은 디지털 계산 하드웨어(또는 그 밖의 경우에는 무관한 물리 법칙)가 만들어 낸 *인공물*일 뿐, 추상적인 계산 문제에 내재된 특징이 아닙니다. 그러나 더 실용적인 이유도 있습니다. 무리수 입력에 대한 알고리즘의 동작은 부동소수점 용량을 사용할 때 *실제로* 나타나는 최악의 경우 동작에 대해 무언가를 알려 줍니다. 그 동작은 끔찍합니다! 매우 합리적인 용량을 사용하더라도 포드-풀커슨을 부주의하게 구현하면 단지 반올림 오차 때문에 정답 근처에도 가지 못한 채 무한 루프에 빠질 수 있습니다.

10.5 흐름의 결합과 분해

흐름은 일반적으로 그래프의 간선 위에서 정의되는 함수로서, 각 정점에서 특정한 제약 조건을 만족하는 것으로 정의됩니다. 그러나 흐름에는 두 번째 특성화도 있으며, 어떤 맥락에서는 이쪽이 더 자연스럽고 유용합니다.

소스 정점 s와 대상 정점 t를 갖는 임의의 그래프 G를 생각해 봅시다. 임의의 두 (s,t)-흐름 f와 g, 그리고 임의의 두 실수 α와 β를 고정하고, 다음과 같이 정의되는 함수 $h : E \to \mathbb{R}$을 생각해 봅시다.

$$h(u \to v) := \alpha \cdot f(u \to v) + \beta \cdot g(u \to v)$$

이는 모든 간선 $u \to v$에 대해 성립하며, 이 정의는 $h = \alpha f + \beta g$로 더 간단히 쓸 수 있습니다. 정의를 그대로 따라가 보면 h 역시 값이 $|h| = \alpha|f| + \beta|g|$인 (s,t)-흐름임을 알 수 있습니다. 더 일반적으로, (s,t)-흐름들의 임의의 선형 결합 역시 (s,t)-흐름입니다.

사실 모든 (s,t)-흐름은 특별한 구조를 갖는 흐름들의 가중합으로 나타낼 수 있습니다. s에서 t로 가는 임의의 방향 경로 P에 대해, 이에 대응하는 ***경로 흐름***을 다음과 같이 정의합니다:

$$P(u \to v) = \begin{cases} 1 & u \to v \in P\text{이면,} \\ -1 & v \to u \in P\text{이면,} \\ 0 & \text{그렇지 않은 경우.} \end{cases}$$

정의를 그대로 따라가 보면 함수 $P : E \to \mathbb{R}$가 실제로 값이 1인 (s,t)-흐름임을 알 수 있습니다. 저는 변수 P가 경로(정점과 방향 간선의 수열)와 그 경로를 따라 흐르는 단위 흐름을 모두 뜻하도록 일부러 중복해서 사용하고 있습니다.

마찬가지로, 임의의 방향 사이클 C에 대해 다음과 같이 설정하여 대응하는 ***사이클 흐름***을 정의합니다

$$C(u \to v) = \begin{cases} 1 & u \to v \in C\text{이면,} \\ -1 & v \to u \in C\text{이면,} \\ 0 & \text{그렇지 않은 경우.} \end{cases}$$

이번에도 $C : E \to \mathbb{R}$이 값이 0인 (s,t)-흐름임을 쉽게 확인할 수 있습니다.

앞서 살펴본 논증에 따르면 경로 흐름과 사이클 흐름의 임의의 선형 결합은 또 다른 흐름이 됩니다. 이러한 가중합을 ***흐름 분해***라고 합니다. 더 나아가, 모든 음이 아닌 흐름은 다음과 같은 특별한 구조를 갖는 흐름 분해를 가집니다.

흐름 분해 정리. *음이 아닌 모든 (s,t)-흐름 f는 방향 (s,t)-경로들과 방향 사이클들의 양의 선형 결합으로 나타낼 수 있습니다. 또한 방향 간선 $u \to v$가 이 경로들이나 사이클들 중 적어도 하나에 나타날 필요충분조건은 $f(u \to v) > 0$이며, 경로와 사이클의 총 개수는 네트워크의 간선 수를 넘지 않습니다.*

증명: 0이 아닌 흐름이 흐르는 간선의 수에 대한 귀납법으로 정리를 증명합니다. 직관적으로는 포드-풀커슨 알고리즘을 거꾸로 실행하는 방식입니다. 그래프에서 적어도 하나의 간선에 양의 흐름이 흐르는 한, 흐름이 흐르는 (s,t)-경로나 방향 사이클 중 하나를 찾을 수 있습니다. 그 경로나 사이클에서 가능한 한 많은 흐름을 빼면 적어도 하나의 간선이 비게 되므로, 분해의 나머지는 재귀 요정이 알려줄 수 있습니다.

이 논증을 형식화하기 위해, 먼저 ***순환***이라는 특수한 경우를 살펴봅니다. 순환은 값이 0인 흐름으로, 모든 정점에서 흐름이 보존됩니다.

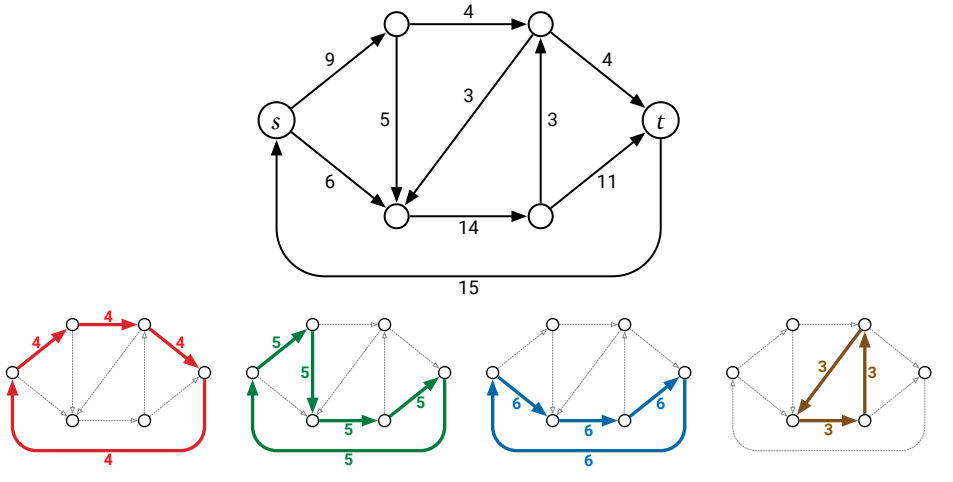


그림 10.9. 순환을 가중치가 있는 방향 사이클들로 분해하기.

임의의 흐름 네트워크에서 임의의 순환 흐름 f를 고정하고, $f(u{\to}v) > 0$을 만족하는 간선 $u{\to}v$의 개수를 $\#f$로 나타냅시다. $\#f$에 대한 귀납법으로, f를 최대 $\max\{0, \#f - 1\}$개의 사이클의 양의 선형 결합으로 분해할 수 있음을 증명합니다. 고려해야 할 경우는 세 가지입니다.

- $\#f = 0$이면, f는 공허하게 0개의 사이클의 선형 결합입니다.
- $f(u{\to}v) > 0$인 간선 $u{\to}v$들이 하나의 방향 사이클을 이룬다고 가정합시다. 그러면 $\#f \geq 2$이고, f는 자명하게 사이클 하나의 선형 결합입니다.
- 그렇지 않다면, $f(u{\to}v) > 0$인 임의의 간선 $u{\to}v$를 하나 고릅니다. $v_0 = u$이고 $v_1 = v$이며 모든 인덱스 i에 대해 $f(v_{i-1}{\to}v_i) > 0$을 만족하는 임의의 워크 $v_0{\to}v_1{\to}v_2{\to}\cdots$를 생각해 봅시다. 보존 제약 조건에 따르면 들어오는 흐름이 있는 모든 정점은 나가는 흐름도 가지므로, 이 워크를 얼마든지 길게 만들 수 있습니다. 특히 이 워크는 결국 어떤 정점을 두 번 이상 방문할 수밖에 없습니다. 어떤 인덱스 $j < k$에 대해 $v_j = v_k$를 만족하는 가장 작은 인덱스를 k라고 합시다. 부분 워크 $v_j{\to}v_{j-1}{\to}\cdots{\to}v_k$는 단순 방향 사이클 C입니다.

  $F := \min_{e\in C} f(e)$로 정의하고, 함수 $f' := f - F \cdot C$를 생각해 봅시다. 좀 더 자세히 풀어 쓰면 다음과 같습니다.

$$f'(u{\to}v) := \begin{cases} f(u{\to}v) - F & u{\to}v \in C\text{이면,} \\ f(u{\to}v) & \text{그렇지 않은 경우.} \end{cases}$$

정의를 그대로 따라가 보면 f' 역시 G의 실현 가능한 순환임을 알 수 있습니다. $f(e) = F$인 간선 $e \in C$가 적어도 하나 존재하므로 $f'(e) = 0$이며, 따라서 $\#f' \leq \#f - 1$입

니다. f'에서 흐름이 지나는 간선의 수가 f에서보다 적으므로, 재귀 요정이 f'을 최대 $\#f' - 1 \le \#f - 2$개의 사이클로 분해할 수 있습니다. 여기에 사이클 C를 따라 F 단위의 흐름을 더하면 f의 흐름 분해를 얻습니다. 더 간결하게 쓰면 $f = f' + F \cdot C$입니다. 이제 f를 임의의 흐름 네트워크에서 $|f| > 0$을 만족하는 임의의 (s,t)-흐름이라고 합시다. 네트워크에 간선 $t \to s$를 추가하고, $f'(t \to s) = |f|$로 두고 원래의 모든 간선 $u \to v$에 대해 $f'(u \to v) = f(u \to v)$로 두어 순환 f'을 정의합니다. 이때 $\#f' = \#f + 1 \ge 2$임에 주목하십시오. 앞의 논증에 따르면 순환 f'은 최대 $\#f' - 1$개의 방향 사이클의 양의 선형 결합입니다. 간선 $t \to s$를 삭제하면 원래 흐름 f를 최대 $\#f' - 1 = \#f$개의 경로와 사이클로 분해할 수 있습니다. 구체적으로 말하면, f'에서 $t \to s$를 포함하는 사이클은 f에서 (s,t)-경로가 되고, f'에서 $t \to s$를 포함하지 않는 사이클은 f에서도 그대로 사이클로 남습니다. □

흐름 분해 정리(Flow Decomposition Theorem)의 증명은 흥미로운 두 가지 특수한 경우에 더 강한 결과를 함의합니다.

- 모든 순환(circulation)은 사이클들의 가중합으로 분해할 수 있으며, 경로는 전혀 필요하지 않습니다.
- 모든 ***비순환*** (s,t)-흐름은 (s,t)-경로들의 가중합으로 분해할 수 있으며, 사이클은 전혀 필요하지 않습니다.

또한, 흐름 사이클이 더 이상 남지 않을 때까지 흐름 사이클을 상쇄하면 어떤 흐름이든 같은 값을 갖는 비순환 흐름으로 변환할 수 있습니다. 특히 모든 흐름 네트워크에는 비순환인 최대 (s,t)-흐름이 존재합니다.

이 증명은 또한 포드-풀커슨과 유사하게, 임의의 (s,t)-흐름을 경로와 사이클로 분해하는 알고리즘으로 곧바로 옮겨집니다. 이 알고리즘은 흐름이 비게 될 때까지 남은 흐름에서 방향 (s,t)-경로나 방향 사이클을 반복해서 찾은 다음, 그 경로나 사이클을 따라 가능한 한 많은 흐름을 뺍니다. 흐름 경로나 사이클은 다음과 같이 $O(V)$ 시간에 찾을 수 있습니다.

- s에서 나가는 간선 중 양의 흐름을 갖는 간선이 하나라도 있다면, 흐름 그래프에서 s로부터 임의의 보행을 따라가되, t에 도달하거나(이 경우 흐름 경로를 얻습니다) 어떤 정점에 두 번째로 도달할 때까지(이 경우 흐름 사이클을 얻습니다) 진행합니다.
- s에서 나가는 간선 중 양의 흐름을 갖는 간선이 하나도 없으면, 양의 유출량을 갖는 다른 정점 v를 아무것이나 찾은 다음, 흐름 그래프에서 v로부터 임의의 보행을 따라가되 어떤 정점에 두 번째로 도달할 때까지 진행합니다(이로써 흐름 사이클을 얻습니다).

두 경우 모두 보존 제약 조건에 의해 이 알고리즘은 절대 막히지 않습니다. 각 반복은 $O(V)$ 시간이 걸리고 흐름 그래프에서 간선을 적어도 하나 제거합니다. 따라서 전체 분해 알고리즘은 ***$O(VE)$ 시간***에 실행됩니다.

흐름 분해는 흐름을 한 번에 경로 하나 또는 사이클 하나씩 구성하는 모든 최대 흐름 알고리즘의 실행 시간에 대한 자연스러운 하한을 제공합니다. 모든 흐름은 최대 E개의 경로와 사이클로 분해할 수 있고, 각 경로와 사이클은 최대 V개의 간선을 사용하므로 흐름 분해의 전체 복잡도는 $O(VE)$입니다. 더욱이 모든 흐름 분해의 복잡도가

$\Omega(VE)$인 흐름을 구성하는 것은 어렵지 않습니다. 따라서 흐름을 한 번에 경로 하나 또는 사이클 하나씩 명시적으로 구성하는 모든 최대 흐름 알고리즘—특히 포드와 풀커슨의 증가 경로 알고리즘의 모든 구현—은 최악의 경우 $\Omega(VE)$ 시간이 걸릴 수밖에 없습니다.

10.6 에드먼즈와 카프의 알고리즘

포드와 풀커슨의 알고리즘은 잔여 그래프에서 어떤 경로를 따라 증가시킬지 지정하지 않습니다. 이 알고리즘의 최악의 경우 성능이 나쁜 것은 증가 경로를 잘못 선택한 탓으로 볼 수 있습니다. 1970년대 초, 잭 에드먼즈와 리처드 카프는 증가 경로를 선택하는 두 가지 자연스러운 규칙을 발표했으며, 두 규칙 모두 더 효율적인 알고리즘으로 이어졌습니다.

가장 굵은 증가 경로

에드먼즈와 카프의 첫 번째 규칙은 본질적으로 탐욕 알고리즘입니다:

> 병목값이 가장 큰 증가 경로를 선택합니다.

방향 그래프에서 최대 병목 (s,t)-경로를 Jarník의 최소 신장 트리 알고리즘이나 Dijkstra의 최단 경로 알고리즘과 비슷한 "최선 우선(best-first)" 순회로 $O(E \log V)$ 시간에 계산할 수 있음을 보이는 것은 어렵지 않습니다. 이 알고리즘은 s를 루트로 하는 방향 트리 T를 한 번에 정점 하나씩 키워 나갑니다. T가 s에서 t로 가는 경로를 포함할 때까지, T에서 나가는 간선 중 용량이 가장 큰 간선을 T에 반복해서 추가합니다. 다른 방법으로 Kruskal 알고리즘을 흉내 낼 수도 있습니다—s에서 t로 가는 경로가 생길 때까지 간선을 용량이 큰 순서대로 하나씩 삽입하는 것입니다—다만 이 방식은 적어도 그래프가 방향 그래프일 때는 효율이 더 떨어집니다.

흐름 알고리즘의 실행 시간 분석을 마무리하려면 알고리즘이 멈출 때까지 반복하는 횟수의 상한이 필요합니다. 사실 용량이 임의의 실수라면 알고리즘이 *절대* 멈추지 않을 수도 있습니다. 연습문제 18을 참고하십시오. 그러나 용량이 정수라면 다음과 같이 반복 횟수의 상한을 최대 흐름 값 $|f^*|$의 함수로 나타낼 수 있습니다.

G에서의 임의의 흐름을 f라 하고, *현재 잔여 그래프* G_f에서의 최대 흐름을 f'이라 합시다. (알고리즘을 시작할 때는 $G_f = G$이고 $f' = f^*$입니다.) f'이 최대 E개의 경로와 사이클로 분해될 수 있음은 이미 증명했습니다. 간단한 평균 논증에 따르면 이 분해에 속한 경로 중 적어도 하나는 반드시 $|f'|/E$ 단위 이상의 흐름을 운반합니다. 따라서 G_f에서 *가장 굵은* (s,t)-경로가 $|f'|/E$ 단위 이상의 흐름을 운반한다는 사실이 곧바로 따라 나옵니다.

따라서 G_f에서 최대 병목 경로를 따라 f를 증가시키면 G_f에 남아 있는 최대 흐름의 값에 많아야 $1 - 1/E$인 인수가 곱해집니다. 다시 말해, 잔여 최대 흐름 값은 반복 횟수에 따라 *지수적으로 감소*합니다. $E \cdot \ln |f^*|$번 반복한 후 G_f의 최대 흐름 값은 많아야 다음과 같습니다.

$$|f^*| \cdot (1 - 1/E)^{E \cdot \ln |f^*|} < |f^*| e^{-\ln |f^*|} = 1.$$

(여기서 e는 오일러 상수 e이지 간선 e가 아닙니다. 죄송합니다.) 특히 $E \cdot \ln |f^*|$번 반복한 뒤에는 잔여 최대 흐름 값이 1보다 작아집니다. *모든 용량이 정수라면* 잔여 최대 흐름 값도 정수이므로 이 값은 반드시 0이어야 합니다. 다시 말해, f는 최대 흐름입니다!

따라서 정수 용량을 갖는 그래프에서 에드먼즈-카프의 "가장 굵은 경로" 알고리즘은 $O(E^2 \log E \log |f^*|)$ 시간에 실행된다는 결론을 얻습니다. 순수한 포드-풀커슨 알고리즘의 최악의 경우 실행 시간과 달리, 이 시간 한계는 실제로 입력 크기에 대한 다항 함수입니다.

원래의 포드-풀커슨 알고리즘과 마찬가지로, "가장 굵은 경로" 알고리즘도 임의의 실수 용량을 갖는 네트워크에서는 무한 루프에 빠질 수 있습니다. 그러나 우리의 분석에 따르면, 이 알고리즘은 결코 멈추지 않더라도 극한에서 최대 흐름에 수렴하는 흐름 f를 유지합니다.

최단 증가 경로

두 번째 에드먼즈-카프 규칙은 사실 포드와 풀커슨이 최대 흐름에 관한 최초의 논문에서 실용적인 휴리스틱으로 제안한 것입니다. 이 규칙의 한 변형은 1970년에 러시아 수학자 예핌 디니츠(Yefim Dinitz)가 독립적으로 제안했습니다.[3]

> 간선 수가 가장 적은 증가 경로를 선택하십시오.

잔여 그래프에서 너비 우선 탐색을 실행하면 최단 증가 경로를 $O(E)$ 시간에 찾을 수 있습니다. 놀랍게도 이렇게 얻은 알고리즘은 실제 간선 용량과 관계없이 다항식 횟수의 반복 후에 종료됩니다!

이 다항식 상한의 증명은 잔여 그래프가 변화하는 과정에 관한 두 가지 관찰에 기반합니다. i번의 증가 단계를 거친 후의 현재 흐름을 f_i라 하고, 이에 대응하는 잔여 그래프를 G_i라 합시다. 특히 f_0은 모든 곳에서 0이고 $G_0 = G$입니다. 각 정점 v에 대해 $\mathit{level}_i(v)$가 G_i에서 s로부터 v까지의 가중치 없는 최단 경로 거리를 나타낸다고 합시다. 이는 s를 루트로 하는 G_i의 너비 우선 탐색 트리에서 v의 *레벨*과 같습니다. 특히 G_i에 s에서 v로 가는 경로가 없으면 $\mathit{level}_i(v) = \infty$입니다($\min \varnothing = \infty$이기 때문입니다).

첫 번째로 관찰할 수 있는 사실은 정점의 레벨이 시간이 지남에 따라 증가할 수만 있다는 것입니다.

보조정리 10.2. *모든 정점 v와 모든 정수 $i > 0$에 대해 $\mathit{level}_i(v) \geq \mathit{level}_{i-1}(v)$입니다.*

증명: 임의의 양의 정수 $i > 0$과 임의의 정점 v를 고정합니다. 이 주장은 (정수 i에 대한 귀납법이 *아니라*) $\mathit{level}_i(v)$에 대한 귀납법으로 증명합니다. 귀납 가정으로서, $\mathit{level}_i(u) < \mathit{level}_i(v)$를 만족하는 모든 정점 u에 대해 $\mathit{level}_i(u) \geq \mathit{level}_{i-1}(u)$라고 가정합니다. 고려해야 할 경우는 세 가지입니다.

- $v = s$이면 곧바로 $\mathit{level}_i(s) = \mathit{level}_{i-1}(s) = 0$임을 알 수 있습니다.
- G_i에 s에서 v로 가는 경로가 없으면 $\mathit{level}_i(v) = \infty \geq \mathit{level}_{i-1}(v)$입니다.

[3] 구체적으로 말하면, 디니츠는 게오르기 아델손-벨스키(AVL 트리의 "AV")가 가르치던 알고리즘 수업을 듣던 학생 시절에 수업 중 연습 문제에 대한 답으로 더 복잡한 최대 흐름 알고리즘을 발견했습니다. 디니츠의 알고리즘도 최단 경로를 따라 흐름을 보내지만, 추가적인 기록 관리를 통해 실행 시간을 $O(VE^2)$에서 $O(V^2E)$로 줄입니다.

- 그렇지 않다면, 그래프 G_i에서 s에서 v로 가는 임의의 가중치 없는 최단 경로를 $s \to \cdots \to u \to v$라고 합시다. 이것은 최단 경로이므로 $level_i(v) = level_i(u) + 1$이 성립하며, 따라서 귀납 가정에 의해 $level_i(u) \geq level_{i-1}(u)$가 성립합니다. 증명을 완성하려면 $level_{i-1}(u) \geq level_{i-1}(v) - 1$임을 보여야 합니다. 고려해야 할 하위 경우는 두 가지입니다.
  - $u \to v$가 G_{i-1}의 간선이면, 레벨은 너비 우선 순회로 정의되므로 $level_{i-1}(v) \leq level_{i-1}(u) + 1$입니다.
  - 반면 $u \to v$가 G_{i-1}의 간선이 아니라면, 그 역방향 간선 $v \to u$는 반드시 i번째 증가 경로의 간선이어야 하며, 이 경로는 정의에 따라 G_{i-1}에서 s로부터 t까지의 최단 경로입니다. 따라서 $level_{i-1}(v) = level_{i-1}(u) - 1 \leq level_{i-1}(u) + 1$이 성립합니다.

  두 하위 경우 모두에서 $level_i(v) = level_i(u) + 1 \geq level_{i-1}(u) + 1 \geq level_{i-1}(v)$라는 결론을 얻습니다. □

흐름을 증가시킬 때마다 증가 경로의 병목 간선은 잔여 그래프에서 사라지고, 증가 경로를 *뒤집은* 경로에 있는 일부 간선이 (다시) 나타날 수 있습니다. 두 번째 관찰은 한 간선이 나타나거나 사라지는 횟수가 그리 많을 수 없다는 것입니다.

보조정리 10.3. *에드먼즈-카프 최단 증가 경로 알고리즘이 실행되는 동안, 각 간선 $u \to v$는 잔여 그래프 G_f에서 최대 $V/2$번 사라집니다.*

증명: 어떤 $i < j$에 대해 $u \to v$가 두 잔여 그래프 G_i와 G_{j+1}에는 있지만 그 사이의 잔여 그래프 $G_{i+1}, \ldots, G_j$에는 하나도 없다고 가정합시다. 그러면 $u \to v$는 i번째 증가 경로에 있어야 하므로 $level_i(v) = level_i(u) + 1$이고, $v \to u$는 j번째 증가 경로 위에 있어야 하므로 $level_j(v) = level_j(u) - 1$입니다. 앞의 보조정리에 따르면 다음이 성립합니다

$$level_j(u) \;=\; level_j(v) + 1 \;\geq\; level_i(v) + 1 \;=\; level_i(u) + 2.$$

다시 말해, $u \to v$가 사라졌다가 다시 나타나는 사이에 s에서 u까지의 거리는 적어도 2만큼 증가했습니다. 모든 레벨은 V보다 작거나 무한대이므로, 사라지는 횟수는 많아야 $V/2$입니다. □

이제 반복 횟수의 상한을 유도할 수 있습니다. 각 간선은 최대 $V/2$번 사라지므로, 전체적으로 간선이 사라지는 횟수는 최대 $EV/2$번입니다. 그런데 반복할 때마다 적어도 하나의 간선이 사라지므로, 알고리즘은 최대 $EV/2$번 반복한 후에 반드시 멈춰야 합니다. 마지막으로, 각 반복에는 $O(E)$ 시간이 걸리므로, 전체 알고리즘은 $\boldsymbol{O(VE^2)}$ *시간*에 실행됩니다.

10.7 추가 진전

최대 흐름 알고리즘의 이야기는 아직 끝나려면 한참 멀었습니다. 이후 수십 년에 걸친 연구를 통해 더 빠른 알고리즘이 여럿 등장했으며, 그중 일부를 그림 10.10에 정리했습니다.[4] 나열된 알고리즘은 모두 여러 번의 반복을 거쳐 최대 흐름을 계산합니다. 이

[4] 표를 짧게 유지하기 위해 실행 시간이 간선 용량이나 최대 흐름 값에 의존하는 알고리즘은 일부러 제외했습니다. 이렇게 제한했는데도 목록은 민망할 정도로 불완전합니다!

알고리즘들은 대부분 두 가지 변형이 있습니다. 하나는 각 반복을 무차별 대입으로 수행하는 더 단순한 버전이고, 다른 하나는 정교한 자료구조를 사용해 흐름 네트워크의 신장 트리를 유지함으로써 각 반복을 로그 시간에 수행하고 신장 트리도 로그 시간에 갱신할 수 있는 더 빠른 변형입니다. 지금까지 알려진 최선의 알고리즘이 최적이라고 믿을 이유는 없습니다. 실제로 최대 흐름은 지금도 매우 활발하게 연구되는 분야입니다.

기법	직접	동적 트리를 이용하는 경우	출처
차단 흐름	$O(V^2E)$	$O(VE\log V)$	[Dinitz; Karzanov; Even과 Itai; Sleator와 Tarjan]
네트워크 심플렉스	$O(V^2E)$	$O(VE\log V)$	[Dantzig; Goldfarb and Hao; Goldberg, Grigoriadis, and Tarjan]
푸시-재레이블(일반형)	$O(V^2E)$	–	[Goldberg와 Tarjan]
푸시-재레이블(FIFO)	$O(V^3)$	$O(VE\log(V^2/E))$	[Goldberg와 Tarjan]
푸시-재레이블(최고 레이블)	$O(V^2\sqrt{E})$	–	[Cheriyan and Maheshwari; Tunçel]
푸시-재레이블-추가 게임	–	$O(VE\log_{E/(V\log V)}V)$	[Cheriyan과 Hagerup; King, Rao, Tarjan]
의사흐름	$O(V^2E)$	$O(VE\log V)$	[Hochbaum]
의사흐름(최고 레이블)	$O(V^3)$	$O(VE\log(V^2/E))$	[Hochbaum and Orlin]
점진적 너비 우선 탐색	$O(V^2E)$	$O(VE\log(V^2/E))$	[Goldberg, Held, Kaplan, Tarjan, 및 Werneck]
간결한 네트워크	–	$O(VE)$	[Orlin]

그림 10.10. 몇 가지 순수 조합적 최대 흐름 알고리즘과 각각의 실행 시간.

2012년 James Orlin이 발표한, 현재 알려진 가장 빠른 (순수 조합적) 최대 흐름 알고리즘은 $O(VE)$ *시간*에 실행되며, 이는 흐름 분해의 최악의 경우 복잡도와 정확히 일치합니다. Orlin의 알고리즘의 세부 사항은 이 책의 범위를 훨씬 넘어섭니다. Orlin은 자신이 새로 고안한 기법 외에도 여러 기존 알고리즘과 자료구조를 블랙박스로 사용하는데, 그중 대부분은 그 자체로도 상당히 복잡합니다. 특히 Orlin의 알고리즘은 명시적인 흐름 분해를 구성하지 *않습니다*. 실제로 간선이 $O(V)$개뿐인 그래프에 대해서는 그의 알고리즘을 확장한 버전이 $O(V^2/\log V)$ 시간 만에 실행됩니다! 그럼에도 최대 흐름을 *사용하는* 알고리즘을 분석할 때 인용해야 하는 시간 한계는 바로 이것입니다. 그러므로 다음 문장을 여러분의 요약 노트에 적어 두고 과제에서 인용하십시오:

> ***최대 흐름은 $O(VE)$ 시간에 계산할 수 있습니다.***

마지막으로, 모든 간선의 용량이 1인 *단위 용량* 네트워크에 대해서는 더 빠른 최대 흐름 알고리즘이 알려져 있습니다. 1973년에 Alexander Karzanov는 Dinitz의 차단 흐름 알고리즘—위 표에 나열된 첫 번째 알고리즘—이 이 설정에서 $O(\min\{V^{2/3}, E^{1/2}\}E)$ 시간에 실행됨을 증명했습니다. (이 시간 한계는 $\Omega(VE)$ 흐름 분해 장벽을 깨는 것처럼 보이지만, 사실 Karzanov의 분석은 단위 용량 네트워크의 모든 흐름을 총 복잡도가 $O(\min\{V^{2/3}, E^{1/2}\}E)$인 경로들로 분해할 수 있음을 함의합니다.) 이 알고리즘은 40년 동안 이 설정에서 알려진 가장 빠른 알고리즘이었습니다.

Karzanov의 기록은 2013년에 마침내 깨졌는데, 이때 Aleksander Mądry가 단위 용량 네트워크에서 최대 흐름을 $O(E^{10/7}\operatorname{polylog} E)$ 시간에 계산하는 실로 놀라운 알고리즘을 발표했습니다. 이번에도 Mądry 알고리즘의 세부 사항은 이 책의 범위를, 아니 사실은 저자의 전문 지식을 훨씬 넘어섭니다.

연습문제

0. 방향 그래프 $G = (V, E)$, 두 정점 s와 t, 용량 함수 $c : E \to \mathbb{R}^+$, 그리고 두 번째 함수 $f : E \to \mathbb{R}$가 주어졌다고 가정합시다. f가 G의 최대 (s,t)-흐름인지 판별하는 알고리즘을 기술하십시오.

1. 흐름 네트워크 G에서 f와 f'이 $|f'| > |f|$를 만족하는 두 실현 가능한 (s,t)-흐름이라고 합시다. 잔여 네트워크 G_f에 값이 $|f'| - |f|$인 실현 가능한 (s,t)-흐름이 존재함을 증명하십시오.

2. 임의의 흐름 네트워크 G의 임의의 간선을 $u \to v$라고 합시다. $u \in S$이고 $v \in T$인 최소 (s,t)-컷 (S,T)가 존재한다면, $u \in T'$이고 $v \in S'$인 최소 컷 (S',T')은 *존재하지 않음*을 증명하십시오.

3. 어떤 흐름 네트워크 G에서 (S,T)와 (S',T')가 최소 (s,t)-컷이라고 합시다. $(S \cap S', T \cup T')$와 $(S \cup S', T \cap T')$ 역시 G의 최소 (s,t)-컷임을 증명하십시오.

4. G를 서로 반대 방향인 간선 쌍 $u \to v$와 $v \to u$를 포함하고 두 간선의 용량이 모두 양수인 흐름 네트워크라고 합시다. G'을 G에서 이 두 간선의 용량을 모두 $\min\{c(u \to v), c(v \to u)\}$만큼 줄여서 얻은 흐름 네트워크라고 합시다. 다시 말하면 다음과 같습니다.

   - $c(u \to v) > c(v \to u)$이면 $u \to v$의 용량을 $c(u \to v) - c(v \to u)$로 바꾸고 $v \to u$를 삭제합니다.
   - $c(u \to v) < c(v \to u)$이면 $v \to u$의 용량을 $c(v \to u) - c(u \to v)$로 바꾸고 $u \to v$를 삭제합니다.
   - 마지막으로, $c(u \to v) = c(v \to u)$이면 $u \to v$와 $v \to u$를 모두 삭제합니다.

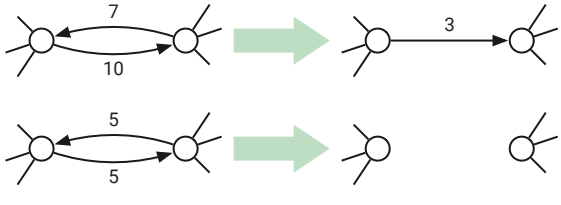


그림 10.11. 단방향 가정 강제하기

(a) G'의 모든 최대 (s,t)-흐름은 G에서도 최대 (s,t)-흐름임을 증명하십시오. (따라서 G에서 서로 반대 방향인 간선 쌍을 모두 단순화하면, G와 최대 흐름 값이 같은 새로운 축소된 흐름 네트워크를 얻습니다.)

(b) G의 모든 최소 (s,t)-컷은 G'에서도 최소 (s,t)-컷이며, *그 역도 성립함*을 증명하십시오.

(c) G의 최대 (s,t)-흐름 중 G'에서는 최대 (s,t)-흐름이 ***아닌*** 것이 적어도 하나 존재함을 증명하십시오.

5. (a) 주어진 흐름 네트워크에 *유일한* 최대 (s,t)-흐름이 존재하는지 판별하는 효율적인 알고리즘을 설명하십시오.

(b) 주어진 흐름 네트워크에 *유일한* 최소 (s,t)-컷이 존재하는지 판별하는 효율적인 알고리즘을 설명하십시오.

(c) 최대 (s,t)-흐름은 유일하지만 최소 (s,t)-컷은 유일하지 않은 흐름 네트워크를 제시하십시오.

(d) 최소 (s,t)-컷은 유일하지만 최대 (s,t)-흐름은 유일하지 않은 흐름 네트워크를 제시하십시오.

6. 네트워크 G의 (s,t)-흐름에서 모든 간선의 흐름 값이 양수인 방향 사이클이 존재하지 않으면, 그 흐름을 *비순환*이라고 합니다. 즉, 흐름 값이 양수인 간선들로 이루어진 부분 그래프가 dag입니다.

(a) 주어진 흐름 네트워크에서 *비순환* 최대 (s,t)-흐름을 계산하는 알고리즘을 설명하고 분석하십시오. 여러분의 알고리즘은 포드-풀커슨과 같은 점근적 실행 시간을 가져야 합니다.

(b) 주어진 흐름 네트워크에서 *모든* 최대 (s,t)-흐름이 비순환인지 판별하는 알고리즘을 설명하고 분석하십시오.

7. $G = (V, E)$가 모든 간선의 용량이 1이고 s에서 t까지의 최단 경로 거리가 d 이상인 흐름 네트워크라고 합시다.

(a) 최대 (s,t)-흐름의 값이 E/d 이하임을 증명하십시오.

(b) 이제 G가 *단순*하다고, 즉 모든 정점 u와 v에 대해 u에서 v로 가는 간선이 최대 하나뿐이라고 가정합시다. (흐름 네트워크에는 평행 간선이 있을 수 있습니다.) 최대 (s,t)-흐름의 값이 $O(V^2/d^2)$ 이하임을 증명하십시오. *[힌트: s를 루트로 하는 BFS 트리의 평균적인 레벨에는 노드가 몇 개 있습니까?]*

8. 모든 간선의 용량이 1인 흐름 네트워크 $G = (V, E)$와 정수 k가 주어졌다고 가정합시다. G에서 k개의 간선을 삭제한 뒤 남은 그래프에서 최대 (s,t)-흐름의 값이 가능한 한 작아지도록 하는 k개의 간선을 찾는 알고리즘을 설명하고 분석하십시오.

9. 흐름 분해 정리(Flow Decomposition Theorem)의 증명에서 수행한 분석은 더 정밀하게 다듬을 수 있습니다. $G = (V, E)$를 임의의 흐름 네트워크라 하고, f를 G에서의 임의의 (s, t)-흐름이라 합시다.

   (a) $|f| = 0$이면 f가 최대 $E - V + 1$개의 방향 사이클의 가중합으로 나타남을 증명하십시오. 이때 각 사이클의 모든 간선 e에 대해 $f(e) > 0$입니다.

   (b) $|f| > 0$이면, f는 최대 $E - V + 2$개의 방향 경로와 방향 사이클의 가중합이고, 이때 이 경로와 사이클 각각에 속한 모든 간선 e에 대해 $f(e) > 0$임을 증명하십시오.

   (c) 앞의 두 상한이 모두 빈틈없음(tight)을 증명하십시오. 즉, 어떤 순환(circulation)은 $E - V + 1$개보다 적은 사이클로 분해할 수 없고, 어떤 흐름은 $E - V + 2$개보다 적은 경로와 사이클로 분해할 수 없는 그래프가 존재합니다. *[힌트: 이것은 쉽습니다.]*

♣10. (s, t)-흐름들의 임의의 선형 결합이 그 자체로 (s, t)-흐름이라는 관찰로부터, 임의의 그래프에서 (반드시 실현 가능하지는 않은) 모든 (s, t)-흐름의 집합이 실제로 실수 *벡터 공간*을 이룬다는 사실을 알 수 있습니다. 이 공간을 그래프의 ***흐름 공간***이라고 부를 수 있습니다.

   (a) 임의의 연결 그래프 $G = (V, E)$의 흐름 공간의 차원이 $E - V + 2$임을 증명하십시오.

   (b) T를 G의 임의의 신장 트리라고 합시다. 다음 경로와 사이클의 모음이 흐름 공간의 기저를 이룸을 증명하십시오:

   - T에서 s부터 t까지의 유일한 경로;
   - 모든 간선 $e \notin T$에 대해, $T \cup \{e\}$에 있는 유일한 사이클.

   (c) G의 임의의 신장 트리를 T라 하고, T에서 임의의 간선 하나를 삭제하여 얻은 숲을 F라 합시다. 다음 경로와 사이클의 모음이 흐름 공간의 기저를 이룸을 증명하십시오:

   - F의 두 컴포넌트에 끝점을 하나씩 두는 모든 간선 $e \notin F$에 대하여, $F \cup \{e\}$에서 s부터 t까지 이어지는 유일한 경로;
   - 두 끝점이 모두 F의 같은 컴포넌트에 속하는 모든 간선 $e \notin F$에 대해, $F \cup \{e\}$ 안의 유일한 사이클입니다.

   (d) 다음 주장을 증명하거나 반증하십시오: 연결된 모든 흐름 네트워크에는 s에서 t로 가는 단순 경로만으로 이루어진 흐름 기저가 존재합니다.

11. 컷은 그래프 정점들의 분할이 아니라 간선들의 부분집합으로 정의되기도 합니다. 이 문제에서는 이 두 정의가 *거의* 동등함을 증명합니다.

   s에서 t로 가는 모든 방향 경로가 X에 속한 (방향) 간선을 적어도 하나 포함하면, (방향) 간선의 부분집합 X가 *s와 t를 분리한다*고 말합니다. *정점*의 임의

의 부분집합 S에 대해, S를 떠나는 방향 간선의 집합을 δS로 나타냅시다. 즉, $\delta S := \{u \to v \mid u \in S, v \notin S\}$입니다.

(a) (S, T)가 (s, t)-컷이면 δS가 s와 t를 분리함을 증명하십시오.

(b) X를 s와 t를 분리하는 간선들의 임의의 부분집합이라고 합시다. $\delta S \subseteq X$를 만족하는 (s, t)-컷 (S, T)가 존재함을 증명하십시오.

(c) s와 t를 분리하는 간선들의 극소 부분집합을 X라고 합시다. (이러한 간선 집합을 **본드**라고 부르기도 합니다.) $\delta S = X$를 만족하는 (s, t)-컷 (S, T)가 존재함을 증명하십시오.

12. 용량 대신, 각 간선 $u \to v$가 음이 아닌 ***요구량*** $d(u \to v)$를 갖는 네트워크를 생각한다고 가정합시다. 이제 (s, t)-흐름 f는 모든 간선 $u \to v$에 대해 $f(u \to v) \geq d(u \to v)$일 때, 그리고 오직 그때에만 *실현 가능*합니다. (이제 실현 가능한 흐름 값은 임의로 커질 수 있습니다.) 이러한 설정에서 자연스럽게 떠오르는 문제는 값이 *최소*인 실현 가능한 (s, t)-흐름을 찾는 것입니다.

(a) 그래프, 수요 함수, 정점 s와 t가 입력으로 주어질 때, 실행 가능한 (s, t)-흐름을 계산하는 효율적인 알고리즘을 설명하십시오. *[힌트: 모든 곳에서 0이 아닌 흐름을 찾은 다음, 그 흐름의 크기를 키워 실행 가능하게 만드십시오.]*

(b) 간선 용량이 있는 네트워크에서 *최대* 흐름을 계산하는 서브루틴 MaxFlow를 사용할 수 있다고 가정합시다. 간선 수요가 있는 네트워크가 주어졌을 때 *최소* 흐름을 계산하는 효율적인 알고리즘을 설명하십시오. 이 알고리즘은 MaxFlow를 정확히 한 번만 호출해야 합니다.

(c) 이 설정에 대해 최대 흐름-최소 컷 정리와 유사한 정리를 서술하고 증명하십시오. (최소 흐름은 최대 컷에 대응합니까?)

13. 임의의 흐름 네트워크 G와 임의의 정점 u, v에 대해, $bottleneck_G(u, v)$가 G에서 u로부터 v로 가는 모든 경로 π에 대하여 π 위의 최소 용량 간선이 갖는 용량의 최댓값을 나타낸다고 합시다.

(a) $bottleneck_G(s, t)$를 $O(E \log V)$ 시간에 계산하는 알고리즘을 기술하고 분석하십시오. 이 값은 에드먼즈-카프(Edmonds-Karp)의 최대 폭 증가 경로(fattest augmenting paths) 알고리즘이 첫 번째 반복에서 흘려보내는 흐름의 양입니다.

(b) 이제 흐름 네트워크 G가 무방향이라고 가정합시다. 이는 모든 정점 쌍 u와 v에 대해 $c(u \to v) = c(v \to u)$라고 가정하는 것과 같습니다. $bottleneck_G(s, t)$를 $O(V + E)$ 시간에 계산하는 알고리즘을 설명하고 분석하십시오. *[힌트: 간선 용량의 중앙값을 찾으십시오.]* 이 속도 향상이 방향 그래프에서는 통하지 않는 이유는 무엇입니까?

♥(c) 다시 한 번 흐름 네트워크 G가 무방향 그래프라고 가정합시다. 모든 정점 u와 v에 대해 $bottleneck_T(u, v) = bottleneck_G(u, v)$를 만족하는 G의 신장 트리 T를 구성하는 알고리즘을 기술하고 분석하십시오. (T의 간선은 G의

용량을 그대로 물려받습니다.) 만점을 받으려면 알고리즘이 $O(E)$ 시간에 실행되어야 합니다.

14. ***정수*** 간선 용량을 가진 흐름 네트워크 G와 G의 ***정수*** 최대 흐름 f^*가 주어졌다고 가정합시다. 다음 연산들을 수행하는 알고리즘을 설명하십시오:

    (a) INCREMENT(e): 간선 e의 용량을 1만큼 증가시키고 최대 흐름을 갱신합니다.

    (b) DECREMENT(e): 간선 e의 용량을 1만큼 줄이고 최대 흐름을 갱신합니다.

    두 알고리즘 모두 최대 흐름을 처음부터 다시 계산하는 것보다 더 빠르게, f^*가 여전히 최대 흐름이 되도록 f^*를 수정해야 합니다.

15. G를 간선 용량이 정수인 네트워크라고 합시다. G의 어떤 간선의 용량을 1만큼 늘렸을 때 G의 최대 흐름 값도 증가하면, 그 간선을 *상한 구속(upper-binding)* 간선이라고 합니다. 마찬가지로 어떤 간선의 용량을 1만큼 줄였을 때 G의 최대 흐름 값도 감소하면, 그 간선을 *하한 구속(lower-binding)* 간선이라고 합니다.

    (a) 모든 네트워크 G에는 상향 구속(upper-binding) 간선이 적어도 하나 존재합니까? 답이 옳음을 증명하십시오.

    (b) 모든 네트워크 G에는 하한 구속 간선(lower-binding edge)이 적어도 하나 존재합니까? 답이 옳음을 증명하십시오.

    (c) G와 G의 최대 흐름이 모두 입력으로 주어졌을 때, G의 모든 상한 결속 간선(upper-binding edge)을 $O(E)$ 시간에 찾는 알고리즘을 설명하십시오.

    (d) G와 G의 최대 흐름이 모두 입력으로 주어질 때, G의 모든 하향 구속 간선을 $O(EV)$ 시간에 찾는 알고리즘을 서술하십시오.

16. 주어진 흐름 네트워크 G에는 최소 (s,t)-컷이 둘 이상 있을 수 있습니다. S에서 T로 건너가는 간선의 수가 가장 적은 최소 컷 (S,T)를 ***최선의*** 최소 (s,t)-컷이라고 정의합시다.

    (a) 용량이 정수일 때 최선의 최소 (s,t)-컷을 찾는 효율적인 알고리즘을 설명하십시오.

    (b) *임의의* 간선 용량에 대해 최선의 최소 (s,t)-컷을 찾는 효율적인 알고리즘을 설명하십시오.

    (c) 주어진 흐름 네트워크에 유일한 *최선의* 최소 (s,t)-컷이 존재하는지 판별하는 효율적인 알고리즘을 설명하십시오.

17. 최대 흐름을 처음 가르치는 신임 조교수가 일반적인 포드-풀커슨 증가 경로 알고리즘을 다음과 같이 탐욕적으로 수정하자고 제안합니다. 잔여 그래프를 유지하는

대신, 그냥[5] 증가 경로를 따라 있는 간선들의 용량을 줄이십시오! 특히 간선을 포화시킬 때마다 그냥 그 간선을 그래프에서 제거하십시오. 그 잔여 그래프 같은 헛소리가 누구에게 필요하겠습니까?

```
GreedyFlow(G, c, s, t):
  for every edge e in G
      f(e) ← 0
  while there is a path from s to t
      π ← an arbitrary path from s to t
      F ← minimum capacity of any edge in π
      for every edge e in π
          f(e) ← f(e) + F
          if c(e) = F
              remove e from G
          else
              c(e) ← c(e) − F
  return f
```

(a) GreedyFlow가 항상 최대 흐름을 계산하지는 않음을 보이십시오.

(b) GreedyFlow가 최대 흐름의 좋은 근삿값을 계산한다는 보장조차 없음을 보이십시오. 즉, 임의의 상수 $\alpha > 1$에 대해, 최대 흐름의 값이 GreedyFlow가 계산한 흐름의 값의 α배보다 큰 흐름 네트워크 G가 존재합니다. *[힌트: GreedyFlow가 매 반복마다 가능한 최악의 경로 π를 선택한다고 가정하십시오.]*

18. 1980년 Maurice Queyranne은 아래에 보인 흐름 네트워크의 예를 발표했는데, 이 예에서는 에드먼즈와 카프의 "가장 굵은 경로" 휴리스틱이 멈추지 않습니다. 원래의 포드-풀커슨 알고리즘에 대한 Zwick의 나쁜 예에서와 마찬가지로, ϕ는 역황금비 $(\sqrt{5}-1)/2$를 나타냅니다. 세 개의 수직 간선은 Zwick의 예에서 수평 간선이 하는 것과 본질적으로 같은 역할을 합니다.

(a) 다음의 무한한 경로 증가 수열이 에드먼즈-카프 "가장 굵은 경로" 알고리즘의 올바른 실행임을 보이십시오. (그림 10.12의 아래쪽을 참고하십시오.)

```
QueyranneFatPaths:
  for i ← 1 to ∞
      push φ^(3i−2) units of flow along s→a→f→g→b→h→c→d→t
      push φ^(3i−1) units of flow along s→f→a→b→g→h→c→t
      push φ^(3i)   units of flow along s→e→f→a→g→b→c→h→t
```

(b) Queyranne의 네트워크에서 최대 흐름을 얻을 수 있는 $O(1)$번의 경로 증강 순서를 설명하십시오.

[5] 부사 *그냥*은 거의 언제나 "세부 사항을 알아내기에는 너무 귀찮지만, 그래도 제 말을 믿으십시오"를 무의식적으로 줄여 말한 것이며, 더 간결하게 말하면 "이것은 아마 틀렸습니다"라는 뜻입니다. *단지*, *간단히*, *분명히*, *자명하게*도 참고하십시오.

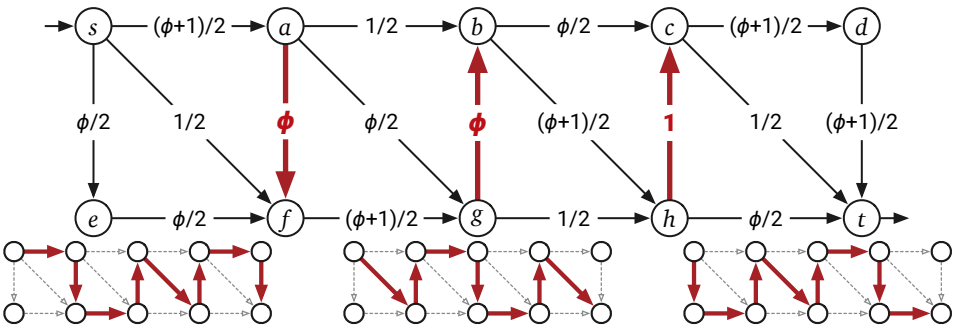


그림 10.12. Queyranne의 네트워크와 일련의 "가장 굵은 경로" 증가.

♥19. (s,t)-***직렬-병렬*** 그래프는 구별되는 두 정점 s와 t를 가지며 다음 구조 중 하나를 갖는 방향 비순환 그래프입니다.

- **기저 사례:** s에서 t로 가는 방향 간선 하나입니다.
- **직렬:** 공통 정점 u 하나만 공유하고 그 밖의 정점이나 간선은 공유하지 않는 (s,u)-직렬-병렬 그래프와 (u,t)-직렬-병렬 그래프의 합집합입니다.
- **병렬:** 소스 s와 대상 t가 같지만 그 밖에는 공통 정점이나 간선이 없는 더 작은 두 (s,t)-직렬-병렬 그래프의 합집합입니다.

모든 (s,t)-직병렬 그래프 G는 ***분해 트리***로 나타낼 수 있습니다. 분해 트리는 세 종류의 노드를 가지는 이진 트리이며, 그 세 종류는 잎(G의 간선에 대응합니다), 직렬 노드(s와 t를 제외한 정점에 대응합니다), 그리고 병렬 노드입니다. 같은 직병렬 그래프를 서로 다른 여러 분해 트리로 나타낼 수도 있습니다.

(a) 특별한 두 정점 s와 t가 있는 방향 그래프 G가 주어졌다고 합시다. G의 분해 트리를 구성하거나, G가 (s,t)-직렬-병렬이 아님을 올바르게 보고하는 알고리즘을 설명하고 분석하십시오. *[힌트: 트리를 아래에서부터 위로 구성하십시오.]*

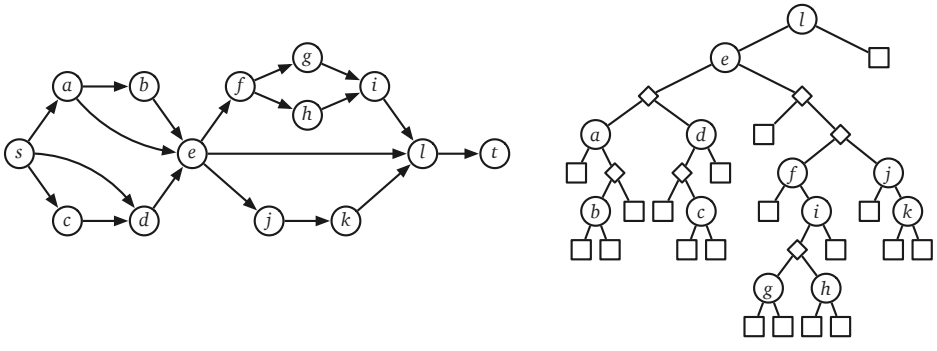


그림 10.13. 직렬-병렬 그래프와 이에 대응하는 분해 트리입니다. 분해 트리에서 정사각형은 리프이고, 마름모는 병렬 노드입니다.

(b) 임의의 간선 용량을 갖는 (s,t)-직렬-병렬 흐름 네트워크가 주어졌을 때, 그 네트워크에서 최대 (s,t)-흐름을 계산하는 알고리즘을 설명하고 분석하십시오. *[힌트: (a)의 결과에 비추어, 분해 트리가 실제로 주어진다고 가정해도 됩니다. 먼저 최대 흐름 값을 계산한 다음, 실제 최대 흐름을 계산하십시오.]*

20. 적어도 용량이 작은 정수인 네트워크에서는 다음 증가 경로에 대한 요구 조건을 완화하여 에드먼즈-카프 "가장 굵은 경로" 알고리즘의 속도를 높일 수 있습니다. 병목 용량이 최대인 증가 경로를 찾는 대신, 다음의 ***용량 스케일링*** 알고리즘을 사용하여 병목 용량이 최댓값의 절반 이상인 경로를 찾습니다. (이 알고리즘은 실제로 에드먼즈와 카프가 제안한 것입니다.)

모든 간선 용량이 어떤 정수 k에 대해 $U = 2^k$보다 작은 양의 정수라고 가정합니다. 스케일링 알고리즘은 병목 임계값 Δ를 유지하며, 처음에는 $\Delta \leftarrow U$로 설정합니다. 각 *단계*에서 알고리즘은 모든 간선의 잔여 용량이 Δ 이상인 s에서 t로 가는 경로를 따라 흐름을 증가시킵니다. 그러한 경로가 없으면 해당 단계가 끝나고, $\Delta \leftarrow \lfloor \Delta/2 \rfloor$로 설정한 뒤 다음 단계가 시작됩니다. 알고리즘은 $\Delta = 0$이 되면 종료됩니다.

(a) 최악의 경우 이 알고리즘은 몇 단계를 실행합니까?

(b) 특정한 Δ 값에 대한 한 단계가 끝났을 때의 흐름을 f라고 합시다. 잔여 그래프 G_f에서 최소 컷의 용량이 $E \cdot \Delta$ 이하임을 증명하십시오.

(c) 스케일링 알고리즘의 각 단계에서 증가가 최대 $2E$번 일어남을 증명하십시오.

(d) 용량 스케일링 알고리즘의 전체 실행 시간은 얼마입니까?

저는 오랫동안 그토록 비싸고 그토록 최첨단인 물건이 어떻게 그렇게까지 쓸모없을 수 있는지 의아했습니다. 그러다 문득 컴퓨터는 믿기 어려울 만큼 똑똑한 일을 해낼 능력을 지닌 멍청한 기계이고, 컴퓨터 프로그래머는 믿기 어려울 만큼 멍청한 일을 저지를 능력을 지닌 똑똑한 사람들이라는 사실을 깨달았습니다. 한마디로 그들은 완벽한 짝입니다.

— Bill Bryson, *Notes from a Big Country* (1999)

1990년 "철의 장막"이 무너진 직후, 둘 다 무기 개발에 종사했던 미국인과 러시아인이 만났습니다. 미국인이 물었습니다. "폭탄을 개발할 때, 그렇게 성능이 낮은 컴퓨터로 어떻게 그토록 엄청난 양의 계산을 해낼 수 있었습니까?" 러시아인이 대답했습니다. "우리는 더 나은 알고리즘을 사용했습니다."

— Yefim Dinitz, "Dinitz' Algorithm: The Original Version and Even's Version" (2006)에서

그래서 듣자 하니 이제 제가 악당이라고 합니다. 제가 흐름을 따르지 않기 때문이랍니다. 절대 흐름을 따르지 마십시오. 스스로 흐름이 되십시오.

— Jay-Z, "Stream of Consciousness" , May 16, 2015

11

흐름과 컷의 응용

11.1 간선 서로소 경로

최대 흐름의 가장 쉬운 응용 가운데 하나는 최대 흐름을 이용하여 방향 그래프 G에서 지정된 두 정점 s와 t 사이의 간선 서로소 경로의 최대 개수를 계산하는 것입니다. G의 각 간선이 경로들 중 많아야 하나에만 나타나면 G의 경로 집합을 *간선 서로소*라고 합니다. 다만 여러 간선 서로소 경로가 같은 정점을 지날 수는 있습니다.

각 간선에 용량 1을 부여하면, s에서 t로 가는 최대 흐름은 각 간선을 따라 0 또는 1 단위의 흐름을 보냅니다. 흐름 분해 정리에 따르면 포화된 간선들로 이루어진 부분 그래프 S는 여러 간선 서로소 경로와 사이클의 합집합입니다. 게다가 이 분해에 포함된 경로의 개수는 흐름 값과 정확히 같습니다. S에서 실제 경로를 추출하는 것은 간단합니다. S에서 s부터 t까지 임의의 방향 경로를 따라간 다음, 그 경로를 S에서 제거하고 재귀하면 됩니다.

반대로, k개의 간선 서로소 경로로 이루어진 임의의 모음은 각 경로를 따라 s에서 t로 한 단위의 흐름을 밀어 보냄으로써 흐름으로 변환할 수 있으며, 그 결과로 얻는 흐름의 값은 정확히 k입니다. 따라서 어떤 최대 흐름 알고리즘이든 실제로는 가능한 가장 큰 간선 서로소 경로 집합을 계산합니다.

Orlin의 알고리즘을 사용하여 최대 흐름을 계산하면 간선 서로소 경로를 $O(VE)$ 시간에 계산할 수 있지만, 이처럼 단순한 응용에 Orlin의 알고리즘을 쓰는 것은 지나칩니다. 컷 $(\{s\}, V \setminus \{s\})$의 용량은 최대 $V-1$이므로, 최대 흐름의 값은 최대 $V-1$입니다. 따라서 포드와 풀커슨의 원래 증가 경로 알고리즘만으로도 이미 $O(|f^*|E) = \boldsymbol{O(VE)}$ *시간*에 실행됩니다.

같은 알고리즘을 *무방향* 그래프에서 간선 서로소 경로를 찾는 데에도 사용할 수 있습니다. 먼저 G의 모든 무방향 간선 uv를 각각 단위 용량을 갖는 한 쌍의 방향 간선 $u \to v$와 $v \to u$로 바꾸고, 그 결과로 얻은 방향 그래프를 G'이라고 합시다. 다음으로 포드-풀커슨을 사용하여 G'에서 최대 (s,t)-흐름 f^*를 계산합니다. G의 임의의 간선 uv에 대해 f^*가 G'의 두 방향 간선 $u \to v$와 $v \to u$를 모두 포화시킨다면, 흐름 값을 바꾸지 않고 두 간선을 모두 흐름에서 제거할 수 있습니다. (더 일반적으로, 길이가 2인 사이클뿐만 아니라 f^*의 모든 사이클을 상쇄하여 G'에서 *비순환* 최대 흐름을 찾을 수 있습니다.) 따라서 일반성을 잃지 않고 f^*가 포화된 각 간선에 고유한 방향을 부여한다고 가정할 수 있습니다. 마지막으로 f^*에 의해 포화된 방향 간선들로 이루어진 부분 그래프를 탐색하여 간선 서로소 경로를 추출할 수 있습니다.

11.2 정점 용량과 정점 서로소 경로

이제 입력 그래프 G의 간선뿐만 아니라 정점에도 용량이 있다고 가정합시다. 다른 제약 조건에 더해, s와 t를 제외한 각 정점 v에 대해 v로 들어오는 흐름의 총량(따라서 v에서 나가는 흐름의 총량)이 어떤 음이 아닌 값 $c(v)$ 이하이도록 요구합니다.

$$\sum_{u \to v} f(u \to v) \le c(v).$$

이러한 새로운 정점 제약 조건이 있어도 여전히 최대 흐름을 계산할 수 있습니까?

1962년에 포드와 풀커슨은 간선 용량만 있는 흐름 네트워크 $\bar{G}$로의 다음과 같은 환원을 제안했습니다. 모든 정점 v를 두 정점 v_{in}과 v_{out}으로 바꾸고 이 둘을 용량이 $c(v)$인 간선 $v_{\text{in}} \to v_{\text{out}}$으로 연결한 다음, 모든 방향 간선 $u \to v$를 간선 $u_{\text{out}} \to v_{\text{in}}$으로 바꿉니다(용량은 그대로 유지합니다). 정의를 차근차근 따라가 보면 $\bar{G}$의 모든 실현 가능한 $(s_{\text{out}}, t_{\text{in}})$-흐름은 원래 그래프 G에서 같은 값을 갖는 실현 가능한 (s,t)-흐름과 동등하며, 그 역도 성립함을 알 수 있습니다. 특히 $\bar{G}$의 모든 최대 흐름은 G의 최대 흐름과 동등합니다. G에서 $\bar{G}$로의 환원에는 $O(E)$ 시간이 걸리며, 그 후 Orlin의 알고리즘을 사용하여 $\bar{G}$의 최대 흐름을 계산할 수 있습니다. 종합하면, G의 최대 흐름을 계산하는 데 $\boldsymbol{O(VE)}$ *시간*이 필요합니다.

이제 s에서 t로 가는 *정점*-서로소 경로의 최대 개수를 $O(VE)$ 시간에 쉽게 계산할 수 있습니다. 모든 정점에 용량 1을 부여하고 최대 흐름을 계산하면 됩니다!

11.3 이분 매칭

최대 흐름의 또 다른 자연스러운 응용은 이분 그래프에서 최대 ***매칭***을 찾는 것입니다. 매칭이란 모든 정점의 차수가 최대 1인 부분 그래프이며, 동치로 말하면 어느 두 간선도

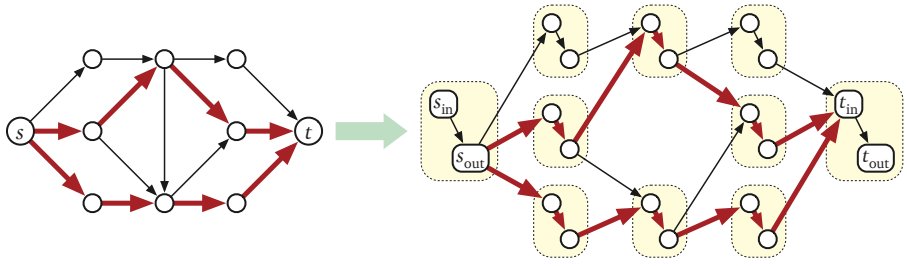


그림 11.1. G의 정점 서로소 경로를 $\bar{G}$의 간선 서로소 경로로 환원

정점을 공유하지 않는 간선들의 모임입니다. 여기서 풀고자 하는 문제는 간선의 수가 최대인 매칭을 찾는 것입니다.

예를 들어, 일자리를 구하는 의사들의 집합과 의사를 구하는 병원들의 집합이 있다고 가정합시다. 각 의사는 자신이 근무할 의향이 있는 병원을 모두 나열하고, 각 병원은 채용할 의향이 있는 의사를 모두 나열합니다. 우리의 과제는 모든 사람이 받아들일 수 있는 의사-병원 채용의 가장 큰 부분집합을 찾는 것입니다.[1] 이 문제는 의사와 병원을 정점으로 하고, 의사와 병원이 서로를 받아들일 수 있다고 여길 때에만 그 둘 사이에 간선이 있는 이분 그래프에서 최대 매칭을 찾는 것과 동치입니다.

이 문제는 다음과 같이 최대 흐름 문제로 환원하여 풀 수 있습니다. G를 정점 집합이 $L \cup R$인 주어진 이분 그래프라고 하고, 모든 간선은 L의 정점과 R의 정점을 잇는다고 합시다. 새로운 *방향* 그래프 G'은 다음 세 단계로 만듭니다. (1) 각 간선의 방향을 L에서 R로 정하고, (2) L의 모든 정점으로 가는 간선을 가진 새로운 소스 정점 s를 추가하며, (3) R의 모든 정점에서 오는 간선을 가진 새로운 대상 정점 t를 추가합니다. 마지막으로 G'의 모든 간선에 용량 1을 부여합니다.

G의 임의의 매칭 M은 다음과 같이 G'의 흐름 f_M으로 변환할 수 있습니다. M의 각 간선 uw에 대해 경로 $s \to u \to w \to t$를 따라 한 단위의 흐름을 보냅니다. 이 경로들은 s와 t를 제외하면 서로 겹치지 않으므로, 그 결과로 얻는 흐름은 용량 제약 조건을 만족합니다. 또한 그 결과로 얻는 흐름의 값은 M에 속한 간선의 개수와 같습니다.

반대로, 포드-풀커슨 증가 경로 알고리즘을 사용하여 계산한 G'의 임의의 (s,t)-흐름 f를 생각해 봅시다. 간선 용량이 정수이므로 포드-풀커슨 알고리즘은 모든 간선에 정수 흐름을 할당합니다. (이는 귀납법으로 쉽게 확인할 수 있습니다. 힌트, 힌트.) 또한 각 간선의 용량이 *단위* 용량이므로, 계산된 흐름은 G'의 모든 간선을 포화시키거나($f(e) = 1$) 피합니다($f(e) = 0$). 마지막으로, U의 어떤 정점에도 최대 한 단위의 흐름만 들어갈 수 있고 W의 어떤 정점에서도 최대 한 단위의 흐름만 나올 수 있으므로, U에서 W로 가는 포화된 간선들은 G에서 매칭을 이룹니다. 이 매칭의 크기는 정확히 $|f|$입니다.

따라서 G의 최대 매칭의 크기는 G'의 최대 흐름 값과 같으며, 증가 경로를 사용하여 최대 흐름을 계산한다면 실제 최대 흐름을 $O(E)$ 시간에 최대 매칭으로 변환할 수 있습니다. 최대 흐름은 Orlin의 알고리즘이나 기성 포드-풀커슨을 사용하여 $\boldsymbol{O(VE)}$ *시간*에 계산할 수 있습니다.

[1] 이 문제는 4장에서 살펴본 안정 매칭 문제와는 매우 다릅니다. 여기서는 각 의사와 병원을 가능한 한

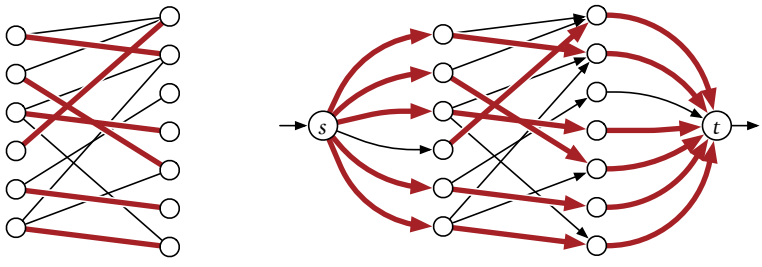


그림 11.2. 이분 그래프 G의 최대 매칭과, 이에 대응하는 G'의 최대 흐름입니다.

원래의 이분 그래프 G의 관점에서 G'에서의 포드-풀커슨의 동작을 해석해 보면 많은 것을 깨달을 수 있습니다. 이 알고리즘은 G에서 매칭 M을 유지하며, M은 처음에 비어 있습니다. M의 간선은 G'에서 흐름을 운반하는 간선에 대응합니다. G의 정점이 M에 속한 어떤 간선의 끝점이면 *매칭된* 정점이라 하고, 그렇지 않으면 *매칭되지 않은* 정점이라 합니다. 알고리즘의 각 반복에서 우리는 G에서 *교대 경로*를 찾습니다. 교대 경로란 L의 매칭되지 않은 정점에서 R의 매칭되지 않은 정점으로 가는 경로로서, M에 속한 간선과 M에 속하지 않은 간선을 번갈아 지나는 경로입니다. (G의 교대 경로는 G'에서 현재 흐름에 대한 증가 경로와 정확히 대응합니다.) 증가 경로 P를 찾으면 M을 대칭차 $M \oplus P$로 갱신하는데, 이렇게 하면 M의 간선 수가 1만큼 늘어납니다. 그런 다음 다음 반복으로 넘어갑니다. 교대 경로가 없으면 최대 흐름-최소 컷 정리에 의해 M이 최대 매칭이므로 알고리즘이 종료됩니다. 교대 경로 하나를 찾는 데 $O(E)$ 시간이 걸리고 알고리즘은 최대 V번 반복한 뒤 멈추므로, 전체 알고리즘은 $\boldsymbol{O(VE)}$ *시간*에 실행됩니다. 그림 11.3은 이 알고리즘이 실제로 동작하는 모습을 보여 줍니다.

교대 경로를 이용해 최대 이분 매칭을 특성화하는 이 결과는 1957년 클로드 베르주(Claude Berge)가 (최대 흐름-최소 컷 정리와는 독립적으로) 증명하였습니다. 하지만 이 특성화는 1955년 해럴드 쿤(Harald Kuhn), 1916년 데네시 쾨니그(Dénes Kőnig), 1836년경 카를 야코비(Carl Jacobi)가 기술한 알고리즘에 이미 암묵적으로 담겨 있었습니다.

1973년에 존 홉크로프트(John Hopcroft)와 리처드 카프가 제안한 더 정교한 알고리즘은 각 반복에서 여러 개의 서로소 교대 경로를 찾음으로써 이분 그래프의 최대 매칭을 단 $O(\sqrt{V}E)$ 시간에 계산합니다.

11.4 튜플 선택

이분 최대 매칭 문제는 제가 ***튜플 선택***이라고 부르는 더 넓은 문제 부류의 가장 단순한 예입니다.[2] 튜플 선택 문제의 입력은 여러 개의 유한 집합 $X_1, X_2, \ldots, X_d$로 이루어지며, 각 집합은 서로 다른 이산 자원을 나타냅니다. 우리가 할 일은 다음과 같은 형태의

만족시키려 하지 않기 때문입니다.

[2] 이러한 문제들을 가리키는 표준 명칭을 찾지 못하여 직접 이름을 지었습니다. 이 문제들은 때때로 "할당 문제"라고도 불리지만, "할당 문제(the assignment problem)"라는 표현은 간선에 가중치가 있는 이분 그래프에서 최대 *가중치* 이분 매칭을 찾는 문제를 가리키는 경우가 더 일반적입니다.

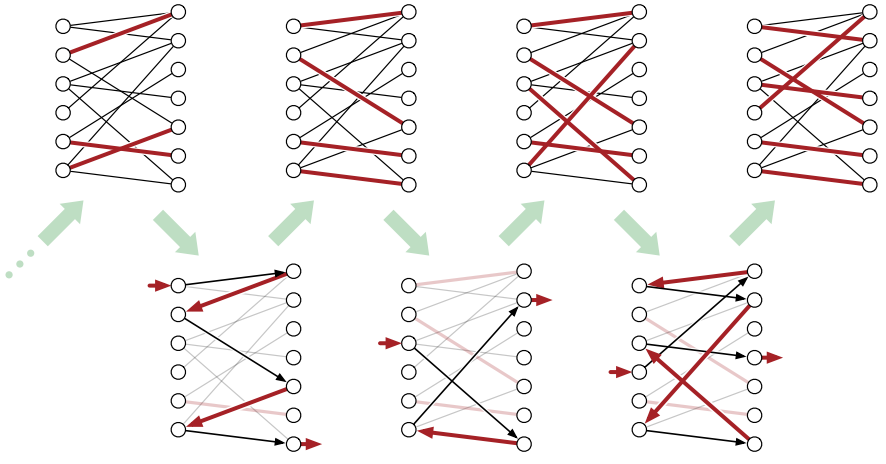


그림 11.3. 교대 경로로 연결된, 크기가 점점 커지는 매칭들의 수열입니다.

여러 *용량* 제약 조건을 만족하면서, 각 집합 X_i에서 정확히 하나씩의 원소를 포함하는 d-튜플들의 집합을 가능한 한 크게 선택하는 것입니다:

- 각 인덱스 i에 대해, 각 원소 $x \in X_i$는 선택된 튜플 중 최대 $c(x)$개에 나타날 수 있습니다.
- 각 인덱스 i에 대해, 임의의 두 원소 $x \in X_i$와 $y \in X_{i+1}$는 선택된 튜플 중 최대 $c(x, y)$개에만 함께 나타날 수 있습니다.

상한 $c(x)$와 $c(x, y)$는 각각 (대개 작은) 음이 아닌 정수이거나 ∞입니다.

최대 매칭 문제에서는 자원이 $d = 2$개이고, 각 원소 x의 용량은 $c(x) = 1$이며, 각 쌍 (x, y)의 용량은 기저 이분 그래프에서 xy가 간선인지 아닌지에 따라 $c(x, y) = 1$ 또는 $c(x, y) = 0$입니다.

자원들이 선형으로 순서가 정해져 있고 *인접한* 부분집합 X_i와 X_{i+1}에 속한 객체 쌍들에만 제약이 걸리므로,[3] 튜플 선택 문제는 다음과 같이 정의되는 방향 그래프 G에서의 최대 흐름 문제로 환원할 수 있습니다:

- G에는 각 집합 X_i의 각 원소마다 하나의 정점이 있으며, 소스 정점 s와 대상 정점 t도 있습니다. (s와 t를 제외한) 각 정점 x의 용량은 $c(x)$입니다.
- G에는 각 원소 $w \in X_1$마다 간선 $s \to w$가 있고, 각 원소 $z \in X_d$마다 간선 $z \to t$가 있으며, 모든 i에 대해 $x \in X_i$이고 $y \in X_{i+1}$인 각 원소 쌍마다 용량이 $c(x, y)$인 간선 $x \to y$가 있습니다. (필요하다면 $c(x, y) = 0$인 간선 $x \to y$는 생략해도 됩니다.)

G에서 s부터 t까지의 모든 경로는 우리가 선택*할 수 있는* d-튜플 하나에 대응합니다 (혹은 그 튜플 "자체입니다"). 반대로, 주어진 제약 조건을 만족하면서 선택할 수 있는

[3] 인접하지 않은 부분집합 쌍($j > i + 1$인 X_i와 X_j)이 단 하나라도 있어 그 쌍에서 나온 객체 쌍들에도 제약이 걸린다면, Exact-3DimensionalMatching으로부터의 간단한 환원에 의해 이 문제는 NP-난해가 됩니다. NP-난해성은 다음 장에서 다루겠습니다.

d-튜플은 모두 G에서 s부터 t까지의 경로 하나에 대응합니다(혹은 그 경로 "자체입니다").

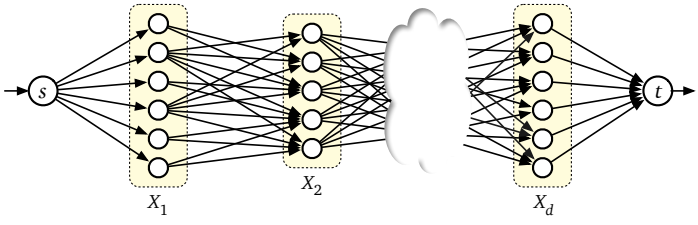


그림 11.4. 튜플 선택 문제를 위한 흐름 네트워크입니다.

더 일반적으로, f를 G에서의 임의의 실현 가능한 정수 (s,t)-흐름이라고 합시다. 모든 용량이 정수이거나 ∞이므로, 흐름 분해 정리에 의해 f는 s에서 t로 가는 $|f|$개의 경로의 합이며, 각 경로는 정확히 한 단위의 흐름을 운반합니다. 정의를 차근차근 따라가 보면 이렇게 얻은 튜플 집합이 모든 용량 제약 조건을 만족함을 알 수 있습니다. 역으로, 용량 제약 조건을 만족하는 임의의 k개 튜플 집합에 대해, 대응하는 k개 경로의 합은 값이 k인 실현 가능한 정수 (s,t)-흐름입니다.

따라서 G에서 최대 (s,t)-흐름 f^*를 계산한 다음 f^*의 흐름 분해를 계산하면, 주어진 용량 제약 조건을 만족하는 튜플을 최대 개수로 선택할 수 있습니다. G의 유한한 용량은 모두 정수이므로, 일반성을 잃지 않고 f^*가 정수 흐름이라고 가정할 수 있으며, 그러므로 (앞 문단에 의해) f^*는 $|f^*|$개의 튜플로 이루어진 유효한 집합에 대응합니다.

시험 일정 잡기

다음의 "실세계" 스케줄링 문제는 앞에서 설명한 일반적인 환원을 더 명확히 이해하는 데 도움이 될 수 있습니다.

Sham-Poobanana 대학교에서 기말시험 일정을 짜는 알고리즘을 작성해 달라며 여러분을 고용했습니다. 서로 다른 n개의 강좌가 있으며, 각 강좌는 t개의 서로 다른 시간대 중 하나에 r개의 강의실 중 하나에서 기말시험을 치르도록 일정을 잡아야 합니다. 각 시간대에 각 강의실에는 많아야 한 강좌의 기말시험만 배정할 수 있습니다. 반대로 한 강좌를 여러 강의실이나 여러 시간대로 나눌 수는 없습니다. 또한 각 시험은 p명의 시험 감독관 중 한 명이 감독해야 합니다.[4] 각 감독관은 한 번에 많아야 하나의 시험만 감독할 수 있고, 특정 시간대에만 감독할 수 있으며, 어떤 감독관도 총 5개를 넘는 시험을 감독할 수 없습니다. 이 스케줄링 문제의 입력은 다음 세 배열로 이루어집니다.

- $E[i]$가 i번째 수업에 등록한 학생 수를 나타내는 정수 배열 $E[1..n]$.
- 정수 배열 $S[1..r]$이 주어지며, 여기서 $S[j]$는 j번째 강의실의 좌석 수입니다. i번째 수업의 기말고사는 $E[i] \le S[j]$일 때, 그리고 오직 그때에만 j번째 강의실에서 치를 수 *있습니다*.

[4] 미국 밖에서는 흔히 invigilator라고 더 잘 알려져 있습니다

- 불리언 배열 $A[1..t, 1..p]$로, ℓ번째 감독관이 k번째 시간대에 감독할 수 있을 때, 그리고 그때에만 $A[k, \ell] = \textsc{True}$입니다.[5]

$N = n + r + tp$를 입력의 전체 크기라고 합시다. 여러분이 할 일은 모든 수업의 기말고사에 강의실, 시간대, 감독관을 배정하거나, 그러한 일정이 불가능하다고 올바르게 보고하는 알고리즘을 설계하는 것입니다.

이것은 수업, 강의실, 시간대, 감독관이라는 네 가지 자원을 갖는 표준적인 튜플 선택 문제입니다. 이 문제를 풀기 위해 그림 11.5에 나타낸 것처럼 여섯 종류의 정점—소스 정점 s', 각 수업에 대한 정점 c_i, 각 강의실에 대한 정점 r_j, 각 시간대에 대한 정점 t_k, 각 감독관에 대한 정점 p_ℓ, 그리고 대상 정점 t'—과 다섯 종류의 간선을 갖는 흐름 네트워크 G를 구성합니다.

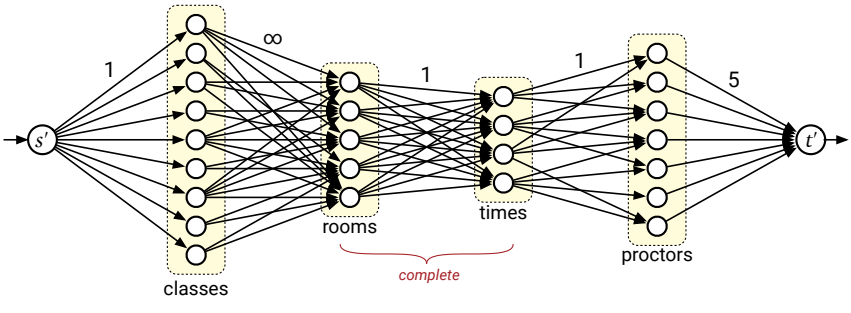


그림 11.5. 시험 스케줄링 문제를 위한 흐름 네트워크.

- 각 수업 i마다 용량이 1인 간선 $s' \to c_i$를 하나씩 둡니다. ("각 수업은 기말시험을 최대 한 번만 치를 수 있습니다.")
- $E[i] \le S[j]$를 만족하는 각 수업 i와 강의실 j에 대해, 용량이 ∞인 간선 $c_i \to r_j$. ("수업 i는 강의실 j에 좌석이 충분할 때, 그리고 오직 그때에만 강의실 j에서 시험을 치를 수 있습니다.") 수강 인원 $E[i]$와 좌석 수 $S[j]$가 사용되는 곳은 이곳뿐입니다.
- 각 강의실 j와 시간대 k마다 용량이 1인 간선 $r_j \to t_k$를 하나씩 둡니다. ("시간 k에 강의실 j에서는 시험을 최대 하나만 치를 수 있습니다.")
- $A[\ell, k] = \textsc{True}$인 시간대 k와 감독관 ℓ마다 용량이 1인 간선 $t_k \to p_\ell$. ("감독관은 어느 시점에든 최대 한 개의 시험만 감독할 수 있으며, 그것도 자신이 감독할 수 있는 시간에만 감독할 수 있습니다.")
- 각 감독관 ℓ마다 용량이 5인 간선 $p_\ell \to t'$입니다. ("각 감독관은 최대 5개의 시험을 감독할 수 있습니다.")

(소스 정점과 대상 정점을 s와 t 대신 s'과 t'으로 부르는 이유는 단지 문제 설명에서 이미 변수 t를 시간대의 개수를 나타내는 데 사용하고 있기 때문입니다.) 전체적으로

[5] 이 정보는 그래프로 표현하는 편이 더 낫다고 볼 수도 있지만, 그렇게 하면 환원이 더 헷갈릴 것이라고 생각했습니다.

G는 $n + r + t + p + 2 = O(N)$개의 정점과 $O(nr + rt + tp) = O(N^2)$개의 간선을 가집니다.

G에서 s'에서 t'로 가는 각 경로는 기말시험 하나에 대한 과목, 강의실, 시간, 감독관의 유효한 선택 하나를 유일하게 나타냅니다. 구체적으로 말하면, 해당 과목은 그 강의실에 수용될 수 있고, 감독관은 그 시간에 감독할 수 있습니다. 역으로, 각각의 유효한 선택 (과목, 강의실, 시간, 감독관)에 대해 G에는 그에 대응하는 s'에서 t'로 가는 경로가 존재합니다. 따라서 G에서 최대 (s', t')-흐름 f^*를 계산하고, f^*를 s'에서 t'로 가는 경로들로 분해한 다음, 각 경로를 과목-강의실-시간-감독관 배정으로 옮겨 적으면 가능한 최대 개수의 시험에 대한 유효한 일정을 구성할 수 있습니다. $|f^*| = n$이면 그렇게 얻은 일정을 반환하면 되고, 그렇지 않으면 n개의 기말시험을 모두 배정하는 것이 불가능하다고 올바르게 보고할 수 있습니다.

주어진 입력 데이터로부터 무차별 대입으로 G를 구성하는 데는 $O(E)$ 시간이 걸립니다. 최대 흐름은 포드-풀커슨($|f^*| \leq n < V$이기 때문입니다) 또는 Orlin의 알고리즘을 사용하여 $O(VE)$ 시간에 계산할 수 있으며, 흐름 분해는 $O(VE)$ 시간에 계산할 수 있습니다. 따라서 전체 알고리즘은 $O(VE) = \boldsymbol{O(N^3)}$ *시간*에 실행됩니다.

11.5 서로소 경로 덮개

방향 그래프 G의 ***경로 덮개***란 G의 모든 정점이 *적어도* 하나의 경로 위에 놓이도록 하는 G 안의 방향 경로들의 모음입니다. G의 ***서로소*** 경로 덮개란 G의 모든 정점이 *정확히* 하나의 경로 위에 놓이는 경로 덮개입니다. 모든 방향 그래프에는 길이가 0인 경로 여러 개로 이루어진 자명한 서로소 경로 덮개가 있지만, 이것은 지루합니다. 대신 경로를 가능한 한 적게 포함하는 서로소 경로 덮개를 찾아봅시다. 이 문제는 일반적으로 NP-난해합니다(그래프가 크기 1인 서로소 경로 덮개를 가질 필요충분조건은 그 그래프가 해밀턴 경로를 포함하는 것입니다). 하지만 방향 *비순환* 그래프에 대해서는 흐름에 기반한 효율적인 알고리즘이 있습니다.

주어진 방향 비순환 그래프 $G = (V, E)$에 대해 이 문제를 풀기 위해, 다음과 같이 새로운 이분 그래프 $G' = (V', E')$를 구성합니다.

- G의 모든 정점 v마다 H에는 두 정점 $v^\flat$과 $v^\sharp$가 있습니다.
- H는 G의 모든 방향 간선 $u \to v$에 대해 무방향 간선 $u^\flat v^\sharp$를 포함합니다.

(G가 인접 행렬로 표현되어 있다면, G'은 같은 인접 행렬로 표현되는 이분 그래프입니다!)

이제 G를 k개의 서로소 경로로 덮을 수 있는 것은 새 그래프 G'에 크기가 $V - k$인 매칭이 있을 때, 그리고 그때에만 성립한다고 주장합니다. 늘 그렇듯이, 이 동치 관계를 두 단계로 증명합니다:

⇐ G에 경로 k개로 이루어진 서로소 경로 덮개 P가 있다고 가정하고, P를 G의 부분 그래프로 생각합시다. P의 모든 정점은 진입 차수가 0 또는 1이며, 게다가 P의 각 경로에는 진입 차수가 0인 정점이 정확히 하나씩 있습니다. 따라서 P에는 간선이 정확히 $V - k$개 있습니다. 이제 G'의 간선들의 부분집합 M을 다음과 같이 정의합

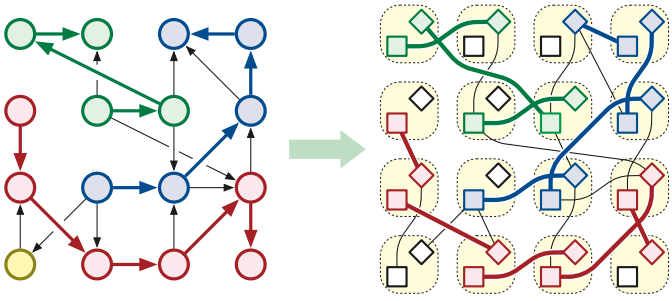


그림 11.6. dag의 최소 서로소 경로 덮개를 최대 이분 매칭으로 환원합니다. 사각형은 플랫$^\flat$이고 마름모는 샤프$^\sharp$입니다.

니다.

$$M := \left\{ u^\flat v^\sharp \in E' \;\middle|\; u \to v \in P \right\}.$$

서로소 경로 덮개의 정의에 따라 G의 모든 정점은 P에 속한 들어오는 간선을 최대 하나, P에 속한 나가는 간선을 최대 하나 가집니다. 따라서 G'의 모든 정점은 M의 간선 중 최대 하나에만 인접합니다. 즉, M은 크기가 $V - k$인 매칭입니다.

⇒ G'에 크기가 $V - k$인 매칭 M이 있다고 가정합시다. 부분 그래프 $P = (V, M')$를 다음과 같이 정의하여 M'을 G로 다시 사영합니다. 여기서

$$M' := \left\{ u \to v \in E \;\middle|\; u^\flat v^\sharp \in M \right\}.$$

매칭의 정의에 의해 G의 모든 정점은 P에서 들어오는 간선을 최대 하나, P에서 나가는 간선을 최대 하나 가집니다. 따라서 P는 G에서 서로소인 방향 경로들의 모음입니다. P가 모든 정점을 포함하므로 P는 간선이 $V - k$개인 서로소 경로 덮개를 정의합니다. P에 있는 경로의 수는 G의 정점 중 M'에서 들어오는 간선이 없는 정점의 수와 같습니다. 그러므로 P는 정확히 k개의 경로를 포함한다는 결론을 얻습니다.

따라서 포드-풀커슨의 최대 흐름 알고리즘으로 G'에서 최대 매칭을 계산하면 G의 최소 서로소 경로 덮개를 $O(V'E') = \boldsymbol{O(VE)}$ *시간*에 찾을 수 있음이 바로 도출됩니다.

dag와 경로라는 용어로 정식화되어 있기는 하지만, 이 문제는 사실 최대 매칭 문제입니다. 우리는 가능한 한 많은 정점을 그래프에서 서로 다른 *후속 정점*에 *매칭*하고자 합니다. dag를 덮는 데 필요한 경로의 수는 후속 정점이 없는 정점의 수와 같습니다. (그리고 물론 모든 이분 최대 매칭 문제는 사실 흐름 문제입니다.)

최소 교수 채용

또 하나의 "현실 세계" 스케줄링 문제를 위해 Sham-Poobanana 대학교로 다시 가봅시다.[6] SPU는 매일 수천 개의 강의를 개설합니다. 극심한 예산 삭감 때문에 대학은 교수진의 규모를 대폭 줄여야 합니다. 그러나 학생들이 등록금을 내기 때문에(그리고

[6] 이 문제를 조금 더 현실적으로(그리고 덜 우울하게) 정식화하려면 교수와 강의 대신 비행기와 항공편, 또는 버스와 버스 노선을 생각해 보십시오.

대학은 변호사를 고용할 여유가 없기 때문에), 대학은 강의 편람에 공지된 모든 강의가 실제로 진행되도록 보장할 만큼 충분한 교수를 남겨 두어야 합니다. SPU는 최소 몇 명의 교수만으로 버틸 수 있을까요? 남은 교수진 각자에게는 하루마다 가르칠 강의들의 수열이 배정됩니다. 각 교수에게 배정된 강의들은 서로 겹쳐서는 안 됩니다. 게다가 각 교수의 일정에는 한 강의에서 다음 강의로 걸어서 이동할 수 있을 만큼의 여유가 있어야 합니다. 이 문제에서는 모든 교수가 모든 강의를 가르칠 수 있으며, 교수들에게는 면담 시간도, 점심 시간도, 화장실에 갈 휴식 시간도 없다고 가정합시다.[7]

구체적으로, m개의 서로 다른 장소에서 n개의 강좌가 개설된다고 가정합시다. 이 문제의 입력은 다음 데이터로 구성됩니다.

- 수업들의 배열 $C[1..n]$입니다. 각 수업 $C[i]$는 시작 시각 $C[i].start$, 종료 시각 $C[i].end$, 장소 $C[i].loc$의 세 필드를 가집니다.
- 2차원 배열 $T[1..m, 1..m]$. 여기서 $T[u, v]$는 위치 u에서 위치 v까지 걸어가는 데 필요한 시간입니다.

모든 수업을 함께 가르칠 수 있는 교수의 최소 인원을 구하려고 합니다. 이때 한 교수가 $C[j].start \geq C[i].start$인 두 수업 i와 j를 맡도록 배정될 때마다 실제로 다음이 성립해야 합니다.

$$C[j].start \;\geq\; C[i].end + T\big[C[i].loc,\, C[j].loc\big].$$

이 문제는 다음과 같이 서로소 경로 덮개 문제로 환원하여 풀 수 있습니다. 정점이 수업이고, 간선이 같은 교수가 가르칠 수 있을 만큼 충분히 시간 간격을 두고 배정된 수업 쌍을 나타내는 dag $G = (V, E)$를 구성합니다. 구체적으로, 방향 간선 $i \to j$는 같은 교수가 수업 i를 가르친 다음 수업 j를 가르칠 수 있음을 나타냅니다. 이 dag는 무차별 대입으로 $O(n^2)$ 시간에 쉽게 구성할 수 있습니다. 그런 다음 앞에서 설명한 매칭 알고리즘을 사용하여 G의 서로소 경로 덮개를 찾습니다. G의 각 방향 경로는 한 교수에게 허용되는 수업 일정을 나타냅니다. 전체 알고리즘은 $O(n^2 + VE) = \boldsymbol{O(n^3)}$ *시간*에 실행됩니다.[8]

처음에는 구간과 거리로 설명되었지만, 이것은 사실 최대 매칭 문제입니다(이는 곧 *진짜로는* 사실 최대 흐름 문제라는 뜻입니다). 구체적으로 말하면, 우리는 가능한 한 많은 수업을 같은 교수가 가르치는 *다음* 수업과 *매칭*하고자 합니다. 필요한 교수의 수는 후속 수업이 배정되지 않은 수업의 수와 같습니다. 후속 수업이 배정되지 않은 각 수업은 어떤 교수가 가르치는 마지막 수업입니다.

[7] 그러나 강의 사이를 걸어서 이동하는 동안 학생들의 이메일에 답장하는 것은 기대됩니다.

[8] 모든 시간 구간 $T[u, v]$가 같다고 가정하면,[9] 이 스케줄링 문제는 실제로 간단한 탐욕 알고리즘을 사용하여 $O(n \log n)$ 시간에 풀 수 있습니다.

[9] 미국의 많은 대학은 같은 캠퍼스 안에서라면 사람이 어느 강의실에서든 다른 어느 강의실까지 10분 안에 걸어갈 수 있다는 놀라운 믿음 아래 수업 사이에 10분의 쉬는 시간을 배정합니다. 이 믿음이 현실적이라고 생각하는 분이 있다면 저희 캠퍼스를 방문하여 한쪽 Siebel Center에서 다른 쪽 Siebel Center까지 걸어 보시기를 권합니다.

11.6 야구 탈락 문제

매년 수백만 명의 미국 야구 팬들이 자신이 응원하는 팀이 플레이오프 진출권을 따내고 궁극적으로는 월드 시리즈에서 우승하기를 바라며 열렬히 경기를 지켜봅니다. 안타깝게도 대부분의 팀은 정규 시즌이 끝나기 며칠, 심지어 몇 주 전에 이미 "수학적으로 탈락"합니다. 어떤 팀이 탈락했는지는 쉽게 알아챌 수 있는 경우가 많습니다—그 팀은 소속 지구의 현재 선두를 따라잡을 만큼 충분히 많은 경기를 이길 수 없습니다. 그러나 때로는 상황이 더 미묘합니다. 예를 들어, 다음은 1996년 8월 30일 아메리칸 리그 동부 지구의 실제 순위입니다.

Team	**Won–Lost**	**Left**	NYY	BAL	BOS	TOR	DET
New York Yankees	75–59	28		3	8	7	3
Baltimore Orioles	71–63	28	3		2	7	4
Boston Red Sox	69–66	27	8	2		0	0
Toronto Blue Jays	63–72	27	7	7	0		0
Detroit Tigers	49–86	27	3	4	0	0	

디트로이트가 분명히 뒤처져 있지만, 일부 열성적인 타이거스 팬들은 자기 팀이 아직 우승할 수 있다는 희망을 놓지 않을지도 모릅니다. 따지고 보면 디트로이트가 남은 27경기를 모두 이기면 76승으로 시즌을 마치게 되며, 이는 현재 다른 어떤 팀의 승수보다도 많습니다. 그러니 다른 모든 팀이 모든 경기에서 지기만 한다면. . . 하지만 그것은 불가능합니다. 그 다른 팀들 중 일부는 아직 서로 경기를 치러야 하기 때문입니다. 다음은 하나의 완전한 논증입니다:[10]

> 디트로이트는 남은 경기를 모두 이기면 76승 86패로 시즌을 마칠 수 있습니다. 양키스가 단 2경기만 더 이기면 77승 85패로 시즌을 마치게 되어 디트로이트를 앞서게 됩니다. 그러므로 타이거스가 남은 시즌 동안 한 경기도 지지 않고 양키스는 더 이상 한 경기도 이기지 못한다고 가정해 봅시다.
>
> 이 시나리오의 문제는 뉴욕이 아직 보스턴과 8경기를 남겨 두고 있다는 점입니다. 레드삭스가 이 경기를 모두 이기면 최소 77승으로 시즌을 마치게 되어 타이거스를 앞서게 됩니다. 따라서 디트로이트가 1위로 시즌을 마칠 가능성이라도 가지려면, 뉴욕이 보스턴과의 8경기 중 정확히 한 경기만 이기고 나머지 경기는 모두 져야 합니다. 한편 레드삭스는 뉴욕 이외의 팀과 치르는 경기를 모두 져야 합니다. 그러면 세 팀이 공동 1위가 됩니다....
>
> 이제 이 시나리오에서 오리올스와 블루제이스에게 어떤 일이 일어나는지 살펴봅시다. 볼티모어는 보스턴과 2경기, 뉴욕과 3경기를 남겨 두고 있습니다. 따라서 모든 일이 앞서 설명한 대로 일어난다면 오리올스는 최소 76승으로 시즌을 마치게 됩니다. 그러므로 디트로이트가 볼티모어를 따라잡으려면 오리올스가 뉴욕과 보스턴을 제외한 팀들과의 경기를 모두 져야만 합니다. 특히 이는 볼티모어가 토론토와 남은 7경기를 모두 져야 한다는 뜻입니다. 블루제이스는 양키스와도 7경기를 남겨 두고 있으며, 디트로이트가 1위로 시즌을 마치려면

[10] 이 예시와 논증은 모두 Eli Olinick의 웹사이트 https://s2.smu.edu/~olinick/riot/detroit.html 에서 가져온 것이며, 이 웹사이트는 Olinick이 Ilan Adler, Alan Erera, Dorit Hochbaum과 함께 수행한 공동 연구를 바탕으로 합니다.

> 토론토가 이 경기들을 모두 이겨야 한다는 것을 이미 확인했습니다. 하지만 그렇게 되면 블루제이스는 최소 14승을 더 거두어 최종 성적이 77승 85패 이상이 되고, 이는 블루제이스가 타이거스보다 높은 순위로 시즌을 마친다는 뜻입니다. 따라서 시즌의 이 시점부터 무슨 일이 일어나든 디트로이트는 아메리칸 리그 동부 지구에서 1위로 시즌을 마칠 수 없습니다.

이것을 알아낼 더 좋은 방법이 분명히 있을 것입니다!

이 문제를 좀 더 추상적으로 정식화하면 다음과 같습니다. 입력은 두 배열 $W[1..n]$과 $G[1..n, 1..n]$으로 이루어지며, 여기서 $W[i]$는 팀 i가 이미 이긴 경기 수이고 $G[i, j]$는 팀 i와 팀 j 사이에 남은 경기 수입니다. 우리는 팀 n이 시즌을 최다승으로 (다른 팀과 공동일 수도 있습니다) 마칠 수 있는지 판별하고자 합니다.[11]

1960년대 중반에 Benjamin Schwartz는 이 질문을 최대 흐름 문제로 모델링할 수 있음을 발견했습니다. 그로부터 약 20년 후, Dan Gusfield, Charles Martel, David Fernández-Baca는 Schwartz의 흐름 정식화를 쌍 선택 문제로 단순화했습니다. 구체적으로 말하면, 우리는 팀 n이 1위를 차지하도록 *각 경기의 승자를* ***선택***하는 것이 가능한지 알고자 합니다. $R[i] = \sum_j G[i, j]$가 팀 i의 남은 경기 수를 나타낸다고 합시다. 팀 n은 남은 $R[n]$개의 경기를 모두 이긴다고 가정하겠습니다. 그러면 팀 n이 1위를 차지할 수 있는 것은 다른 모든 팀 i가 남은 $R[i]$개의 경기 중 최대 $W[n] + R[n] - W[i]$개를 이기는 경우이고, 오직 그 경우뿐입니다.

각 경기마다 이긴 팀을 ***선택***하려고 하므로, 먼저 경기와 팀을 노드로 나타내는 이분 그래프를 만듭니다. *경기* 노드 $g_{i,j}$는 각 쌍 $1 \le i < j < n$마다 하나씩 있으므로 모두 $\binom{n}{2}$개입니다. *팀* 노드 t_i는 각 $1 \le i < n$마다 하나씩 있으므로 모두 $n - 1$개입니다. 각 쌍 i, j에 대해 용량이 무한인 간선 $g_{i,j} \to t_i$와 $g_{i,j} \to t_j$를 추가합니다. 소스 정점 s를 추가하고, 각 쌍 i, j에 대해 용량이 $G[i, j]$인 간선 $s \to g_{i,j}$를 추가합니다. 마지막으로 대상 노드 t를 추가하고, 각 팀 i에 대해 용량이 $W[n] - W[i] + R[n]$인 간선 $t_i \to t$를 추가합니다.

그림 11.7은 1996년 아메리칸 리그 동부 지구 순위표에서 얻은 그래프를 보여 주며, 여기서 "팀 n"은 디트로이트 타이거스입니다. 레이블이 없는 모든 간선의 용량은 무한대입니다.

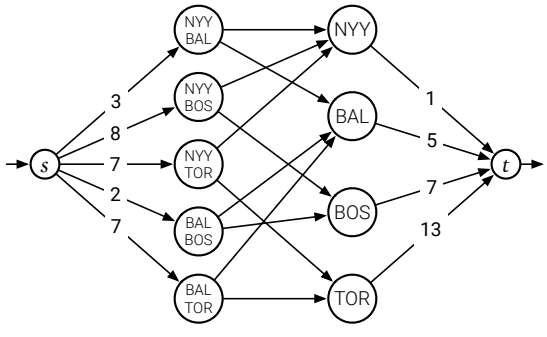


그림 11.7. 컵스가 이겼습니다! 컵스가 이겼습니다!

[11] 여기서는 무승부로 끝나는 경기가 없고 모든 경기가 실제로 치러진다고 암묵적으로 가정합니다. 전쟁이나 자연재해, 벌 떼의 습격 같은 일만 없다면, 두 가정 모두 적어도 포스트시즌 순위에 영향을 주는 경기에 대해서는 메이저리그 야구 규정과 부합합니다.

정리. *s에서 나가는 모든 간선을 포화시키는 실현 가능한 흐름이 이 그래프에 존재할 때, 그리고 오직 그때에만 팀 n은 시즌을 1위로 마칠 수 있습니다.*

증명: 팀 n이 시즌을 1위로 마칠 수 있다고 가정합시다. 그러면 모든 팀 $i < n$은 남은 경기 중 최대 $W[n] + R[n] - W[i]$경기를 이깁니다. 팀 i와 팀 j 사이의 경기 중 팀 i가 이기는 각 경기마다 경로 $s \to g_{i,j} \to t_i \to t$를 따라 흐름 한 단위를 추가합니다. 팀 i와 팀 j 사이에는 정확히 $G[i,j]$경기가 있으므로 s에서 나가는 모든 간선은 포화됩니다. 각 팀 i는 최대 $W[n] + R[n] - W[i]$경기를 이기므로 그 결과로 얻은 흐름은 실현 가능한 흐름입니다.

반대로, f가 s에서 나가는 모든 간선을 포화시키는 가능 흐름이라고 합시다. 모든 i와 j에 대해 팀 i가 팀 j를 상대로 정확히 $f(g_{i,j} \to t_i)$경기를 이긴다고 가정합시다. 그러면 팀 i와 팀 j는 $f(g_{i,j} \to t_i) + f(g_{i,j} \to t_j) = f(s \to g_{i,j}) = G[i,j]$경기를 치르므로, 남은 모든 경기가 치러집니다. 또한 각 팀 i는 남은 경기 중 총 $\sum_j f(g_{i,j} \to t_i) = f(t_i \to t) \leq W[n] + R[n] - W[i]$경기를 이기므로, 전체적으로는 많아야 $W[n] + R[n]$경기를 이깁니다. 따라서 팀 n이 남은 경기를 모두 이기면 시즌을 1위로 마칩니다. □

요약하면, 우리가 응원하는 팀이 우승할 수 있는지 판단하기 위해 흐름 네트워크를 구성하고 최대 흐름을 계산한 다음, 그 최대 흐름이 s에서 나가는 모든 간선을 포화시키는지를 보고합니다. 예를 들어 그림 11.7의 그래프에서 s를 떠나는 간선들의 총 용량은 27입니다(남은 경기가 27경기이기 때문입니다). 반면에 t로 들어가는 간선들의 총 용량은 26에 불과하며, 이는 최대 흐름 값이 최대 26이라는 것을 의미합니다. 따라서 디트로이트는 수학적으로 탈락했다고 결론지을 수 있습니다.[12]

이 흐름 네트워크는 $O(n^2)$개의 정점과 $O(n^2)$개의 간선을 가지며, $O(n^2)$ 시간에 구성할 수 있습니다. Orlin의 알고리즘을 사용하면 최대 흐름을 $O(VE) = \boldsymbol{O(n^4)}$ ***시간***에 계산할 수 있습니다.

이것이 야구 탈락 문제를 푸는 가장 빠른 알고리즘은 아닙니다. 2001년에 Kevin Wayne은 사실상 단 한 번의 최대 흐름 계산만으로 수학적으로 탈락한 ***모든*** 팀을 단 $\boldsymbol{O(n^3)}$ ***시간***에 판별할 수 있음을 증명했습니다.

11.7 프로젝트 선택

마지막 예로, 수행할 수 있는 n개의 프로젝트로 이루어진 집합이 주어졌다고 가정합시다. 일부 프로젝트는 특정한 다른 프로젝트가 완료될 때까지 시작할 수 없습니다. 프로젝트와 그 의존 관계는 프로젝트를 정점으로 하는 방향 비순환 그래프 G로 기술되며, 각 간선 $u \to v$는 프로젝트 u를 프로젝트 v보다 먼저 수행할 수 없음을 나타냅니다. (이는 6.4장에서 살펴본 의존성 그래프와 정확히 같은 형태입니다.) 마지막으로, 각 프로젝트 v에는 그 프로젝트를 완료하면 받게 되는 *이익* $\$(v)$가 연관되어 있습니다. 일부 프로젝트는 이익이 음수이며, 이는 양수인 *비용*으로 해석합니다. 우리는 의존 대상을 모두 포함하는 프로젝트 부분집합 X라면 어느 것이든 골라 완료할 수 있습니다. 즉, 모든 프로젝트 $x \in X$에 대해 x가 의존하는 모든 프로젝트도 X에 속해야 합니다.

[12] 이 경우에는 운이 좋았습니다(혹은 나빴습니다). t로 들어가는 모든 간선의 총 용량이 s에서 나가는 간선들의 총 용량보다 작지 않더라도 팀이 탈락할 수 있습니다.

우리의 목표는 총 이익이 최대한 큰 유효한 프로젝트 부분집합을 찾는 것입니다. 특히 모든 작업의 이익이 음수라면 아무것도 하지 않는 것이 올바른 답입니다.

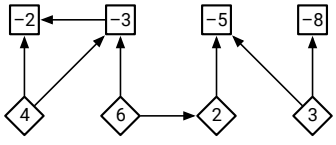


그림 11.8. 프로젝트 여덟 개로 이루어진 집합의 의존성 그래프입니다. 마름모는 수익이 나는 프로젝트를, 정사각형은 비용이 드는 프로젝트를 나타냅니다. 각 간선 $u \to v$는 u가 v에 의존한다는 뜻입니다.

큰 틀에서 보면, 우리의 과제는 프로젝트들을 두 부분집합 S와 T, 즉 우리가 *선택하는*(Select) 작업들과 *거절하는*(Turn down) 작업들로 분할하는 것입니다. 따라서 직관적으로는 이 문제를 어떤 그래프에서의 최소 컷 문제로 모델링하고 싶습니다. 그렇다면 어떤 그래프입니까? 선행 조건은 어떻게 강제합니까? 우리는 이익을 *최대화*하고 싶지만, 우리가 찾는 방법을 알고 있는 것은 *최소* 컷뿐입니다. 그리고 음의 이익을 양의 용량으로 어떻게 변환합니까?

주어진 제약 그래프 G를 흐름 네트워크 G'으로 변환하기 위해, 의존성 그래프에 소스 정점 s와 대상 정점 t를 추가합니다. 그리고 수익이 나는 모든 작업 v($\$(v) > 0$인 작업)에 대해 간선 $s \to v$를 추가하고, 비용이 드는 모든 작업 u($\$(u) < 0$인 작업)에 대해 간선 $u \to t$를 추가합니다. 직관적으로, s는 수익/비용이 0이고 마지막에 반드시 수행해야 하는 새로운 작업("잠자기!")으로 생각할 수 있습니다. G'의 간선에는 다음과 같이 용량을 할당합니다:

- 수익성 있는 모든 작업 v에 대해 $c(s \to v) = \$(v)$입니다;
- 비용이 드는 모든 작업 u에 대해 $c(u \to t) = -\$(u)$입니다;
- 모든 의존성 간선 $u \to v$에 대해 $c(u \to v) = \infty$입니다.

모든 간선 용량이 양수이므로, 이는 최대 컷 문제에 대한 유효한 입력입니다.

이제 G'에서 임의의 (s,t)-컷 (S,T)를 생각해 봅시다. 원래 의존성 그래프의 임의의 간선 $u \to v$에 대해 $u \in S$이고 $v \in T$이면 $\|S,T\| = \infty$입니다. 따라서 S에 속한 작업들을 적법하게 선택할 수 있는 것은 컷 (S,T)의 용량이 유한할 때, 그리고 오직 그때뿐입니다.

실제로 용량이 더 작은 컷일수록 더 높은 이익을 내는 작업 선택에 대응한다는 사실이 드러납니다. 구체적으로 말하면, S에 속한 작업들을 선택할 때 얻는 총이익은 $P - \|S,T\|$이며, 여기서 P는 모든 양의 이익의 합이라고 주장합니다:

$$P = \sum_v \max\{0, \$(v)\} = \sum_{\$(v)>0} \$(v).$$

이 주장은 다음과 같이 정의를 차례로 따라가는 간단한 방법으로 증명할 수 있습니다. 프로젝트들의 임의의 부분집합 X에 대해 세 가지 값을 정의합니다. (여기서도 늘 그

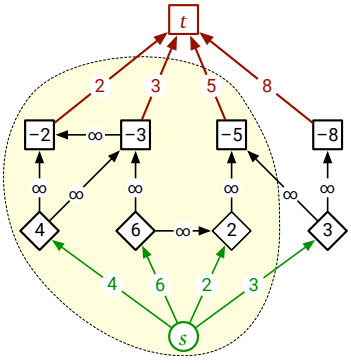


그림 11.9. 예시 의존성 그래프에 대한 흐름 네트워크와 그 최소 컷입니다. 이 컷의 용량은 13이고 $P = 15$이므로, 선택된 작업들의 총 이익은 2입니다.

렇듯이, $u \to v$가 간선이 아닐 때에는 $c(u \to v) = 0$으로 정의합니다.)

$$cost(X) := \sum_{\substack{u \in X \\ \$(u) < 0}} -\$(u) \;=\; \sum_{u \in X} c(u \to t)$$

$$yield(X) := \sum_{\substack{v \in X \\ \$(v) > 0}} \$(v) \;=\; \sum_{v \in X} c(s \to v)$$

$$profit(X) := \sum_{v \in X} \$(v) \;=\; yield(X) - cost(X).$$

정의에 따라 $P = yield(V) = yield(S) + yield(T)$입니다. 컷 (S, T)의 용량이 유한하므로 $s \to v$ 형태와 $u \to t$ 형태의 간선만 컷을 가로지를 수 있습니다. 구성 방식에 따라 모든 간선 $s \to v$는 수익성 있는 작업을 가리키고, 각 간선 $u \to t$는 비용이 드는 작업에서 나옵니다. 따라서 $\|S, T\| = cost(S) + yield(T)$입니다. 이로부터 곧바로 주장한 대로 $P - \|S, T\| = yield(S) - cost(S) = profit(S)$라는 결론을 얻습니다.

따라서 G'에서 *최소* 컷을 계산하면 총이익을 *최대화*할 수 있음을 곧바로 알 수 있습니다. G로부터 G'을 $O(V + E)$ 시간에 쉽게 구성할 수 있으며, Orlin의 알고리즘을 사용하면 G'에서 최소 (s, t)-컷을 $O(VE)$ 시간에 계산할 수 있습니다. 결론적으로 전체 프로젝트 선택 알고리즘은 ***O(VE) 시간***에 실행됩니다.

연습문제

1. $G = (V, E)$를 각 정점 v마다 v의 진입 차수와 진출 차수가 같은 방향 그래프라고 합시다. G가 어떤 정점 u에서 다른 정점 v로 가는 간선 서로소 경로를 k개 포함한다고 가정합시다. 이러한 조건에서 G는 v에서 u로 가는 간선 서로소 경로도 반드시 k개 포함해야 합니까? 증명을 제시하거나, 설명과 함께 반례를 제시하십시오.

2. 세 정점 u, v, w가 주어진 무방향 그래프 $G = (V, E)$에서, u에서 w로 가면서 v를 지나는 경로가 존재하는지 판별하는 알고리즘을 설명하고 분석하십시오. *[힌트: G가 방향 그래프였다면 이 문제는 NP-난해했을 것입니다!]*

3. 여러 개의 소스 정점 $s_1, s_2, \ldots, s_\sigma$와 대상 정점 $t_1, t_1, \ldots, t_\tau$를 갖는 방향 그래프 $G = (V, E)$를 생각해 봅시다. 이때 어떤 정점도 소스이면서 동시에 대상일 수는 없습니다. *다중 단자 흐름*이란 소스도 대상도 아닌 모든 정점에서 흐름 보존 제약 조건을 만족하는 함수 $f : E \to \mathbb{R}_{\geq 0}$입니다. 다중 단자 흐름의 값 $|f|$는 모든 소스 정점에서 나가는 초과 흐름의 총합입니다:

$$|f| := \sum_{i=1}^{\sigma} \left(\sum_{w} f(s_i \to w) - \sum_{u} f(u \to s_i) \right)$$

늘 그렇듯이, 우리는 간선의 용량 제약 조건을 만족하면서 값이 최대인 흐름을 찾는 데 관심이 있습니다. (특히, 어떤 특정 소스에서 어떤 특정 대상으로 흐름이 얼마나 이동하는지는 신경 쓰지 않습니다.)

(a) 다중 터미널 흐름을 계산하는 다음 알고리즘을 생각해 봅시다. 변수 f와 f'은 흐름 함수를 나타냅니다. 서브루틴 MaxFlow(G, s, t)는 소스가 s이고 대상이 t인 표준 최대 흐름 문제를 풉니다.

```
MaxMultiFlow(G, s[1..σ], t[1..τ]):
  f ← 0                          《흐름을 초기화하십시오》
  for i ← 1 to σ
      for j ← 1 to τ
          f' ← MaxFlow(G_f, s[i], t[j])
          f ← f + f'             《흐름을 갱신합니다》
  return f
```

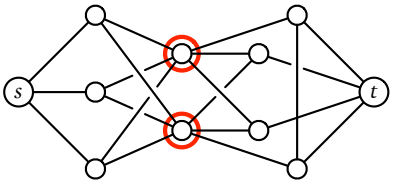
이 알고리즘이 G에서 최대 다중 터미널 흐름을 올바르게 계산함을 증명하십시오.

(b) G에서 최대 다중 터미널 흐름을 계산하는 더 효율적인 알고리즘을 설명하십시오.

4. 소도어 섬에는 방대한 철도망으로 연결된 수많은 마을과 촌락이 있습니다. 최근 스카를로이 마을에서 치명적인 전염병(돼지 독감인지 좀비인지는 보고가 분명하지 않습니다) 사례가 여러 건 보고되었습니다. 소도어 철도의 관리자는 이 병이 자신의 고향인 티드머스로 퍼지지 않도록 일부 기차역을 폐쇄할 계획입니다. 폐쇄된 역은 어떤 기차도 통과할 수 없습니다. 비용(그리고 대중의 이목)을 최소화하기 위해, 그는 가능한 한 적은 수의 역만 폐쇄하려고 합니다. 하지만 스카를로이 역은 폐쇄할 수 없습니다. 그렇게 하면 자신이 병에 노출되기 때문입니다. 티드머스 역도 폐쇄할 수 없습니다. 그러면 자신이 가장 좋아하는 펍에 갈 수 없기 때문입니다.

스카를로이에서 티드머스로 가는 모든 철도 이동을 차단하기 위해 폐쇄해야 하는 역의 최소 개수를 구하는 알고리즘을 설명하고 분석하십시오. 소도어 철도망은 무방향 그래프로 표현되며, 각 역마다 정점이 하나씩 있고 두 역 사이의 각 철도 연결마다 간선이 하나씩 있습니다. 두 개의 특별한 정점 s와 t는 각각 스카를로이와 티드머스에 있는 역을 나타냅니다.

예를 들어, 다음과 같은 입력 그래프가 주어지면 알고리즘은 정수 2를 반환해야 합니다.



5. $n \times n$ 격자는 n^2개의 정점이 n개의 행과 n개의 열로 배열된 무방향 그래프입니다. i번째 행과 j번째 열에 있는 정점을 (i,j)로 나타냅니다. $i = 1$, $i = n$, $j = 1$ 또는 $j = n$인 *경계* 정점을 제외하면, 모든 정점 (i,j)는 정확히 네 개의 이웃 $(i-1,j)$, $(i+1,j)$, $(i,j-1)$, $(i,j+1)$을 가집니다.

   $(x_1,y_1),(x_2,y_2),\ldots,(x_m,y_m)$을 $n \times n$ 격자의 서로 다른 정점이라고 하고, 이를 *터미널*이라고 부릅니다. ***탈출 문제***는 터미널들을 서로 다른 임의의 경계 정점 m개에 연결하는 정점 서로소 경로 m개가 격자에 존재하는지 판정하는 문제입니다.

   (a) 탈출 문제를 해결하는 효율적인 알고리즘을 서술하고 분석하십시오. 알고리즘의 실행 시간은 n에 대한 작은 차수의 다항 함수여야 합니다.

   (b) 이제 탈출 문제의 입력이 정수 하나 n과 m개의 말단 정점 목록으로 이루어져 있다고 가정합시다. m이 매우 작으면 앞의 실행 시간은 사실 입력 크기에 대해 *지수적*입니다! 탈출 문제를 *m에 대한 다항* 시간에 해결하는 알고리즘을 설명하고 분석하십시오.

   ♥(c) 탈출 경로가 존재하는 경우 그 경로를 명시적으로 기술하여 출력하도록 이전 알고리즘을 수정하십시오. 이때도 실행 시간은 여전히 m에 대한 다항 시간이어야 합니다.

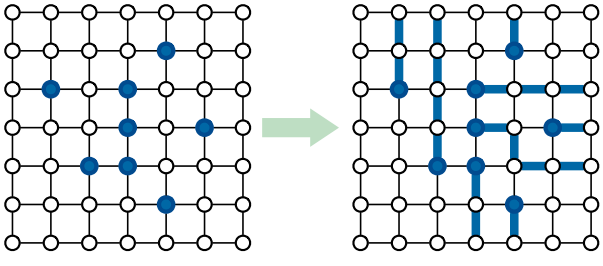


그림 11.10. 탈출 문제의 긍정 인스턴스와 그 해입니다.

6. SPU 통근자 정숙부가 See-Bull 센터 지하에 미니 골프 코스를 설치하고 있습니다! 경기장은 직각으로 만나는 m개의 수평 및 수직 선분으로 둘러싸인 닫힌 다각형입니다. 코스에는 n개의 *출발점*과 n개의 홀이 있으며, 이들은 일대일로 대응합니다. 각 출발점에서 대응하는 홀까지는 항상 경기장의 경계에 닿지 않고

직선을 따라 곧바로 공을 칠 수 있습니다. (선수들은 골프공을 벽에 튕겨서는 안 됩니다. 유리가 너무 많기 때문입니다.) n개의 출발점과 n개의 홀은 모두 서로 다른 위치에 있습니다.

안타깝게도 공사가 막 시작되려던 참에 건축가의 컴퓨터가 고장 나고 말았습니다. 시스템 관리자들의 초인적인 노력 덕분에 시작점과 구멍의 *위치*는 복구할 수 있었지만, 어느 시작점이 어느 구멍에 대응하는지에 관한 정보는 모두 사라져 버렸습니다!

직선 조건을 만족하는 시작점과 홀 사이의 일대일 대응을 계산하거나, 그러한 대응이 존재하지 않음을 보고하는 알고리즘을 설명하고 분석하십시오. 입력은 경기장의 꼭짓점 m개, 시작점 n개, 홀 n개의 x 좌표와 y 좌표로 이루어집니다. 두 선분의 끝점의 x 좌표와 y 좌표가 주어지면 두 선분이 교차하는지를 상수 시간에 판별할 수 있다고 가정하십시오.

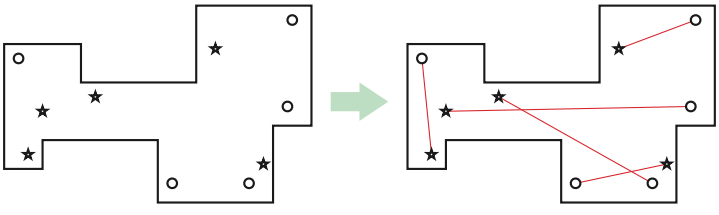


그림 11.11. 다섯 개의 출발점(⋆)과 홀(○)이 있는 미니골프 코스와, 이들 사이의 유효한 대응 관계입니다.

7. 주어진 방향 그래프 $G = (V, E)$의 ***사이클 덮개***란 G의 모든 정점을 덮는, 정점이 서로소인 사이클들의 집합입니다. 주어진 그래프의 사이클 덮개를 찾거나, 사이클 덮개가 존재하지 않음을 올바르게 보고하는 효율적인 알고리즘을 설명하고 분석하십시오. *[힌트: 이분 매칭을 사용하십시오!]*

8. 일부 칸이 삭제된 $n \times n$ 체커보드가 주어졌다고 가정합시다. 여러분에게는 체커보드의 두 칸을 덮기에 딱 맞는 크기의 도미노가 많이 있습니다. 이 도미노로 보드를 빈틈없이 덮을 수 있는지 판별하는 알고리즘을 기술하고 분석하십시오. 이때 각 도미노는 삭제되지 않은 칸을 정확히 두 개 덮어야 하고, 삭제되지 않은 각 칸은 정확히 하나의 도미노로 덮여야 합니다.

   입력은 불리언 배열 $Deleted[1..n, 1..n]$이며, i행 j열의 칸이 삭제되었을 때, 그리고 오직 그때에만 $Deleted[i, j] = \textsc{True}$입니다. 출력은 불리언 값 하나이며, 실제 도미노 배치를 계산할 필요는 ***없습니다***. 예를 들어 그림 11.12에 나온 판에 대해서는 알고리즘이 True를 반환해야 합니다.

9. $n \times n$ 정사각형 격자가 주어지고, 그 칸들 중 일부는 검은색으로, 나머지는 흰색으로 칠해져 있다고 합시다. 다음 조건을 만족하도록 격자 위에 토큰을 놓을 수 있는지 판정하는 알고리즘을 설명하고 분석하십시오:

   - 모든 토큰이 흰 칸 위에 있습니다.

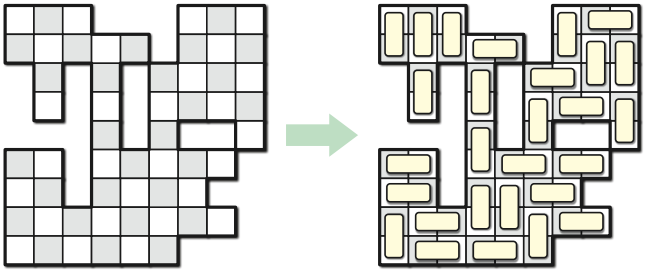


그림 11.12. 일부 칸이 빠진 체커판을 도미노로 덮기.

- 격자의 모든 행에는 정확히 하나의 토큰이 있으며,
- 격자의 모든 열에는 토큰이 정확히 하나씩 있습니다.

입력은 어느 칸이 흰색인지를 나타내는 불리언 값의 2차원 배열 $IsWhite[1..n, 1..n]$입니다. 출력은 불리언 값 하나입니다. 예를 들어 그림 11.13의 격자가 입력으로 주어지면 알고리즘은 TRUE를 반환해야 합니다.

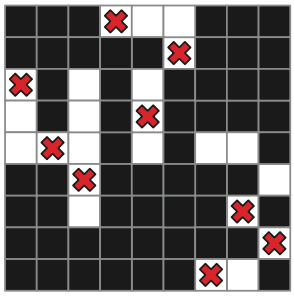


그림 11.13. 격자의 모든 행과 열 표시하기.

10. 각 상자의 높이, 너비, 깊이가 센티미터 단위로 주어진 상자들의 집합이 있다고 합시다. 모든 상자의 세 변의 길이는 모두 10cm보다 크고 20cm보다 작습니다. 예상하시다시피, 첫 번째 상자를 회전시켜 그 높이, 너비, 깊이가 각각 두 번째 상자의 높이, 너비, 깊이보다 작아지게 할 수 있다면 첫 번째 상자를 두 번째 상자 안에 넣을 수 있습니다. 상자는 재귀적으로 중첩될 수 있습니다. 어떤 상자가 다른 상자 안에 들어 있지 않으면 그 상자를 *보이는* 상자라고 부릅니다.

    보이는 상자의 개수가 가능한 한 적어지도록 상자들을 중첩하는 알고리즘을 설명하고 분석하십시오.

11. 일부 칸에 표시가 되어 있는 $n \times n$ 격자가 주어졌다고 가정합시다. 이 격자는 불리언 배열 $M[1..n, 1..n]$로 표현되며, 칸 (i, j)에 표시가 되어 있을 때 그리고 그때에만 $M[i, j] =$ TRUE입니다. 격자를 지나는 *단조* 경로는 왼쪽 위 칸에서 시작하여 매 단계마다 오른쪽이나 아래쪽으로만 이동하고, 오른쪽 아래 칸에서

끝납니다. 우리의 목표는 가능한 한 적은 수의 단조 경로로 표시된 칸을 모두 덮는 것입니다.

(a) 표시된 칸을 가장 많이 지나는 단조 경로를 찾는 알고리즘을 설명하십시오.

(b) 단조 경로들로 작은 덮개를 찾는 자연스러운 탐욕 휴리스틱이 있습니다. 표시된 칸이 남아 있으면 표시된 칸을 가장 많이 덮는 단조 경로 π를 찾고, π가 덮는 표시된 칸들의 표시를 모두 해제한 다음 재귀합니다. 이 알고리즘이 항상 최적해를 계산하지는 *않음*을 보이십시오.

(c) 표시된 모든 칸을 덮는 단조 경로들의 최소 집합을 계산하는 효율적인 알고리즘을 설명하고 분석하십시오.

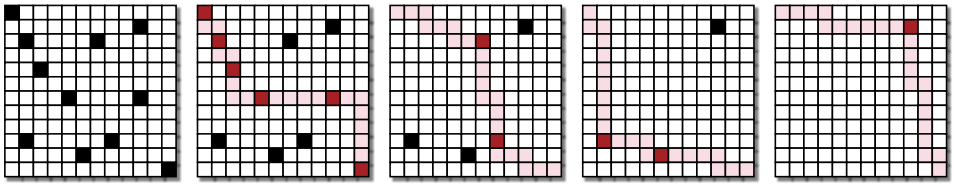


그림 11.14. 격자의 표시된 칸들을 네 개의 단조 경로로 탐욕적으로 덮기.

12. Sham-Poobanana 대학교 교수 평의회는 최근 불명예를 안은 팩토텀 남작을 대신하여 엉클 개비, 보보 코넬리어스 교수, 초능력 고릴라 모포 중 누구를 SPU 운동부(The Fighting Pooh-bahs)의 새 공식 마스코트 *상징*으로 삼을지 결정하기 위해 위원회를 소집하기로 했습니다. 이 위원회에서 활동할 교수는 각 학과에서 정확히 한 명씩 선출해야 합니다. 일부 교수는 여러 학과에 소속되어 있지만, 각 위원은 한 학과만 대표할 수 있습니다. 예를 들어, 블라고예비치 교수가 부패학과와 어리석음학과에 모두 소속되어 있고 어리석음학과 대표로 선출되었다면, 부패학과는 다른 사람이 대표해야 합니다. 마지막으로, 대학 규정에 따라 모든 교수 위원회에는 조교수, 부교수, 정교수가 정확히 같은 수만큼 있어야 합니다. 다행히 학과 수는 3의 배수입니다.

SPU 교수진 중 일부를 선발하여 포스트-팩토텀 유인원 마스코트 *상징* 위원회를 구성하거나, 유효한 위원회를 구성할 수 없음을 올바르게 보고하는 알고리즘을 설명하고 분석하십시오. 입력은 어느 교수가 어느 학과에 소속되어 있는지를 나타내는 이분 그래프이며, 각 교수 정점에는 해당 교수의 직급(조교수, 부교수, 정교수)이 표시되어 있습니다.

13. Sham-Poobanana 대학교의 통근자 침묵학과는 유연한 교육과정을 운영하며, 그 졸업 요건은 복잡합니다. 이 학과는 서로 다른 과목 n개를 개설하며, 서로 다른 요건이 m개 있습니다. 각 요건은 n개 과목의 부분집합 하나와 그 부분집합에서 이수해야 하는 과목 수를 지정합니다. 서로 다른 요건의 부분집합은 겹칠 수 있지만, 각 과목은 *최대 하나의* 요건을 충족하는 데에만 사용할 수 있습니다.

예를 들어, 과목 A, B, C, D, E가 $n = 5$개 있고 졸업 요건이 $m = 2$개 있다고 가정합시다.

- 부분집합 $\{A, B, C\}$에서 적어도 2개의 과목을 수강해야 합니다.
- 부분집합 $\{C, D, E\}$에서 최소 2개 과목을 수강해야 합니다.

그러면 B, C, D 과목만 수강한 학생은 졸업할 수 없지만, A, B, C, D 또는 B, C, D, E를 수강한 학생은 졸업할 수 있습니다.

주어진 학생이 졸업할 수 있는지 판정하는 알고리즘을 기술하고 분석하십시오. 알고리즘의 입력은 m개의 요건 목록(각 요건은 n개의 과목 중 일부로 이루어진 부분집합과 그 부분집합에서 수강해야 하는 과목 수를 지정합니다)과 그 학생이 수강한 과목 목록입니다.

14. 여러분은 금요일, 토요일, 일요일 사흘 내내 열리는 제1회 연례 SPU 통근자 침묵 72시간 댄스 교류회(First Annual SPU Commuter Silence 72-Hour Dance Exchange)를 준비하고 있습니다. 행사 기간 동안 30분짜리 음악 세트가 여러 번 재생될 예정이며, 많은 DJ가 공연을 신청했습니다. 여러분은 다음 제약 조건에 따라 DJ를 고용해야 합니다.

    - 매일 정확히 k개의 음악 세트를 연주해야 하므로, 모두 합쳐 $3k$개의 세트를 연주하게 됩니다.
    - 각 세트는 한 명의 DJ가 일관된 하나의 음악 장르(앰비언트, 버블검, 덥스텝, 호러코어, K-팝, 콰이토, 마리아치, 스트레이트어헤드 재즈, 트립합, 내슈빌 컨트리, 파라파라, 스카, …)로 연주해야 합니다.
    - 각 장르는 하루에 최대 한 번까지만 재생할 수 있습니다.
    - 각 DJ 후보는 자신이 기꺼이 틀 수 있는 장르의 목록을 여러분에게 제출했습니다.
    - 각 DJ는 전체 행사 동안 최대 세 세트까지 공연할 수 있습니다.

    후보 DJ가 n명 있고, 선택할 수 있는 서로 다른 음악 장르가 g가지 있다고 가정합시다. $3k$개의 세트 각각에 DJ와 장르를 배정하거나, 그러한 배정이 불가능하다고 올바르게 보고하는 효율적인 알고리즘을 설명하고 분석하십시오.

15. 매일 같은 사람들이 방문하는 웹 사이트를 운영한다고 가정해 봅시다. 각 방문자는 하나 이상의 *인구통계 집단*에 속한다고 밝힙니다. 예를 들어 어떤 방문자는 자신을 남성, 40–50세, 아버지, 일리노이주 거주자, 학자, 블로거, 그리고 길버트와 설리번의 팬이라고 소개할 수 있습니다.[13] 여러분의 사이트는 광고주의 후원으로 운영됩니다. 각 광고주는 자신의 광고를 어떤 인구통계 집단에게 보여 주어야 하는지, 그리고 매일 자신의 광고를 몇 번 보여 주어야 하는지를 여러분에게 알려 주었습니다. 전체적으로 방문자는 n명, 인구통계 집단은 k개, 광고주는 m명입니다.

    앞 문단에서 설명한 모든 데이터가 주어졌을 때, 모든 광고주가 원하는 개수만큼 광고를 게재하고 모든 광고가 적절한 인구 통계 집단에 속한 누군가에게 노출

[13] 저는 아주 훌륭한 이론 전산학자이며, 구체적으로는 기하 알고리즘 전문가입니다.

되도록 각 방문자에게 하루에 정확히 *하나*의 광고를 보여 줄 수 있는지 판별하는 효율적인 알고리즘을 설명하십시오.

16. 음이 아닌 실수로 이루어진 배열 $A[1..m][1..n]$이 주어졌다고 가정합시다. A의 각 원소 x를 $\lfloor x \rfloor$ 또는 $\lceil x \rceil$로 바꾸되 A의 어느 행이나 열에서도 원소의 합이 달라지지 않도록 하여, A를 정수 행렬로 *반올림*하고자 합니다. 예를 들면 다음과 같습니다.

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longmapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

(a) 이런 방식으로 A를 반올림하거나, 그러한 반올림이 불가능하다고 올바르게 보고하는 효율적인 알고리즘을 설명하고 분석하십시오.

(b) 각 행의 원소 합이 정수이고 각 열의 원소 합도 정수일 *때, 그리고 오직 그 때에만* 적법한 반올림이 가능함을 증명하십시오. 다시 말해, (a)에서 만든 알고리즘이 적법한 반올림을 반환하거나 적법한 반올림이 *명백히* 불가능하거나, 둘 중 하나임을 증명하십시오.

♥(c) 입력 행렬 A의 원소 중 어느 것도 정수가 아님이 보장된다고 가정합시다. A를 반올림하거나 그러한 반올림이 불가능함을 올바르게 보고하는, 훨씬 더 빠른 알고리즘을 기술하고 분석하십시오. 만점을 받으려면 알고리즘이 $O(mn)$ 시간에 실행되어야 합니다. *[힌트: 흐름을* ***사용하지 마십시오.****]*

17. ***애드혹 네트워크***는 저전력 무선 장치로 이루어집니다. 원칙적으로는[14] 이러한 네트워크를 전장이나 최근 자연재해를 입은 지역, 그 밖에 접근하기 어려운 지역에서 사용할 수 있습니다. 저렴하고 단순한 장치를 대량으로 대상 지역 곳곳에 (예를 들어 비행기에서 떨어뜨리는 방식으로) 뿌려 두면, 장치들이 스스로를 자동으로 구성하여 제대로 작동하는 무선 네트워크를 이룬다는 발상입니다.

이 장치들은 제한된 범위 안에서만 통신할 수 있습니다. 모든 장치가 동일하다고 가정합니다. 즉, 어떤 거리 D가 존재하여, 두 장치는 둘 사이의 거리가 D 이하일 때, 그리고 오직 그때에만 통신할 수 있습니다.

우리는 애드혹 네트워크가 신뢰성을 갖추기를 바라지만, 장치들이 저렴하고 저전력이기 때문에 자주 고장 납니다. 어떤 장치가 자신이 고장 날 가능성이 높다는 것을 감지하면, 자신의 통신 범위 안에 있는 다른 *백업* 장치로 자신의 정보를 전송해야 합니다. 우리는 각 장치 x가 모두 x로부터 거리 D 이내에 있는 k개의 잠재적 백업 장치를 가지도록 요구하며, 이 k개의 장치를 x의 ***백업 집합***이라고 부릅니다. 또한 우리는 어떤 장치도 지나치게 많은 다른 장치의 백업 집합에 속하지 않기를 바랍니다. 그렇지 않으면 단 하나의 고장이 네트워크의 상당 부분에 영향을 미칠 수 있기 때문입니다.

이제 통신 반경 D, 매개변수 b와 k, 그리고 거리 배열 $d[1..n, 1..n]$이 주어진다고 가정합시다. 여기서 $d[i, j]$는 장치 i와 장치 j 사이의 거리입니다. n개의

[14] 실제로는 그다지 그렇지 않지만

장치 각각에 대해 크기가 k인 백업 집합을 계산하되 어떤 장치도 b개보다 많은 백업 집합에 나타나지 않도록 하거나, 좋은 백업 집합 모음이 존재하지 않는다고 (올바르게) 보고하는 알고리즘을 설명하십시오.

18. 가혹할 정도로 극심한 예산 삭감의 위협에 직면한 포템킨 대학교는 개설하는 모든 강의가 빈자리 없이 꽉 차도록 배우들을 고용하여 "학생"으로 강의실에 앉혀 두기로 결정했습니다. 배우를 고용하는 데에는 비용이 많이 들기 때문에, 대학교는 가능한 한 적은 수의 배우만 고용하고자 합니다.

    지금은 문을 닫은 Sham-Poobanana 대학교에서 쌓은 리더십 경험을 바탕으로, Potemkin의 행정가들은 여러분에게 방향 비순환 그래프 $G = (V, E)$를 제공했습니다. 이 그래프의 정점은 수업을 나타내며, 각 간선 $i \to j$는 같은 "학생"이 수업 i를 수강한 뒤 나중에 수업 j를 수강할 수 있음을 나타냅니다. 또한 각 수업을 수강할 수 있는 "학생"의 최대 수를 나열한 배열 $cap[1..V]$도 주어집니다. 모든 수업을 정원까지 채울 수 있게 하는 "학생"의 최소 수를 계산하는 알고리즘을 설명하고 분석하십시오.

19. 퀜틴, 앨리스, 그리고 브레이크빌스의 다른 물리 계열 학생들은 네더랜드를 거쳐 필로리로 가는 여행을 계획하고 있습니다. 네더랜드는 여러 광장으로 이루어진 거대하고 황량한 도시이며, 각 광장에는 사람들을 마법으로 다른 세계로 보내 주는 분수가 하나씩 있습니다. 인접한 광장들은 문으로 연결되어 있는데, 이 문들은 야수의 저주를 받았습니다. 광장 사이의 문들은 매시간 5분 동안만 모두 동시에 열리고—12:00부터 12:05까지, 그다음 1:00부터 1:05까지, 이런 식으로—그 밖의 시간에는 잠겨 있습니다. 이 5분 동안 어느 한 문을 두 명 이상이 통과하면 야수가 그들의 존재를 알아챕니다.[15] 게다가 잠긴 문을 열려고 하거나, 같은 5분 동안 둘 이상의 문을 통과하려고 하는 사람은 누구든 니핀으로 변해 버립니다.[16] 하지만 같은 시각에 *서로 다른* 문을 통과하거나 같은 문을 *서로 다른* 시각에 통과하는 것은 몇 명이든 안전하게 할 수 있습니다.

    여러분에게는 니더랜드의 지도가 주어집니다. 이 지도는 각 분수마다 정점 하나, 각 문마다 간선 하나가 있는 그래프 G이며, 지구와 필로리로 통하는 분수들이 명확히 표시되어 있습니다.

    (a) 양의 정수 h도 주어진다고 가정합시다. 아무도 야수에게 들키거나 니핀으로 변하지 않으면서 최대 h시간 안에, 즉 문이 최대 h번 열리는 동안 지구의 분수에서 필로리의 분수까지 걸어갈 수 있는 최대 인원을 계산하는 알고리즘을 설명하고 분석하십시오. 알고리즘의 실행 시간은 h에 의존해야 합니다. *[힌트: 다른 그래프를 구성하십시오.]*

    ♣♥(b) (a)를 푸는 알고리즘 가운데 실행 시간이 V와 E에 대한 다항식이며 h에는 *전혀* 의존하지 않는 알고리즘을 설명하고 분석하십시오.

[15] 이는 매우 나쁜 일입니다.
[16] 이는 매우 매우 나쁜 일입니다.

(c) 한편, 정수 k도 함께 주어진다고 가정합시다. 아무도 야수에게 들키거나 니핀으로 변하지 않으면서 k명의 사람이 지구 분수에서 필로리 분수까지 걸어갈 수 있게 하는 최소 시간 수를 계산하는 알고리즘을 설명하고 분석하십시오. *[힌트: (a)번을 이용하십시오.]*

♥20. $G = (L \sqcup R, E)$를 이분 그래프라고 합시다. 이 그래프의 왼쪽 정점 L에는 $\ell_1, \ell_2, \ldots, \ell_n$으로, 오른쪽 정점에는 $r_1, r_2, \ldots, r_n$으로 번호가 매겨져 있습니다. G의 매칭 M에 속한 모든 간선 쌍 $\ell_i r_j$와 $\ell_{i'} r_{j'}$에 대해 $i < i'$인 것과 $j < j'$인 것이 동치이면, M을 ***비교차*** 매칭이라고 합니다.

(a) G에서 가장 큰 비교차 매칭을 찾는 알고리즘을 설명하고 분석하십시오. *[힌트: 사실 이것은 흐름 문제가 아닙니다.]*

(b) G의 각 간선이 정확히 하나의 매칭 M_i에만 속하도록 하는, 서로 교차하지 않는 매칭 $M_1, M_2, \ldots, M_k$의 최소 개수를 구하는 알고리즘을 설명하고 분석하십시오. *[힌트: 이것은 사실 흐름 문제입니다.]*

♥21. $G = (L \sqcup R, E)$를 이분 그래프라고 하고, 그 왼쪽 정점 L에는 임의의 순서로 $\ell_1, \ell_2, \ldots, \ell_n$의 번호가 매겨져 있다고 합시다.

(a) G의 매칭 M에서 L에 매칭되지 않은 정점이 연속으로 나타나지 않으면, M을 ***조밀하다***고 합니다. 즉, 모든 인덱스 i에 대해 정점 ℓ_i와 ℓ_{i+1} 중 적어도 하나는 M에 속한 간선에 접합니다. G에 조밀한 매칭이 있는지 판별하는 알고리즘을 설명하십시오.

(b) G의 매칭 M은 L에 연속으로 *매칭된* 정점이 없을 때 ***희소***하다고 합니다. 즉, 모든 인덱스 i에 대해 정점 ℓ_i와 ℓ_{i+1} 중 적어도 하나는 M의 간선에 접하지 *않습니다*. (특히 빈 매칭은 희소합니다.) G에서 가장 큰 희소 매칭을 찾는 알고리즘을 설명하십시오.

(c) 모든 인덱스 i에 대해 ℓ_i와 ℓ_{n-i+1}이 둘 다 M의 간선에 접하거나, ℓ_i와 ℓ_{n-i+1} 중 어느 것도 M의 간선에 접하지 않으면, G의 매칭 M을 ***회문적***이라고 합니다. (특히, 빈 매칭은 회문적입니다.) G에서 가장 큰 회문적 매칭을 찾는 알고리즘을 설명하십시오.

이 문제들 중 어느 것도 R의 어떤 정점이 매칭되고 어떤 정점이 매칭되지 않는지를 제한하지 않습니다.

♥22. *루트 트리*는 방향 비순환 그래프로, 루트를 제외한 모든 정점이 정확히 하나의 들어오는 간선을 가지며 루트에는 들어오는 간선이 없습니다. 같은 말로, 루트 트리는 루트 정점 하나로 이루어지고, 이 루트 정점에서 0개 이상의 더 작은 루트 트리의 루트로 향하는 간선이 나갑니다. 두 루트 트리 A와 B가 주어질 때, A의 부분 그래프 및 B의 부분 그래프와 모두 동형인 가장 큰 루트 트리를 계산하는 효율적인 알고리즘을 설명하십시오. 더 간단히 말해, 두 루트 트리의 가장 큰 공통 부분 트리를 찾는 알고리즘을 설명하십시오.

[힌트: 모든 노드의 자식이 $O(1)$개이거나 각 노드의 자식들이 왼쪽에서 오른쪽으로 순서가 정해져 있다면 이것은 비교적 간단한 동적 계획법 문제일 것입니다. 그러나 차수가 큰 순서 없는 트리의 경우에는 재귀적 부분 문제를 효율적으로 결합하기 위한 다른 기법이 필요합니다.]

[그]는 짧고 단편적인 논고에서 영원한 본보기를 제시합니다—법칙의 본보기도, 심지어 방법의 본보기도 아닙니다. 매우 지적이라는 것 외에는 어떤 방법도 없기 때문입니다. 그것은 감각을 분석하여 원리와 정의에 이르기까지 신속하게 나아가는 지성 그 자체의 본보기입니다.

— T. S. 엘리엇, 아리스토텔레스에 관하여, "완벽한 비평가(The Perfect Critic)", *성스러운 숲(The Sacred Wood)* (1921)

표준의 좋은 점은 고를 수 있는 표준이 아주 많다는 것입니다. 게다가 그중 어느 것도 마음에 들지 않는다면, 그냥 내년 모델을 기다리면 됩니다.

— Andrew S. Tannenbaum, *Computer Networks* (1981)

지금까지 존재하지 않던 것을 눈부실 만큼 자명하게 만들어 낼 수 있는 정신은 참으로 드뭅니다. "나도 그건 생각해 낼 수 있었을 텐데" 라는 외침은 매우 흔하면서도 오해를 부르는 말입니다. 실제로 그들은 그것을 생각해 내지 못했기 때문이며, 이는 매우 의미심장하고 많은 것을 드러내는 사실이기도 합니다.

— 더크 젠틀리가 리처드 맥더프에게
더글러스 애덤스, *더크 젠틀리의 전체론적 탐정 사무소* (1987)에서

어떤 문제에 해답이 없다면, 그것은 문제가 아니라 사실일지도 모릅니다 – 해결할 대상이 아니라 시간을 두고 감당해 나가야 할 대상입니다.

— 시몬 페레스, 데이비드 럼즈펠드의 인용, *Rumsfeld's Rules* (2001)

12

NP-난해성

12.1 이길 수 없는 게임

수상할 만큼 톰 웨이츠를 닮은, 빨간 정장 차림의 세일즈맨이 앞면에 n개의 이진 스위치가 달려 있고 윗면에 전구가 하나 달린 검은 강철 상자를 여러분에게 내민다고 상상해 보십시오. 세일즈맨은 전구의 상태가 복잡한 *불 회로*—전선으로 연결된 AND, OR, NOT 게이트의 모음으로, 스위치마다 입력 전선이 하나씩 있고 전구로 이어지는 출력 전선이 하나 있습니다—에 의해 제어된다고 말합니다. 그러고 나서 그는 여러분에게 간단한 질문을 던집니다. 전구가 켜지도록 스위치를 설정하는 것이 가능합니까? 이 질문에 올바르게 답할 수 있다면 그는 당신에게 100만 1000억 달러를 줄 것입니다. 틀리게 답하거나 답하지 못한 채 죽는다면 그는 당신의 영혼을 가져갈 것입니다.

여러분이 알 수 있는 한, 적대자는 스위치를 전구에 전혀 연결하지 않았으므로 스위치를 어떻게 설정하든 전구는 계속 꺼져 있을 것입니다. 여러분이 전구를 켜는 것이 *가능하다*고 선언하면, 적대자는 상자를 열어 회로가 아예 없다는 것을 보여 줄

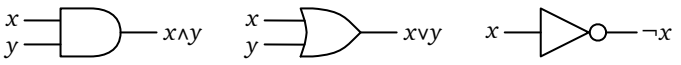


그림 12.1. AND 게이트, OR 게이트, NOT 게이트입니다.

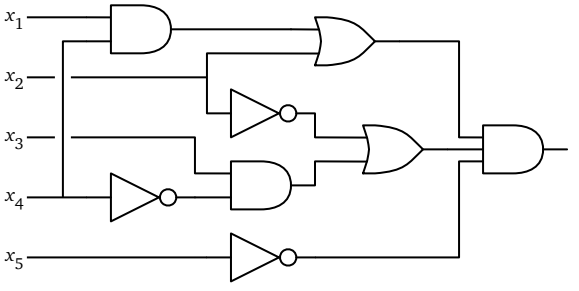


그림 12.2. 불 회로입니다. 입력은 왼쪽에서 들어오고, 출력은 오른쪽으로 나갑니다.

것입니다. 그러나 2^n가지 설정을 모두 시험해 보기 전에 전구를 켜는 것이 *불가능하다*고 선언하면, 적대자는 여러분이 아직 시험하지 않은 설정 중 하나로 스위치가 놓여 있을 *때, 그리고 오직 그때에만* 전구를 켜는 회로를 마법처럼 상자 안에 만들어 낸 뒤, 스위치를 그 설정으로 바꾸어 전구를 켤 것입니다. (끝날 때까지 상자 안을 볼 수 없으므로 여러분은 적대자의 속임수를 알아챌 수 없습니다.) 적대자의 질문에 *증명 가능하게* 올바로 답하는 유일한 방법은 가능한 2^n가지 설정을 모두 시험해 보는 것입니다. 여러분은 이 일이 여러분이 살 것으로 예상하는 기간보다 *훨씬* 오래 걸린다는 것을 곧 깨닫고, 적대자의 제안을 정중히 거절합니다.

적대자는 미소를 지으며, 말보로 한 보루를 다 피우고 난 히스 레저의 조커처럼 으르렁거리는 목소리로 말합니다. "아, 네, 물론입니다. 당신에게는 저를 믿을 이유가 없습니다. 하지만 어쩌면 당신을 안심시켜 드릴 수 있을지도 모르겠습니다." 그는 회로도가 그려진(혹은 어쩌면 문신으로 새겨진) 커다란 양피지 두루마리를 당신에게 건넵니다—부디 양의 가죽으로 만든 것이기를 바랄 뿐입니다. "여기 상자 안에 든 회로의 완전한 설계도가 있습니다. 설계도가 정확한지 확인하도록 상자 안을 마음껏 살펴보십시오. 아니면 이 설계도대로 직접 회로를 만들어 보십시오. 아니면 회로를 시뮬레이션하는 컴퓨터 프로그램을 작성하십시오. 무엇이든 원하시는 대로 하십시오. 설계도가 상자 안의 실제 회로와 일치하지 않는다는 것을 발견하면, 천억 달러는 당신의 것입니다." 몇 군데를 표본 점검해 보니 설계도에 명백한 결함은 없다는 확신이 듭니다. 교묘한 속임수도 불가능해 보입니다.

그래도 여전히 적대자의 "관대한" 제안은 거절해야 합니다. 적대자가 내는 문제는 ***회로 충족 가능성*** 또는 ***CIRCUITSAT***이라고 부릅니다. 불 회로가 주어졌을 때 회로가 TRUE를 출력하게 만드는 입력 집합이 존재하는지, 또는 반대로 회로가 *항상* FALSE를 출력하는지를 묻는 문제입니다. *특정한* 입력 설정 하나에 대해서는 깊이 우선 탐색을 사용하여 다항 시간(사실은 *선형* 시간)에 회로의 출력을 계산할 수 있습니다. 하지만 회로에 가능한 2^n개의 입력을 전수 조사로 모두 시도하는 것보다 빠르게 CIRCUITSAT을 푸는 방법은 아무도 알지 못하며, 전수 조사에는 지수 시간이 걸립니다. 물론 전수

조사보다 나은 방법이 없다는 것을 실제로 엄밀하게 *증명한* 사람은 아무도 없습니다—어쩌면, 정말 *어쩌면*, 아직 발견되지 않았을 뿐인 영리한 알고리즘이 있을지도 모릅니다—하지만 반중력 유니콘이 존재하지 않는다는 것을 실제로 엄밀하게 증명한 사람도 아무도 없습니다. 실용적인 관점에서는 CircuitSat을 푸는 빠른 알고리즘이 존재하지 않는다고 가정해도 무방합니다.

여러분은 판매원에게 거절합니다. 그는 미소를 지으며 "꼬마야, 보기보다 똑똑하구나"라고 말하고는 반중력 유니콘을 타고 날아가 버립니다.

12.2 P 대 NP

알고리즘이 "효율적"이라고 인정받기 위한 최소한의 조건은 실행 시간이 입력 크기에 대한 다항 함수로 제한되는 것입니다. 즉, 입력의 크기를 n이라 할 때 어떤 상수 c에 대해 $O(n^c)$여야 합니다.[1] 연구자들은 모든 문제를 이렇게 빠르게 풀 수는 없다는 사실을 일찍부터 알고 있었지만, 정확히 어떤 문제는 그렇게 풀 수 있고 어떤 문제는 풀 수 없는지 밝혀내는 데에는 큰 어려움을 겪었습니다. 이른바 ***NP-난해*** 문제가 여럿 있는데, 아무도 초다항식 하한을 증명하지 못했음에도 대부분의 사람들은 이 문제들을 다항 시간에 *풀 수 없다*고 믿습니다.

*결정 문제*란 출력이 Yes 또는 No라는 불리언 값 하나뿐인 문제입니다. 결정 문제의 세 가지 클래스를 다음과 같이 정의하겠습니다.

- ***P***는 다항 시간 안에 풀 수 있는 결정 문제들의 집합입니다. 직관적으로 말하면, P는 빠르게 풀 수 있는 문제들의 집합입니다.
- ***NP***는 다음 성질을 갖는 결정 문제들의 집합입니다. 답이 Yes이면, 그 사실에 대한 *증명*이 존재하며 그 증명은 다항 시간 안에 검증할 수 있습니다. 직관적으로 말하면, NP는 해답이 눈앞에 주어졌을 때 Yes라는 답을 빠르게 검증할 수 있는 결정 문제들의 집합입니다.
- ***co-NP***는 본질적으로 NP의 반대입니다. co-NP에 속하는 문제의 답이 No라면, 이 사실에 대한 증명이 존재하며 그 증명은 다항 시간 안에 검증할 수 있습니다.

예를 들어, 회로 충족 가능성 문제는 NP에 속합니다. 주어진 불 회로가 충족 가능하다면, True 출력을 만들어 내는 m개의 입력 값 집합은 어느 것이든 그 회로가 충족 가능하다는 증명이 됩니다. 이 증명은 회로를 다항 시간에 평가하여 확인할 수 있습니다. 회로 충족 가능성은 P에도, co-NP에도 속하지 *않는다고* 널리 믿어지고 있지만, 실제로 아는 사람은 아무도 없습니다.

P에 속하는 모든 결정 문제는 NP에도 속합니다. 어떤 문제가 P에 속한다면, 답을 처음부터 다시 계산하여 Yes 답을 다항 시간 안에 검증할 수 있습니다! 마찬가지로, P에 속하는 모든 문제는 co-NP에도 속합니다.

이론 컴퓨터 과학(Theoretical Computer Science)에서—아니, 컴퓨터 과학 전체에서—아니, ***과학*** 전체에서—아직 답이 나오지 않은 단 하나의 가장 중요한 질문은

[1] 이러한 효율성 개념은 1965년 앨런 코범(Alan Cobham), 1965년 잭 에드먼즈(Jack Edmonds), 1966년 마이클 라빈(Michael Rabin)이 각각 독립적으로 정형화했지만, 이와 비슷한 개념은 그보다 10여 년 앞서 쿠르트 괴델(Kurt Gödel), 존 내시(John Nash), 존 폰 노이만(John von Neumann)이 이미 고려한 바 있습니다.

아마도 복잡도 클래스 P와 NP가 실제로 서로 다른지 여부일 것입니다. 직관적으로는 대부분의 사람에게 P ≠ NP가 자명해 보입니다. 알고리즘과 자료구조 수업의 과제와 시험을 거치면서 여러분은 (바라건대) 돌이켜 보면 해답이 단순한 경우에도 문제를 푸는 일이 엄청나게 어려울 수 있다는 것을 확신하게 되었을 것입니다. 이는 너무나 자명합니다. 문제를 처음부터 푸는 것은 *당연히* 주어진 해답이 올바른지 검증하는 것보다 어렵습니다. 우리는 "P ≠ NP"라는 명제를 맥스웰 방정식, 일반 상대성 이론, 내일 아침에도 해가 뜬다는 사실처럼 증거로 강력하게 뒷받침되지만 수학적 증명은 없는 다른 자연 법칙과 마찬가지로 하나의 자연 법칙으로 받아들이는 것이 합리적이며—실제로 대부분의 알고리즘 설계자는 이를 그렇게 *받아들입니다*.

하지만 수학적으로 엄밀하게 따지자면, P ≠ NP임을 *증명*하는 방법을 아는 사람은 아무도 없다는 사실을 인정해야 합니다. 실제로 지난 수십 년 동안 증명을 향한 실질적인 진전은 거의 또는 전혀 없었습니다.[2] 클레이 수학 연구소는 P 대 NP 문제를 7개의 밀레니엄 난제 중 첫 번째로 꼽고 있으며, 이 문제를 해결하면 $1,000,000의 상금을 수여합니다. 그리고 실제로 여러 사람이 이 문제를 풀려고 시도하다가 영혼을, 적어도 제정신을 *정말로* 잃었습니다.

이보다 더 미묘하지만 여전히 풀리지 않은 문제는 복잡도 클래스 NP와 co-NP가 서로 다른지입니다. 모든 Yes 답을 빠르게 검증할 수 있더라도 No 답까지 빠르게 검증할 수 있다고 믿을 근거는 없습니다. 예를 들어, 우리가 아는 한 불 회로가 충족 가능하지 *않다*는 것을 보이는 짧은 증명은 없습니다. 일반적으로 NP ≠ co-NP라고 여겨지지만, 이번에도 이를 증명하는 방법은 아무도 알지 못합니다.

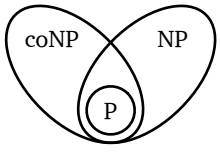


그림 12.3. 우리가 *생각하는* 세상의 모습입니다.

12.3 NP-난해, NP-쉬움, NP-완전

Π에 대한 다항 시간 알고리즘이 존재하면 *NP에 속하는 모든 문제*에 대한 다항 시간 알고리즘도 존재하게 되는 경우, 문제 Π를 ***NP-난해***라고 합니다. 다시 말하면 다음과 같습니다.

Π는 NP-난해입니다 ⟺ Π를 다항 시간에 풀 수 있다면 P=NP입니다

직관적으로 말하면, 특정한 NP-난해 문제 하나를 빠르게 풀 수 있다면, 그 특별한 문제의 해법을 서브루틴으로 사용하여 해를 쉽게 이해할 수 있는 *어떤* 문제라도 빠르게 풀 수 있습니다. NP-난해 문제는 적어도 NP에 속하는 모든 문제만큼 어렵습니다.

[2] 아마도 가장 의미 있는 진전은 *장벽* 결과의 형태로 이루어졌을 것입니다. 이 결과들은 공략 방향 전체가 실패할 수밖에 없음을 뜻합니다. 아주 실질적인 의미에서, 우리는 P ≠ NP를 증명하는 방법을 전혀 모를 뿐만 아니라, P ≠ NP를 증명하는 방법을 우리가 전혀 모른다는 사실을 실제로 *증명*할 수 있습니다!

마지막으로, 어떤 문제가 NP-난해이면서 동시에 NP의 원소(또는 "NP-쉬움")이면 그 문제를 ***NP-완전***이라고 합니다. 비형식적으로 말하면, NP-완전 문제는 NP에서 가장 어려운 문제입니다. NP-완전 문제 단 하나에 대해서라도 다항 시간 알고리즘이 있다면, 곧바로 모든 NP-완전 문제에 대한 다항 시간 알고리즘이 있다는 결론이 나옵니다. 말 그대로 *수천* 개의 문제가 NP-완전임이 증명되었으므로, 그중 하나(따라서 전부)에 대한 다항 시간 알고리즘이 존재할 가능성은 극히 낮아 보입니다.

어떤 문제를 NP-난해라고 부르는 것은 "내가 개를 키운다면, 그 개는 영어를 유창하게 할 수 있습니다"라고 말하는 것과 같습니다. 여러분은 아마 제가 개를 키우는지 아닌지 모르겠지만, 제가 *말하는* 개를 키우지 않는다는 것만큼은 꽤 확신할 것입니다. 개가 영어를 할 수 없다는 수학적 *증명*을 가진 사람은 아무도 없습니다—개가 영어로 말하는 것을 들어 본 사람이 아무도 없다는 사실은 *증거*이며, 알맞은 입 모양과 지능을 갖추지 못한 개들을 대상으로 한 수백 건의 검사 역시 증거이지만, 단순한 증거는 수학적 *증명*이 아닙니다. 그럼에도 불구하고 제가 영어를 유창하게 하는 개를 키운다고 말한다면, 제정신인 사람은 아무도 저를 믿지 않을 것입니다.[3] 따라서 "내가 개를 키운다면, 그 개는 영어를 유창하게 할 수 있습니다"라는 진술에는 자연스러운 따름정리가 있습니다. 제정신인 사람이라면 누구도 제가 개를 키운다고 믿어서는 안 됩니다! 마찬가지로, 어떤 문제가 NP-난해라면 제정신인 사람이라면 누구도 그 문제를 다항 시간에 풀 수 있다고 믿어서는 안 됩니다.

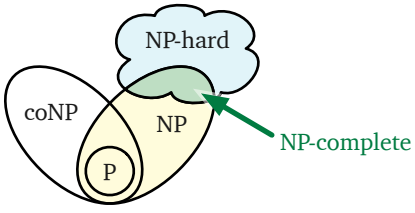


그림 12.4. 세상이 어떤 모습이라고 우리가 *생각하는지*를 조금 더 보여 줍니다.

NP-난해한 문제가 *하나라도* 존재한다는 사실은 곧바로 자명하지 않습니다. 다음의 놀라운 정리는 1971년 스티븐 쿡(Stephen Cook)이 처음 발표했으며, 1973년에는 레오니드 레빈(Leonid Levin)이 독자적으로 발표했습니다.[4]

쿡-레빈 정리. *회로 충족 가능성 문제는 NP-난해입니다.*

정의를 (일부러) 모호하게 제시했으므로, 여기서는 증명의 개요조차 제시하지 않겠습니다.[5]

[3]… 학과장은 슬프게 고개를 저으며 말합니다. "아, 이것 보십시오, 그냥 짖는 소리로 들립니다. *제가* 질문을 하나 하겠습니다. 역사상 가장 위대한 복잡도 이론가는 누구였습니까?" 개는 고개를 갸웃하고 몇 초 동안 멈추었다가 "카프!"라고 말합니다. 학과장이 그들을 연구실에서 쫓아낸 뒤, 개는 주인을 돌아보며 말합니다. "임팔리아초라고 했어야 했습니까?"

[4]레빈은 아직 박사 과정 학생이던 1971년에 모스크바의 세미나에서 자신의 결과를 처음 보고했습니다. 쿡의 결과에 관한 소식은 레빈의 결과 발표가 출판된 뒤인 적어도 1973년까지 소련에 전해지지 않았습니다. 그런데도 스티글러의 법칙에 따라 이 결과는 흔히 "쿡의 정리"라고 불립니다. 레빈은 정치적인 이유로 모스크바 대학교에서 박사 학위를 받지 못했습니다. 그는 1978년 미국으로 이주했고, 1년 뒤 MIT에서 박사 학위를 받았습니다. 쿡은 자신의 기념비적인 논문을 발표하기 불과 1년 전인 1970년에 버클리 수학과에서 종신 재직권을 거부당했습니다. 이후 그는 (레빈과 달리) 이 증명으로 튜링상을 받았습니다.

[5]관심 있는 독자는 http://algorithms.wtf 에 있는 비결정적 튜링 기계에 관한 제 강의 노트나 Boaz

12.4 ♥형식적 정의 (*HC SVNT DRACONES*)

형식적으로 복잡도 클래스 P, NP, co-NP는 *언어*와 *튜링 기계*를 사용하여 정의됩니다. 언어는 어떤 유한 알파벳 Σ 위의 문자열들의 집합이며, 일반성을 잃지 않고 $\Sigma = \{0, 1\}$이라고 가정할 수 있습니다. 튜링 기계는 매우 제한적인 유형의 컴퓨터로서—거칠게 말하면 무한한 메모리 테이프를 가진 유한 상태 기계입니다—놀랍게도 그 정확한 정의는 별로 중요하지 않습니다. P는 결정적 단일 테이프 튜링 기계로 ***P***olynomial(다항) 시간에 판정할 수 있는 언어들의 집합입니다. 마찬가지로 NP는 비결정적 튜링 기계로 다항 시간에 판정할 수 있는 모든 언어의 집합이며, NP는 ***N***ondeterministic ***P***olynomial-time(비결정적 다항 시간)의 약자입니다.

다항 시간이라는 요구 조건은 충분히 느슨하므로 튜링 기계의 정확한 형태(테이프의 개수, 헤드의 개수, 트랙의 개수, 테이프 알파벳의 크기 등)를 구체적으로 정할 필요가 없습니다. 실제로 임의 접근 기계[6]에서 $T(n)$ 시간에 실행되는 알고리즘은 모두 $O(T(n)^4)$ 시간에 실행되는 단일 테이프, 단일 트랙, 단일 헤드 튜링 기계로 시뮬레이션할 수 있습니다. 이 시뮬레이션 결과 덕분에 모든 것을 튜링 기계로 직접 기술하는 대신, 배열, 반복문, 재귀 같은 표준적인 고수준 프로그래밍 구성 요소를 사용하여 계산 복잡도를 형식적으로 논증할 수 있습니다.

형식적으로, 문제 Π가 NP-난해라는 것은 모든 언어 $\Pi' \in \text{NP}$에 대해 Π'에서 Π로의 다항 시간 ***튜링 환원***이 존재한다는 것과 동치입니다. 튜링 환원이란 튜링 기계에서 실행할 수 있는 환원을 뜻합니다. 즉, Π를 푸는 또 다른 튜링 기계 M'을 블랙박스 서브루틴으로 사용하여 Π'을 풀 수 있는 튜링 기계 M을 말합니다. 튜링 환원은 *오라클 환원*이라고도 하며, 다항 시간 튜링 환원은 *쿡 환원*이라고도 합니다.

복잡도 이론 연구자들은 NP-난해성을 다항 시간 ***다대일 환원(many-one reduction)***으로 정의하는 것을 선호하며, 이 환원은 *카프 환원(Karp reduction)*이라고도 합니다. 한 언어 $L' \subseteq \Sigma^*$에서 다른 언어 $L \subseteq \Sigma^*$로의 *다대일* 환원이란 $x \in L'$일 필요충분조건이 $f(x) \in L$인 함수 $f : \Sigma^* \to \Sigma^*$를 말합니다. 그러면 임의의 언어 $L' \in \text{NP}$에 대해 L'에서 L로의 다대일 환원 중 다항 시간에 계산할 수 있는 것이 존재할 때, 그리고 오직 그때에만 *언어* L을 NP-난해라고 정의할 수 있습니다.

모든 카프 환원은 쿡 환원"이지만", 그 역은 성립하지 않습니다. 구체적으로, 한 결정 문제 Π에서 다른 결정 문제 Π'로의 카프 환원은 Π의 입력을 Π'의 입력으로 변환하고, Π'에 대한 오라클(즉, 서브루틴)을 호출한 다음, 그 답을 그대로 반환하는 것과 동등합니다. 그러나 우리가 아는 한, 모든 쿡 환원을 카프 환원으로 시뮬레이션할 수 있는 것은 아닙니다.

Barak의 훌륭한 저서 *Introduction to Theoretical Computer Science*에서 증명을 찾을 수 있습니다.

[6] 임의 접근 기계는 실제 물리적 컴퓨터를 더 충실하게 모델링하는 계산 모델입니다. 표준 임의 접근 기계는 크기 제한이 없는 임의 접근 메모리를 가지며, 이 메모리는 크기 제한이 없는 배열 $M[0..\infty]$로 모델링됩니다. 여기서 어떤 고정된 정수 w에 대해 각 주소 $M[i]$에는 w비트 정수 하나가 저장되며, 이 기계는 어떤 메모리 주소든 상수 시간에 읽거나 쓸 수 있습니다. RAM 알고리즘은 형식적으로는 어셈블리와 비슷한 언어로 작성하며, `ADD` i, j, k("$M[i] \leftarrow M[j] + M[k]$"라는 뜻), `INDIR` i, j("$M[i] \leftarrow M[M[j]]$"라는 뜻), `IF0GOTO` i, ℓ("$M[i] = 0$이면 ℓ번 줄로 이동"이라는 뜻) 같은 명령어를 사용합니다. 하지만 정확히 어떤 명령어 집합을 쓰는지는 놀라울 만큼 중요하지 않습니다. 정의에 따라 각 명령어는 단위 시간에 실행됩니다. 실제로는 산술 정밀도에만 주의한다면 RAM 알고리즘을 더 고수준의 의사코드로 충실하게 기술할 수 있습니다.

복잡도 이론가들이 카프 환원을 선호하는 주된 이유는 NP가 카프 환원에 대해서는 닫혀 있지만 쿡 환원에 대해서는 *닫혀 있지 않기* 때문입니다(NP=co-NP가 아닌 한 그러하며, NP=co-NP일 가능성은 낮다고 여겨집니다). (1) 쿡 환원에 대해서는 NP-난해이지만, (2) 카프 환원에 대해서는 P=NP인 경우에만 NP-난해인 자연스러운 문제들이 존재합니다. 그러한 문제의 자명한 예 하나는 UNSAT입니다. 불 논리식이 주어졌을 때, 그 논리식은 *항상 거짓*입니까? 반면에 다대일 환원은 결정 문제(더 형식적으로는 언어)에*만* 적용됩니다. 형식적으로 말하면, 어떤 최적화 문제나 구성 문제도 카프-NP-난해가 아닙니다.

설상가상으로, 쿡과 카프는 원래 NP-난해성을 ***로그 공간*** 환원으로 정의했습니다. 모든 로그 공간 환원은 다항 시간 환원이지만, (우리가 아는 한) 그 역은 성립하지 않습니다. 허용되는 (쿡 또는 카프) 환원의 집합을 로그 공간에서 다항 시간으로 완화하면 NP-난해 문제의 집합이 달라지는지는 아직 풀리지 않은 문제입니다.

다행히도 실제로는 이러한 미묘한 문제들이 전혀 불거지지 않습니다—특히 이 장에서 설명하는 모든 환원은 로그 공간 다대일 환원으로 형식화할 수 있습니다—그러니 이제 잠에서 깨셔도 됩니다.

12.5 환원과 SAT

회로 충족 가능성 이외의 문제가 NP-난해임을 증명하려면 *환원* 논증을 사용합니다. 문제 A를 다른 문제 B로 환원한다는 것은 문제 B를 푸는 알고리즘이 이미 존재한다는 가정 아래 문제 A를 푸는 알고리즘을 기술하는 것을 뜻합니다. 여러분은 이 책을 시작하기 전부터 이미 수년간 환원을 해 왔습니다. 다만 아마도 서브루틴, 유틸리티 함수, 모듈식 프로그래밍, 계산기 사용 같은 다른 이름으로 불렀을 것입니다. 어떤 문제가 NP-난해임을 증명할 때에도 문제 사이의 비슷한 변환을 기술하지만, 그 방향은 대부분의 사람들이 예상하는 방향과 다릅니다.

다음 규칙은 손등에 문신으로 새겨 두십시오. 엄마 생일과 모노폴리의 *진짜* 규칙 바로 옆에 말입니다.[7]

> **문제 *A*가 NP-난해임을 증명하려면, 이미 알려진 NP-난해 문제를 *A*로 환원하십시오.**

다시 말해, 여러분의 문제가 어렵다는 것을 증명하려면, *여러분의* 문제를 푸는 가상의 효율적인 알고리즘을 블랙박스 서브루틴으로 사용하여, 이미 어렵다고 알고 있는 *다른* 문제를 푸는 효율적인 알고리즘을 기술해야 합니다. 핵심 논리는 귀류법입니다.

[7] 플레이어가 주인 없는 부동산 칸에 도착했는데 그 부동산을 사지 않거나 살 수 없으면, 그 부동산은 즉시 경매에 부쳐져 가장 높은 값을 부른 사람에게 넘어갑니다. 처음에 그 부동산을 사지 않은 플레이어도 입찰할 수 있으며, 입찰가는 정가보다 얼마든지 높거나 낮아도 됩니다. 감옥에 있는 플레이어도 부동산을 사고팔 수 있고, 집과 호텔을 사고팔 수 있으며, 임대료도 받을 수 있습니다. 게임에는 집 32채와 호텔 12채가 있으며, 한번 다 떨어지면 그것으로 끝입니다. 특히 모든 집이 이미 판 위에 나와 있다면 호텔을 집 네 채로 낮출 수 없으며, 그 그룹의 호텔을 모두 완전히 허물어야 합니다. 플레이어끼리는 *개발되지 않은* 부동산을 팔거나 교환할 수 있지만, 부동산을 은행에 되팔 수는 없습니다. 반면 건물은 은행에 (반값에) 팔 수 있지만, 플레이어끼리 팔거나 교환할 수는 없습니다. 모든 벌금은 은행에 직접 냅니다. 무료 주차 칸에 도착한 플레이어는 아무것도 받지 못합니다. 출발 칸에 도착한 플레이어는 정확히 $200를 받으며, 그 이상은 받지 못합니다. 철도는 마법의 순간 이동 장치가 아닙니다. 마지막으로, Jeff는 *언제나* 부츠 말을 가져갑니다. 아니요, 티라노사우루스나 펭귄이 아니라—부츠입니다, 젠장.

환원에 따르면 여러분의 문제가 쉽다면 다른 문제도 쉬울 것인데, 그 문제는 전혀 쉽지 않습니다. 달리 말하면, 다른 문제가 어렵다는 것을 알고 있으므로 환원에 따라 여러분의 문제도 어려울 수밖에 없습니다. 즉, 여러분이 가정한 효율적인 알고리즘은 실제로 존재하지 않습니다.

대표적인 예로, 흔히 간단히 ***Sat***라고 부르는 *논리식 충족 가능성* 문제를 살펴봅시다. Sat의 입력은 다음과 같은 불리언 *논리식*입니다

$$(a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee \overline{(\bar{a} \Rightarrow d)} \vee (c \neq a \wedge b)),$$

문제는 전체 논리식이 True로 평가되도록 변수 $a, b, c, \ldots$ 에 불리언 값을 할당할 수 있는지 여부입니다.

Sat가 NP-난해임을 증명하려면 이미 알려진 NP-난해 문제로부터의 환원을 제시해야 합니다. 지금까지 NP-난해 문제로 알고 있는 유일한 문제는 CircuitSat이므로, 여기서부터 시작합시다.

K를 임의의 불 회로라고 합시다. 다음과 같이 K를 불 논리식 Φ로 변환(더 정확하게는 *전사*)할 수 있습니다. 먼저 각 내부 전선에 새로운 변수 y_j로 레이블을 붙이고, 출력 전선에는 새로운 변수 z로 레이블을 붙입니다. 논리식 Φ는 게이트마다 하나씩 있는 등식들을 And로 구분하여 나열하고, 마지막에 $\wedge z$를 덧붙인 것입니다. 그림 12.5는 예제 회로에 대해 이렇게 얻은 전사 결과를 보여 줍니다.

이제 원래 회로 K가 만족 가능할 ***필요충분조건은*** 그 결과로 얻은 논리식 Φ가 만족 가능한 것이라고 주장합니다. 다른 모든 "필요충분조건" 명제와 마찬가지로, 이 주장은 두 단계로 나누어 증명합니다:

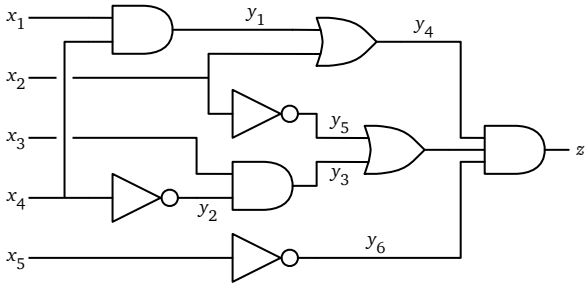
⇒ 회로 K를 충족하는 입력 집합이 주어지면, K의 모든 게이트의 출력을 계산하여 논리식 Φ에 대한 만족하는 할당을 유도할 수 있습니다.

⇐ 식 Φ를 만족하는 할당이 주어지면, 내부 배선 변수 y_i와 출력 변수 z를 무시하기만 하면 회로를 만족하는 입력을 얻을 수 있습니다.

회로를 논리식으로 바꾸는 변환 전체는 선형 시간에 수행할 수 있습니다. 게다가 결과로 얻는 논리식의 크기는 회로를 합리적으로 표현한 어떤 방식과 비교해도 기껏해야 상수 배만큼만 더 큽니다.

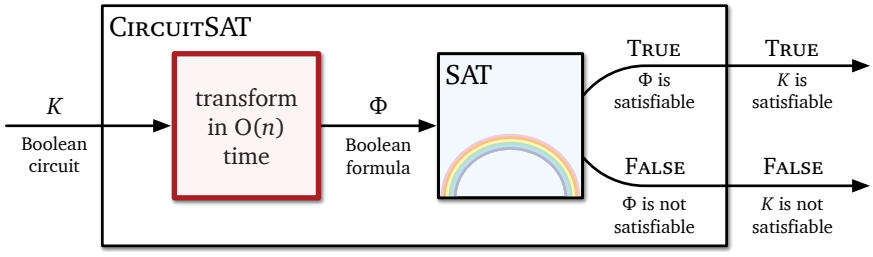
이제 논의를 위해, 주어진 불 논리식이 충족 가능한지를 다항 시간 안에 판정할 수 있는 알고리즘이 존재한다고 가정해 봅시다. 그러면 임의의 불 회로 K가 주어졌을 때, 먼저 위에서 설명한 대로 K를 불 논리식 Φ로 변환한 다음, 다음 만화에서 보듯이 마법 같은 미지의 Sat 알고리즘에 Φ가 충족 가능한지 물어봄으로써 K가 충족 가능한지를 판정할 수 있습니다. 각 상자는 알고리즘을 나타냅니다. 왼쪽의 빨간 상자는 변환 서브루틴입니다. 오른쪽의 상자는 가상의 마법 Sat 알고리즘입니다. 그 위에 *무지개*가 있으니 이것은 마법임이 *틀림없습니다*.[8]

[8] Kay Erickson, 개인적인 대화, 2011. 흑백으로 인쇄된 사본을 읽고 계신 분들께: 네, 그 둥근 것은 무지개입니다.



$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge$$
$$(y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (z = y_4 \wedge y_7 \wedge y_6) \wedge z$$

그림 12.5. 불 회로를 불 논리식으로 옮겨 적기.



마법 상자보다 마법 의사코드를 선호한다면 다음과 같습니다:

```
CIRCUITSAT(K):
    transcribe K into a boolean formula Φ
    return SAT(Φ)    ⟪★★ ★ ★MAGIC★ ★ ★★⟫
```

K를 Φ로 옮겨 적는 데에는 다항 시간만 걸리므로(사실은 *선형* 시간이면 충분하지만, 어느 쪽이든 상관없습니다), 전체 CIRCUITSAT 알고리즘 역시 다항 시간에 실행됩니다.

$$T_{\text{CIRCUITSAT}}(n) \le O(n) + T_{\text{SAT}}(O(n))$$

따라서 SAT에 대한 다항 시간 알고리즘이 있다면 CIRCUITSAT에 대한 다항 시간 알고리즘을 얻게 되고, 이는 다시 P=NP를 함의한다는 결론을 얻습니다. 그러므로 SAT은 NP-난해합니다!

12.6 3SAT (CIRCUITSAT에서 환원)

NP-난해성 결과를 증명하는 데 특히 유용한 SAT의 특수한 경우를 ***3CNF-SAT***, 또는 더 흔히 간단하게 ***3SAT***이라고 부릅니다.

불 논리식이 여러 개의 *절*의 논리곱(AND)이고, 각 절은 여러 개의 *리터럴*의 논리합(OR)이며, 각 리터럴은 변수이거나 그 변수의 부정이면, 그 불 논리식은 *논리곱*

정규형(CNF)에 있다고 합니다. 예를 들면 다음과 같습니다.

$$\overbrace{(a \vee b \vee c \vee d)}^{\text{절}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})$$

3CNF 논리식은 각 절에 리터럴이 정확히 세 개씩 있는 CNF 논리식입니다. 앞의 예는 첫 번째 절에 리터럴이 네 개 있고 마지막 절에는 두 개밖에 없으므로 3CNF 논리식이 아닙니다. 3SAT은 SAT을 3CNF 논리식으로 제한한 문제입니다. 즉, 3CNF 논리식이 주어졌을 때 논리식의 값을 TRUE로 만드는 변수 할당이 존재합니까?

더 일반적인 SAT 문제로부터의 환원을 통해 3SAT이 NP-난해임을 증명할 수도 있지만, 실제로는 처음부터 새로 시작하여 CIRCUITSAT으로부터 직접 환원하는 편이 더 쉽습니다.

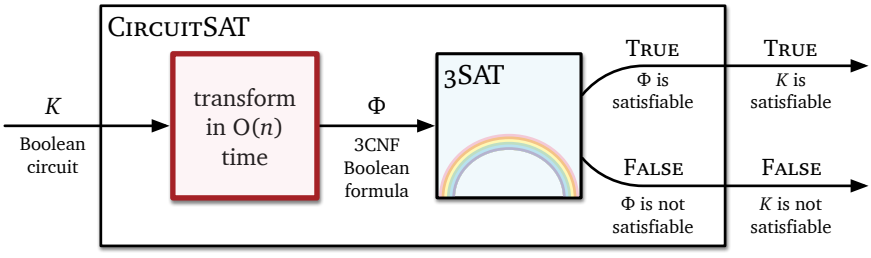


그림 12.6. CIRCUITSAT에서 3SAT으로의 다항 시간 환원입니다.

임의의 불 회로 K가 주어지면, 여러 단계를 거쳐 K를 동치인 3CNF 논리식으로 변환합니다. 맨 마지막 단계를 제외하면, 이 환원은 사실 쿡과 레빈이 쿡-레빈 정리의 증명을 발표하기 5년 전인 1966년에 그리고리 체이틴(Grigorii Tseitin)이 이미 기술한 것입니다. (같은 1966년 논문에서 체이틴은 오늘날 우리가 CNF-SAT라고 부르는 문제를 처음으로 기술했을 가능성이 있습니다.) 각 단계를 설명하면서 그 단계가 올바르다는 것도 함께 증명하겠습니다.

- *K에 있는 모든 AND 게이트와 OR 게이트가 정확히 두 개의 입력을 갖도록 합니다.* 어떤 게이트의 입력이 $k > 2$개이면, 그 게이트를 $k-1$개의 이진 게이트로 이루어진 이진 트리로 대체합니다. 이렇게 얻은 회로를 K'이라고 합시다. 회로 K와 K'은 논리적으로 동치인 회로이므로, K를 만족시키는 모든 입력은 K'도 만족시키며 그 역도 성립합니다.

- *K'를 게이트마다 절 하나를 갖는 불 논리식 Φ_1로 옮겨 적습니다.* 이는 앞서 SAT로 환원할 때와 정확히 같습니다. K'에 대한 모든 만족하는 입력은 Φ_1에 대한 만족하는 할당으로 변환될 수 있고 그 역도 성립한다는 것을 우리는 이미 증명했습니다.

- *Φ_1의 각 절을 CNF 논리식으로 바꾸십시오.* Φ_1에는 세 가지 유형의 절만 있으며, 각 유형은 K'의 게이트 유형 하나에 대응합니다:

$$\begin{aligned} a = b \wedge c &\longmapsto (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c) \\ a = b \vee c &\longmapsto (\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}) \\ a = \bar{b} &\longmapsto (a \vee b) \wedge (\bar{a} \vee \bar{b}) \end{aligned}$$

결과로 얻은 CNF 논리식을 Φ_2라고 부릅시다. Φ_1과 Φ_2는 논리적으로 동치인 논리식이므로, Φ_1을 만족하는 모든 할당은 Φ_2를 만족하는 할당이기도 하며, 그 역도 성립합니다.

- Φ_2*의 각 절을 3CNF 식으로 바꿉니다.* Φ_2의 모든 절에는 리터럴이 *최대* 세 개 있지만, 우리에게 필요한 것은 리터럴이 *정확히* 세 개인 절입니다. 3CNF 식을 얻기 위해, Φ_2에서 리터럴이 두 개인 각 절은 새 변수 하나를 도입하여 리터럴이 세 개인 절 두 개로 확장하고, Φ_2의 마지막 절인 리터럴이 하나인 절은 새 변수 두 개를 도입하여 리터럴이 세 개인 절 네 개로 확장합니다.

$$\begin{aligned} a \vee b &\longmapsto (a \vee b \vee x) \wedge (a \vee b \vee \bar{x}) \\ z &\longmapsto (z \vee x \vee y) \wedge (z \vee \bar{x} \vee y) \wedge (z \vee x \vee \bar{y}) \wedge (z \vee \bar{x} \vee \bar{y}) \end{aligned}$$

  이렇게 얻은 3CNF 식을 Φ_3이라고 합시다. Φ_2를 만족하는 모든 할당은 새 변수(x와 y)에 *임의의* 값을 할당하여 Φ_3을 만족하는 할당으로 바꿀 수 있습니다. 반대로, Φ_3을 만족하는 모든 할당은 새 변수를 무시하여 Φ_2를 만족하는 할당으로 바꿀 수 있습니다.

예를 들어, 앞의 예제 회로는 다음과 같은 3CNF 논리식으로 변환됩니다. 그림 12.5와 비교해 보십시오.

$$\begin{gathered}
(y_1 \vee \overline{x_1} \vee \overline{x_4}) \wedge (\overline{y_1} \vee x_1 \vee z_1) \wedge (\overline{y_1} \vee x_1 \vee \overline{z_1}) \wedge (\overline{y_1} \vee x_4 \vee z_2) \wedge (\overline{y_1} \vee x_4 \vee \overline{z_2}) \\
\wedge (y_2 \vee x_4 \vee z_3) \wedge (y_2 \vee x_4 \vee \overline{z_3}) \wedge (\overline{y_2} \vee \overline{x_4} \vee z_4) \wedge (\overline{y_2} \vee \overline{x_4} \vee \overline{z_4}) \\
\wedge (y_3 \vee \overline{x_3} \vee \overline{y_2}) \wedge (\overline{y_3} \vee x_3 \vee z_5) \wedge (\overline{y_3} \vee x_3 \vee \overline{z_5}) \wedge (\overline{y_3} \vee y_2 \vee z_6) \wedge (\overline{y_3} \vee y_2 \vee \overline{z_6}) \\
\wedge (\overline{y_4} \vee y_1 \vee x_2) \wedge (y_4 \vee \overline{x_2} \vee z_7) \wedge (y_4 \vee \overline{x_2} \vee \overline{z_7}) \wedge (y_4 \vee \overline{y_1} \vee z_8) \wedge (y_4 \vee \overline{y_1} \vee \overline{z_8}) \\
\wedge (y_5 \vee x_2 \vee z_9) \wedge (y_5 \vee x_2 \vee \overline{z_9}) \wedge (\overline{y_5} \vee \overline{x_2} \vee z_{10}) \wedge (\overline{y_5} \vee \overline{x_2} \vee \overline{z_{10}}) \\
\wedge (y_6 \vee x_5 \vee z_{11}) \wedge (y_6 \vee x_5 \vee \overline{z_{11}}) \wedge (\overline{y_6} \vee \overline{x_5} \vee z_{12}) \wedge (\overline{y_6} \vee \overline{x_5} \vee \overline{z_{12}}) \\
\wedge (\overline{y_7} \vee y_3 \vee y_5) \wedge (y_7 \vee \overline{y_3} \vee z_{13}) \wedge (y_7 \vee \overline{y_3} \vee \overline{z_{13}}) \wedge (y_7 \vee \overline{y_5} \vee z_{14}) \wedge (y_7 \vee \overline{y_5} \vee \overline{z_{14}}) \\
\wedge (y_8 \vee \overline{y_4} \vee \overline{y_7}) \wedge (\overline{y_8} \vee y_4 \vee z_{15}) \wedge (\overline{y_8} \vee y_4 \vee \overline{z_{15}}) \wedge (\overline{y_8} \vee y_7 \vee z_{16}) \wedge (\overline{y_8} \vee y_7 \vee \overline{z_{16}}) \\
\wedge (y_9 \vee \overline{y_8} \vee \overline{y_6}) \wedge (\overline{y_9} \vee y_8 \vee z_{17}) \wedge (\overline{y_9} \vee y_6 \vee z_{18}) \wedge (\overline{y_9} \vee y_6 \vee \overline{z_{18}}) \wedge (\overline{y_9} \vee y_8 \vee \overline{z_{17}}) \\
\wedge (y_9 \vee z_{19} \vee z_{20}) \wedge (y_9 \vee \overline{z_{19}} \vee z_{20}) \wedge (y_9 \vee z_{19} \vee \overline{z_{20}}) \wedge (y_9 \vee \overline{z_{19}} \vee \overline{z_{20}})
\end{gathered}$$

이런! 언뜻 보면 이 논리식은 원래 회로보다 *훨씬* 더 복잡해 보일 수 있지만, 실제로는 상수 배만큼만 더 클 뿐입니다. 구체적으로, 단순화된 회로 K'의 전선 수는 많아야 원래 회로 K의 두 배이며, K'의 각 이진 게이트는 Φ_3에서 많아야 다섯 개의 절로 변환됩니다. 설령 논리식의 크기가 회로 크기에 대한 큰 *다항* 함수(n^{374} 같은)였다 하더라도, 여전히 유효한 환원이었을 것입니다.

우리의 환원은 임의의 불 회로 K를 다항 시간 안에(사실은 선형 시간 안에) 3CNF 논리식 Φ_3로 변환합니다. 게다가 입력 회로 K를 만족하는 입력은 모두 Φ_3에 대한 만족하는 할당으로 변환할 수 있고, Φ_3에 대한 만족하는 할당은 모두 K를 만족하는 입력으로 변환할 수 있습니다. 다시 말해, K가 충족 가능한 것은 Φ_3가 충족 가능한 것과 필요충분조건입니다. 따라서 3SAT를 다항 시간 안에 풀 수 있다면 CIRCUITSAT도 다항 시간 안에 풀 수 있고, 이는 P = NP임을 뜻합니다. 그러므로 3SAT는 NP-난해라고 결론 내릴 수 있습니다.

12.7 최대 독립 집합 (3SAT로부터)

이제부터 살펴볼 몇 가지 문제에서 입력은 단순하고 가중치가 없는 무방향 그래프이며, 문제는 어떤 구조적 성질을 만족하는 가장 큰 부분 그래프 또는 가장 작은 부분 그래프의 크기를 구하는 것입니다.

G를 임의의 그래프라고 합시다. G의 ***독립 집합***은 서로 간에 간선이 하나도 없는 G의 정점들로 이루어진 부분집합입니다. *최대 독립 집합* 문제는 주어진 그래프에서 가장 큰 독립 집합의 크기를 구하는 문제이며, 앞으로 ***MaxIndSet***으로 줄여 부르겠습니다. 그림 12.7에서 보인 것처럼, 3Sat으로부터의 환원을 이용하여 MaxIndSet이 NP-난해임을 증명하겠습니다.

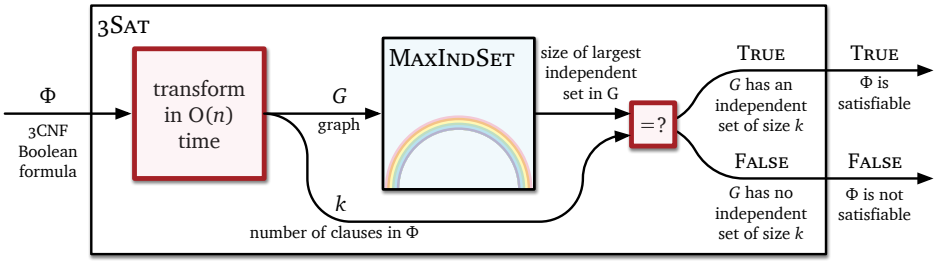
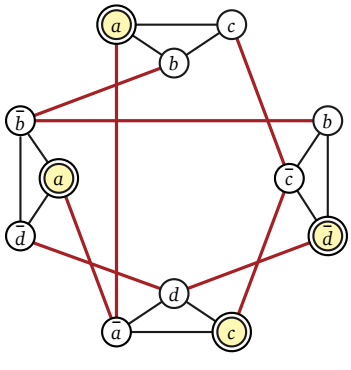


그림 12.7. 3Sat에서 MaxIndSet으로의 다항 시간 환원.

임의의 3CNF 논리식 Φ가 주어지면 다음과 같이 그래프 G를 구성합니다. k를 Φ에 있는 절의 개수라고 합시다. 그래프 G는 Φ의 리터럴마다 하나씩, 정확히 $3k$개의 정점을 포함합니다. G의 두 정점은 (1) 같은 절에 있는 리터럴에 대응하거나 (2) 어떤 변수와 그 부정에 대응할 때, 그리고 오직 그때에만 간선으로 연결됩니다. 예를 들어, 논리식 $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$는 그림 12.8에 나타난 그래프로 변환됩니다.



$$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$$

그림 12.8. 절 4개로 이루어진 충족 가능한 3CNF 논리식에서 유도된 그래프와 크기가 4인 독립 집합입니다.

G의 모든 독립 집합은 각 절 삼각형에서 정점을 최대 하나만 포함합니다. 각 삼각

형에 속한 두 정점은 어느 쌍이든 서로 연결되어 있기 때문입니다. 따라서 G에서 가장 큰 독립 집합의 크기는 *많아야* k입니다. 저는 원래 논리식 Φ가 충족 가능할 때, 그리고 오직 그때에만 G가 크기가 *정확히* k인 독립 집합을 포함한다고 주장합니다. " 일 때, 그리고 오직 그때에만" 형태의 명제가 늘 그렇듯이, 증명은 두 부분으로 이루어집니다.

⇒ Φ가 충족 가능하다고 가정합시다. 만족하는 할당을 하나 임의로 고정합니다. 정의에 따라 Φ의 각 절은 True인 리터럴을 적어도 하나 포함합니다. 따라서 G에서 k개의 정점으로 이루어진 부분집합 S를 고를 수 있습니다. 이 S는 절 삼각형마다 정확히 하나의 정점을 포함하고, 그에 대응하는 k개의 리터럴은 모두 True입니다. 각 삼각형에는 S에 속한 정점이 많아야 하나이므로, S의 어떤 두 정점도 삼각형 간선으로 연결되지 않습니다. S의 정점에 대응하는 모든 리터럴이 True이므로, S의 어떤 두 정점도 부정 간선으로 연결되지 않습니다. 따라서 S는 G에서 크기가 k인 독립 집합입니다.

⇐ 반면 G가 크기 k인 독립 집합 S를 포함한다고 가정합시다. S의 각 정점은 서로 다른 절 삼각형에 있어야 합니다. S에 속한 각 리터럴에 True 값을 할당한다고 가정합시다. 서로 모순되는 리터럴은 간선으로 연결되어 있으므로 이 할당은 일관적입니다. x와 $\bar{x}$ 모두 S의 정점에 대응하지 않는 변수 x가 있을 수 있는데, 이러한 변수에는 원하는 값을 아무것이나 지정할 수 있습니다. S는 각 절 삼각형마다 정점을 하나씩 포함하므로, Φ의 각 절에는 True인 리터럴이 (적어도) 하나 있습니다. 따라서 Φ는 충족 가능하다는 결론을 얻습니다.

3CNF 논리식 Φ를 그래프 G로 변환하는 데는 모든 것을 무차별 대입으로 처리하더라도 다항 시간이 걸립니다. 따라서 MaxIndSet을 다항 시간에 풀 수 있다면, 입력 논리식 Φ를 그래프 G로 변환하고 G에서 가장 큰 독립 집합의 크기를 Φ의 절 개수와 비교함으로써 3Sat도 다항 시간에 풀 수 있을 것입니다. 하지만 그렇다면 P=NP가 성립하게 되는데, 이는 터무니없는 일입니다! 따라서 MaxIndSet은 NP-난해하다고 결론 내립니다.

12.8 일반적인 패턴

모든 NP-난해성 증명은—더 일반적으로는 모든 다항 시간 환원은—같은 일반적인 틀을 따릅니다. 문제 X를 문제 Y로 다항 시간에 환원하려면 다음 세 가지를 해야 합니다:

1. X의 ***임의의*** 인스턴스 x를 Y의 *특수한* 인스턴스 y로 변환하는 다항 시간 알고리즘을 설명하십시오.
2. x가 X의 "좋은" 인스턴스이면 y가 Y의 "좋은" 인스턴스임을 증명하십시오.
3. y가 Y의 "좋은" 인스턴스이면 x도 X의 "좋은" 인스턴스임을 증명하십시오. (보통 이 부분이 가장 까다롭습니다.)

물론 올바른 환원을 *개발한다*는 것이 이 세 가지 작업을 하나씩 차례로 처리한다는 뜻은 아닙니다. 작동하는 *것처럼 보이는* 알고리즘을 *먼저* 적어 두고 *그다음에* 그것이 *실제로* 작동함을 증명하려 하는 방식은, 특히 시험처럼 시간이 제한된 상황에서는 거

의 성공하지 못합니다. 우리는 알고리즘과 "if" 증명, "only if" 증명을 *반드시* 동시에 개발해야 합니다.

작고한 위대한 리키 제이(Ricky Jay)의 말을 인용하겠습니다:[9] 이것은 *후천적으로 습득하는* 기술입니다.

많은 학생이 혼란스러워하는 점은 환원 알고리즘은 "한 방향으로만 작동"하는데—X에서 Y로—정확성 증명은 "양방향으로 작동"해야 한다는 것입니다. 그러나 정확성 증명은 실제로 대칭적이지 않습니다. " 이면(if)" 증명은 X의 *임의의* 인스턴스를 다루어야 하지만, " 일 때만(only if)" 증명은 환원 알고리즘이 만들어 내는 Y의 *특수한* 인스턴스만 다루면 됩니다. 이 비대칭성을 활용하는 것이 올바른 환원을 성공적으로 설계하는 핵심입니다.

저는 인스턴스 자체와 함께 *증명서*—주어진 인스턴스가 "좋다"는 증명—를 변환한다는 관점에서 생각하는 것이 유용하다고 봅니다. 예를 들어, CircuitSat의 증명서는 전구를 켜는 입력의 집합이고, Sat나 3Sat의 증명서는 만족하는 할당이며, MaxIndSet의 증명서는 큰 독립 집합입니다. X를 Y로 환원하려면 실제로는 다음 각 작업에 하나씩, 모두 *세 개*의 알고리즘을 설계해야 합니다.

- X의 임의의 인스턴스 x를 다항 시간 안에 Y의 특수한 인스턴스 y로 변환합니다.
- x에 대한 임의의 증명서를 y에 대한 증명서로 변환하고,
- y에 대한 임의의 증명서를 x에 대한 증명서로 변환하십시오.

두 번째와 세 번째 작업은 첫 번째 알고리즘의 입력과 출력에 관한 것입니다. 가역적이어야 하는 것은 *인스턴스* 변환이 아니라 *증명서* 변환입니다. Y의 인스턴스를 변환해야 하는 경우는 결코 없으며, Y의 *임의의* 인스턴스에 대해서는 아예 생각할 필요도 없습니다. 다항 시간에 실행되어야 하는 것은 첫 번째 알고리즘뿐입니다(물론 실제로는 두 번째와 세 번째 알고리즘이 첫 번째 알고리즘보다 거의 항상 더 단순합니다).

예를 들어, CircuitSat에서 3Sat으로의 환원은 다음 세 가지 알고리즘으로 구성됩니다:

- 첫 번째 환원은 *임의의* 불 회로 K를 특수한 3CNF 불 논리식 Φ_3로 다항 시간 안에 변환합니다. (각 전선은 변수로, 각 게이트는 부분 논리식으로 인코딩한 다음, 각 부분 논리식을 3CNF로 전개합니다.)
- 두 번째 변환은 K를 만족하는 *임의의* 입력을 Φ_3에 대한 만족하는 할당으로 변환합니다. (입력을 회로를 따라 추적하면서 각 전선의 값을 대응하는 변수에 옮기고, 추가된 변수들에는 임의의 값을 부여하십시오.)
- 세 번째는 Φ_3에 대한 *임의의* 만족하는 할당을 K에 대한 만족하는 입력으로 변환합니다. (Φ_3의 각 전선 변수의 값을 K의 대응하는 전선으로 옮기십시오.)

이 환원이 성립하는 이유는 첫 번째 알고리즘이 임의의 불 회로 K를 고도로 구조화된 3CNF 논리식 Φ_3로 *인코딩*하기 때문입니다. Φ_3의 특정한 구조는 그것이 *어떻게* 충족될 수 있는지를 제한합니다. Φ_3에 대한 모든 만족하는 할당은 K에 대한 어떤 만족하는

[9] 그의 1996년 오프브로드웨이 공연 *Ricky Jay and his 52 Assistants*에서 인용했습니다.

입력으로부터 "나와야" 합니다. 우리는 임의의 3CNF 논리식에 대해서는 전혀 생각할 필요가 없습니다.

마찬가지로, 3Sat에서 MaxIndSet으로의 환원은 세 가지 알고리즘으로 구성됩니다.

- 첫 번째 환원은 *임의의* 3CNF 논리식 Φ를 다항 시간 안에 특수한 그래프 G와 특정 정수 k로 변환합니다.
- 두 번째 함수는 Φ에 대한 *임의의* 만족하는 할당을 G에서 크기가 k인 독립 집합으로 변환합니다.
- 세 번째는 G에서 크기가 k인 *임의의* 독립 집합을 Φ를 만족하는 할당으로 변환합니다.

다시 말해, 첫 번째 변환은 입력 논리식 Φ를 고도로 구조화된 그래프 G와 특정 정수 k로 *인코딩*합니다. G의 구조는 크기가 k인 모든 독립 집합이 Φ의 만족하는 할당으로부터 "나온다"는 것을 보장합니다. 우리는 임의의 그래프나 임의의 독립 집합 크기는 전혀 고려하지 않습니다.

12.9 클리크와 정점 덮개 (독립 집합으로부터)

클리크는 완전 그래프, 즉 모든 정점 쌍이 간선으로 연결된 그래프를 달리 부르는 이름입니다. MaxClique 문제는 주어진 그래프에서 가장 큰 완전 부분 그래프의 노드 수를 구하는 문제입니다. 그래프의 ***정점 덮개***란 그래프의 모든 간선에 닿는 정점들의 집합입니다. MinVertexCover 문제는 주어진 그래프에서 가장 작은 정점 덮개의 크기를 구하는 문제입니다.

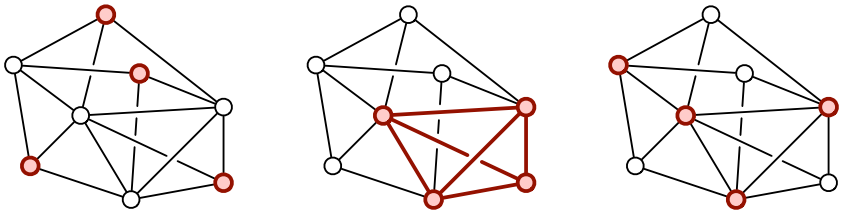


그림 12.9. 가장 큰 독립 집합, 가장 큰 클리크, 가장 작은 정점 덮개의 크기가 모두 4인 그래프입니다.

MaxIndSet으로부터의 다음과 같은 간단한 환원을 이용하면 MaxClique가 NP-난해임을 증명할 수 있습니다. 모든 그래프 G에는 같은 정점들을 가지되 간선 집합은 정확히 반대인 *간선 여그래프* $\overline{G}$가 있습니다—uv가 $\overline{G}$의 간선일 필요충분조건은 uv가 G의 간선이 *아닌* 것입니다. 정점 집합이 G에서 독립일 필요충분조건은 같은 정점들이 $\overline{G}$에서 클리크를 이루는 것입니다. 따라서 G의 최대 독립 집합은 G의 여그래프의 최대 클리크와 같은 정점들을 가지며, 따라서 크기도 같습니다.

MinVertexCover가 NP-난해임을 보이는 증명은 이보다 훨씬 더 간단합니다. 다음과 같은 쉬운 관찰에 기대기 때문입니다. I가 그래프 $G = (V, E)$의 독립 집합인 것은 그 여집합 $V \setminus I$가 같은 그래프 G의 정점 덮개인 것과 동치입니다. 따라서 임의의

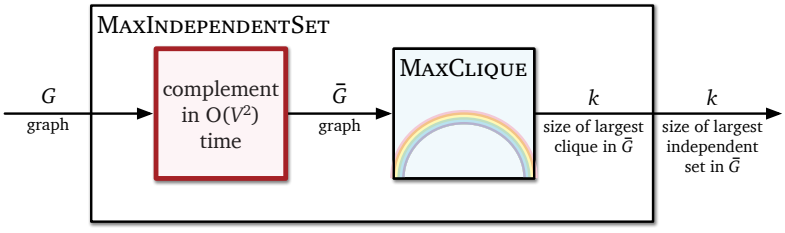


그림 12.10. MaxIndSet에서 MaxClique로의 간단한 환원입니다.

그래프에서 *가장 큰* 독립 집합은 같은 그래프의 *가장 작은* 정점 덮개의 여집합입니다! 그러므로 정점이 n개인 그래프에서 가장 작은 정점 덮개의 크기가 k이면, 가장 큰 독립 집합의 크기는 $n - k$입니다.

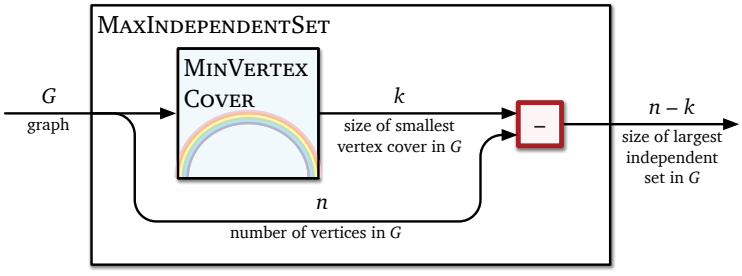
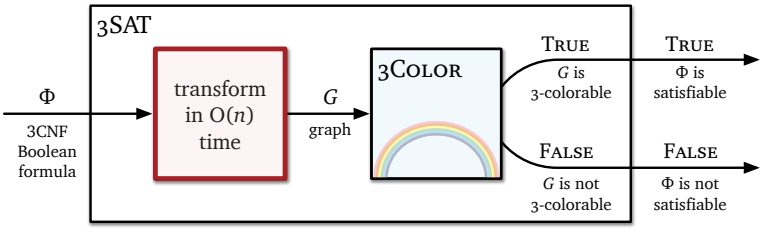


그림 12.11. MaxIndSet에서 MinVertexCover로의 훨씬 더 쉬운 환원입니다.

12.10 그래프 채색(3Sat로부터의 환원)

그래프 $G = (V, E)$의 ***적절한 k-색칠***은 각 정점에 k개의 "색" 중 하나를 배정하여 모든 간선의 두 끝점이 서로 다른 색을 갖도록 하는 함수 $C\colon V \to \{1, 2, \ldots, k\}$입니다. (여기서 "색"은 사실 임의의 레이블일 뿐이며, 편의상 전자기파의 주파수나 CMYK 벡터, 팬톤 번호 같은 것 대신 작은 양의 정수로 나타냅니다.) 그래프 색칠 문제는 주어진 그래프를 올바르게 색칠하는 데 필요한 색의 최소 개수를 구하는 문제입니다.

그래프 채색이 NP-난해임을 증명하려면 결정 문제 ***3Color***만 고려하면 충분합니다. 즉, 그래프가 주어졌을 때 그 그래프에 적절한 3-채색이 존재합니까? 3Sat로부터의 환원을 이용하여 3Color가 NP-난해임을 증명합니다. (왜 3Sat일까요? 이름에 3이 들어 있기 때문입니다. 아마 제가 농담한다고 생각하시겠지만, 농담이 아닙니다.) 3CNF 논리식 Φ가 주어지면, 늘 보던 그림이 보여 주듯이 Φ가 충족 가능할 때, 그리고 오직 그때에만 3-채색 가능한 그래프 G를 구성합니다.



이 환원은 출력 그래프 G를 *가젯(gadget)*으로 분해하는 표준 전략을 사용하여 설명합니다. 가젯은 입력 논리식 Φ의 여러 의미를 그래프 색칠의 언어로 강제하는 부분 그래프입니다. 환원을 개별 가젯으로 분해하면 기존 환원을 이해하고 그 정확성을 증명하는 데 도움이 될 뿐만 아니라, 새로운 NP-난해성 환원을 설계하는 데에도 도움이 됩니다.[10] 이 논리식-그래프 환원은 세 종류의 가젯을 사용합니다:

- 하나의 *진리 가젯*이 있습니다. 이는 세 정점 T, F, X로 이루어진 삼각형이며, 이 정점들은 직관적으로 각각 True, False, Other를 나타냅니다. 이 정점들은 모두 서로 연결되어 있으므로, 어떤 3-색칠에서든 서로 다른 색을 가져야 합니다. 편의상 이 색들을 True, False, Other라고 *부르겠습니다*. 따라서 어떤 노드가 True로 색칠되었다고 말하는 것은 그 노드가 정점 T와 같은 색을 가진다는 뜻입니다.

- 그래프에는 각 변수 a마다 *변수 가젯*이 있습니다. 변수 가젯은 a와 $\overline{a}$라는 레이블이 붙은 새 노드 두 개를 진리 가젯의 노드 X와 연결하는 삼각형입니다. 노드 a는 True나 False 중 한 색으로 칠해야 합니다. 따라서 노드 $\overline{a}$는 그에 대응하여 각각 False나 True로 칠해야 합니다.

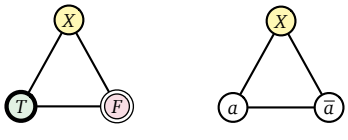


그림 12.12. 참 가젯과 a에 대한 변수 가젯입니다.

- 마지막으로, 그래프에는 Φ의 각 절마다 *절 가젯*이 하나씩 있습니다. 각 절 가젯은 아래 그림과 같이 레이블 없는 새 노드 다섯 개와 간선 열 개를 사용하여 (해당 변수 가젯에서 온) 리터럴 노드 세 개를 (참 가젯에서 온) 노드 T에 연결합니다.

  사실상 절 가젯의 각 삼각형은 "다수결 게이트"처럼 동작합니다. 올바른 3-색칠이라면 어느 것이든, 삼각형 왼쪽에 있는 두 정점의 색이 같으면 삼각형의 가장 오른쪽 정점도 반드시 같은 색이어야 합니다. 반면 왼쪽의 두 정점의 색이 서로 다르면 오른쪽 정점의 색은 임의로 고를 수 있습니다. 그림 12.14를 참고하십시오.

  따라서 세 리터럴 노드가 모두 False로 색칠된 절 가젯의 올바른 3-색칠은 존재하지 않습니다. 반면에 리터럴 노드를 두 가지 이상의 색으로 색칠한 경우에는 어떤

[10] CircuitSat에서 Sat으로의 환원은 입력 회로의 각 게이트를 출력 논리식의 절 하나로 인코딩했습니다. 이 절들이 바로 "게이트 가젯"입니다. 마찬가지로 3Sat에서 MaxIndSet으로의 환원은 두 종류의 가젯, 즉 "절 가젯"(삼각형)과 "변수 가젯"(서로 모순되는 리터럴 사이의 간선)을 사용했습니다.

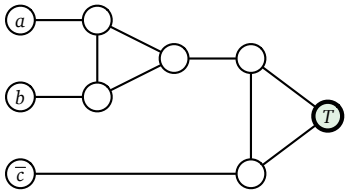


그림 12.13. $(a \vee b \vee \bar{c})$에 대한 절 가젯입니다.

색칠이든 절 가젯의 올바른 3-색칠로 확장할 수 있습니다. 변수 가젯은 각 리터럴 노드가 True 또는 False 중 하나로 색칠되도록 강제합니다. 따라서 절 가젯의 올바른 3-색칠에서는 언제나 적어도 하나의 리터럴 노드가 True로 색칠됩니다.

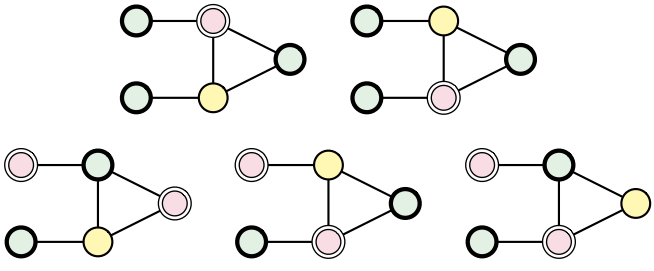


그림 12.14. 색의 순열 차이를 무시한 "반쪽 가젯"의 모든 유효한 3-색칠

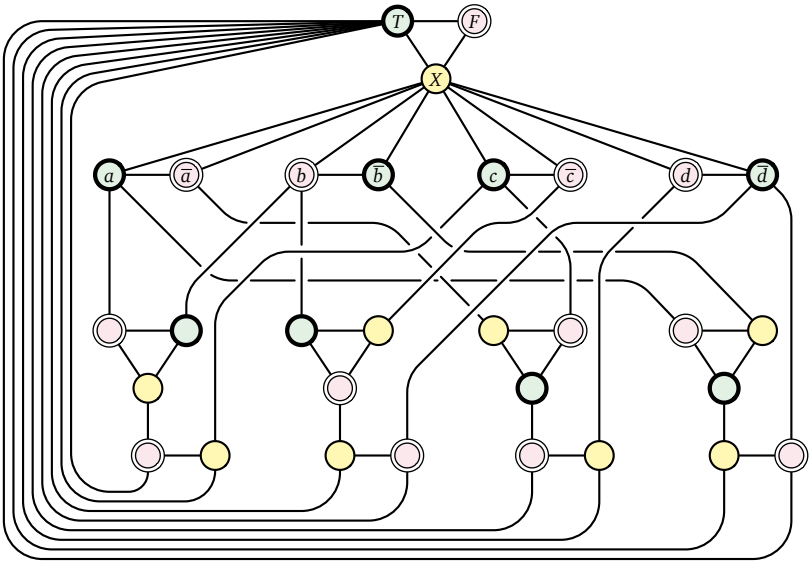
최종 그래프 G에는 노드 T가 정확히 *하나*, 노드 F가 정확히 *하나* 있고, 각 변수마다 노드 a와 $\bar{a}$가 정확히 두 개 있습니다. 예를 들어 그림 12.15는 앞서 그림 12.8에서 MaxIndSet 환원을 설명할 때 사용한 것과 같은 3CNF 논리식 $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$에서 얻어지는 그래프를 보여 줍니다. 그림의 3-색칠은 만족하는 할당 $a = c =$ True, $b = d =$ False에 대응하는 여러 3-색칠 중 하나입니다.

정확성 증명에 필요한 작업은 이미 대부분 마쳤습니다. 논리식이 충족 가능하다면, 임의의 만족하는 할당에 따라 리터럴 노드를 색칠한 다음 (각 절이 만족되므로) 그 색칠을 모든 절 가젯으로 확장할 수 있습니다. 반대로 그래프가 3-색칠 가능하다면, 임의의 3-색칠로부터 만족하는 할당을 추출할 수 있습니다—모든 절 가젯에서 세 리터럴 노드 중 적어도 하나는 True로 색칠되어 있기 때문입니다.

3Color는 더 일반적인 그래프 색칠 문제—최소 몇 가지 색이 필요한가?—의 특수한 경우이므로, 더 일반적인 이 최적화 문제 역시 NP-난해합니다.

12.11 해밀턴 사이클

그래프에서 ***해밀턴 사이클***은 모든 정점을 정확히 한 번씩 방문하는 사이클입니다. (이는 모든 *간선*을 정확히 한 번씩 지나는 닫힌 *보행*인 *오일러 회로*와 매우 다릅니다. 오일러 회로는 깊이 우선 탐색을 사용하여 선형 시간에 쉽게 찾고 구성할 수 있습니다.)



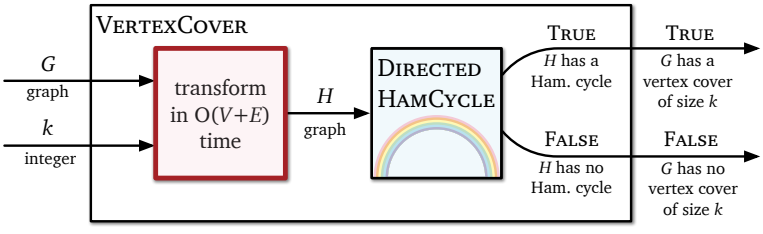
$$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$$

그림 12.15. 충족 가능한 3CNF 논리식으로부터 유도된 3-색칠 가능 그래프.

여기서는 *방향* 그래프에서 해밀턴 사이클 문제가 NP-난해임을 보이는 서로 다른 두 가지 증명을 살펴봅니다.

정점 덮개로부터

첫 번째 NP-난해성 증명은 정점 덮개 문제의 결정 버전으로부터 환원합니다. *무방향* 그래프 G와 정수 k가 주어지면, G에 크기가 k인 정점 덮개가 있을 때 그리고 그때에만 H에 해밀턴 사이클이 있도록 하는 *방향* 그래프 H를 구성합니다. 앞선 환원들에서와 마찬가지로, 출력 그래프 H는 여러 가젯으로 구성되며, 각 가젯은 입력 G와 k의 특정한 특징에 대응합니다.



- G의 각 무방향 간선 uv에 대해, 방향 그래프 H는 네 정점 (u, v, in), (u, v, out), (v, u, in), (v, u, out)과 여섯 개의 방향 간선으로 이루어진 *간선 가젯*을 포함합

니다.

$$(u,v,\text{in})\to(u,v,\text{out}) \qquad (u,v,\text{in})\to(v,u,\text{in}) \qquad (v,u,\text{in})\to(u,v,\text{in})$$
$$(v,u,\text{in})\to(v,u,\text{out}) \qquad (u,v,\text{out})\to(v,u,\text{out}) \qquad (v,u,\text{out})\to(u,v,\text{out})$$

그림 12.16에 나타낸 것과 같습니다. 각 "in" 정점에는 들어오는 간선이 하나 더 있고, 각 "out" 정점에는 나가는 간선이 하나 더 있습니다. H의 모든 해밀턴 사이클은 세 가지 방식 중 하나로 간선 가젯을 통과해야 합니다—양쪽 모두를 곧장 통과하거나, 한쪽에서 다른 쪽으로 우회했다가 되돌아오는 방식입니다. 결국 이 선택지들은 각각 u와 v가 모두, u만, 또는 v만 어떤 정점 덮개에 속하는 경우에 대응하게 됩니다.

- G의 각 정점 u에 대해, u에 접속한 간선 uv들의 간선 가젯은 모두 H에서 하나의 방향 경로로 연결되며, 이를 *정점 사슬*이라고 부릅니다. 구체적으로, 정점 u에 d개의 이웃 $v_1, v_2, \ldots, v_d$가 있다고 합시다. 그러면 H에는 1부터 $d-1$까지의 각 i에 대해 추가 간선 $(u, v_i, \text{out})\to(u, v_{i+1}, \text{in})$, 즉 $d-1$개의 추가 간선이 있습니다.

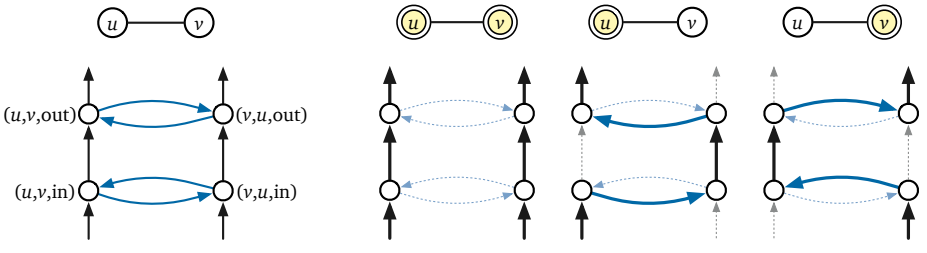


그림 12.16. 간선 가젯과, 해밀턴 사이클이 이 가젯과 교차할 수 있는 유일한 경우들입니다.

- 마지막으로, H에는 k개의 *덮개 정점* $x_1, x_2, \ldots, x_k$도 있습니다. 각 덮개 정점에서 각 정점 사슬의 첫 번째 정점으로 향하는 방향 간선이 있고, 각 정점 사슬의 마지막 정점에서 그 덮개 정점으로 향하는 방향 간선이 있습니다.

그림 12.17은 이 변환의 전체 예시를 보여 줍니다. 양쪽 끝에 화살표가 있는 각 파란색 선분은 방향 간선 한 쌍을 나타냅니다.

늘 그렇듯이, 이 환원이 올바름을 두 단계로 나누어 증명합니다.

⇒ 먼저 $C = \{u_1, u_2, \ldots, u_k\}$가 G의 크기 k인 정점 덮개라고 가정합시다. C를 "인코딩"하는 H의 해밀턴 사이클을 다음과 같이 구성할 수 있습니다. 1부터 k까지의 각 인덱스 i에 대해, 덮개 정점 x_i에서 출발하여 u_i에 대한 정점 사슬을 거쳐 덮개 정점 x_{i+1}(또는 $i = k$이면 덮개 정점 x_1)까지 이르는 경로를 순회합니다. 각 정점 u_i에 대한 사슬을 순회하면서, 각 노드 (u_i, v, in)에서 어떻게 진행할지를 다음과 같이 결정합니다.

  - $v \in C$이면 간선 $(u_i, v, \text{in})\to(u_i, v, \text{out})$를 따라갑니다.
  - $v \notin C$이면 $(u_i, v, \text{in})\to(v, u_i, \text{in})\to(v, u_i, \text{out})\to(u_i, v, \text{out})$를 거쳐 우회합니다.

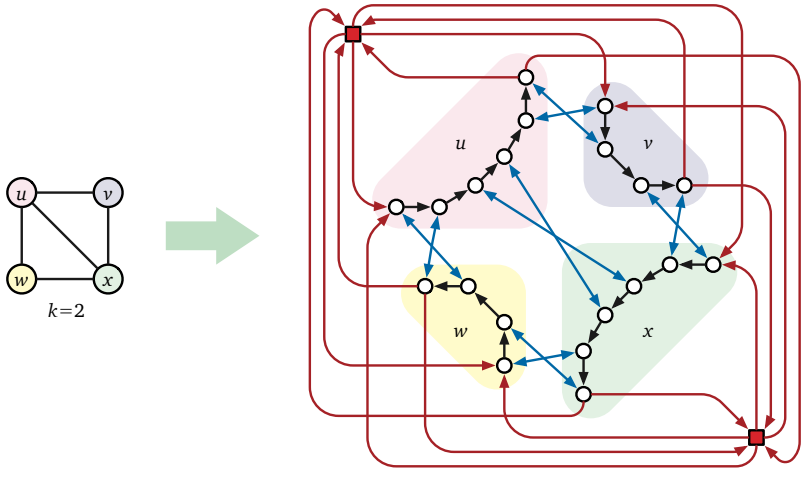


그림 12.17. VertexCover에서 DirectedHamCycle로의 환원 예시입니다.

따라서 G의 각 간선 uv에 대해, 해밀턴 사이클은 $u \in C$이면 u의 정점 사슬의 일부로, 그렇지 않으면 v의 정점 사슬의 일부로 (u, v, in)과 (u, v, out)을 방문합니다. 그림 12.18을 참고하십시오.

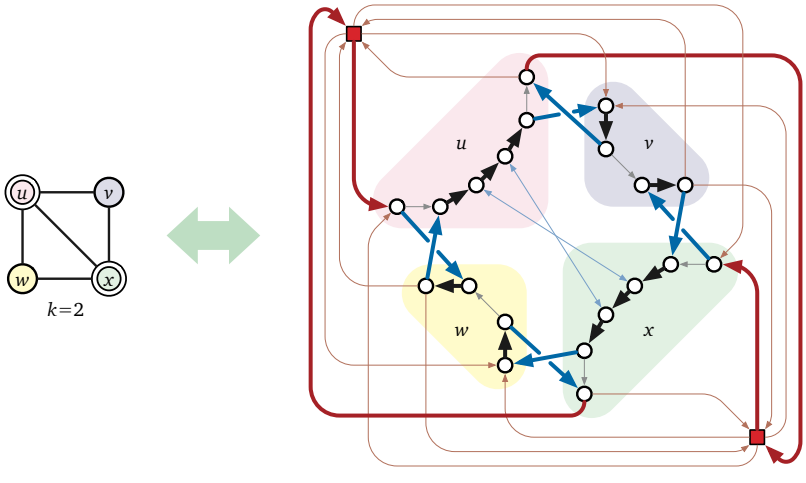


그림 12.18. G에서 크기가 k인 모든 정점 덮개는 H의 해밀턴 사이클에 대응하며, 그 역도 성립합니다.

⇐ 반면에 H가 해밀턴 사이클 C를 포함한다고 가정합시다. 이 사이클은 각 덮개 정점에서 어떤 정점 사슬의 시작으로 가는 간선을 포함해야 합니다. 간선 가젯에 대한 경우 분석으로부터 귀납적으로, C가 어떤 정점 u의 정점 사슬에 들어간 뒤에는 그 정점 사슬 전체를 순회해야 함을 알 수 있습니다. 구체적으로, 각 정점 (u, v, in)에서 사이클은 단일 간선 $(u, v, \text{in}) \to (u, v, \text{out})$ 또는 우회 경로 $(u, v, \text{in}) \to (v, u, \text{in}) \to (v, u, \text{out}) \to (u, v, \text{out})$ 중 하나를 포함해야 하며, 그다음에는 u의 정점 사슬에 있는 다음 간선 가젯으로 가는 간선이 이어지거나, 이것이 u의

정점 사슬의 마지막 간선 가젯이라면 덮개 정점으로 가는 간선이 이어져야 합니다. 특히 C가 우회 간선 $(u, v, \text{in}) \to (v, u, \text{in})$을 포함한다면, C는 어떤 덮개 정점과 v의 정점 사슬 사이의 간선도 포함할 수 없습니다. 따라서 C는 정확히 k개의 정점 사슬을 순회합니다. 더욱이 C는 G의 모든 간선 uv에 대해 정점 (u, v, in)을 방문하므로, 이 정점 사슬들은 원래 그래프 G의 정점 덮개를 나타냅니다.

따라서 G에 크기가 k인 정점 덮개가 존재하는 것은 H에 해밀턴 사이클이 존재하는 것과 동치라는 결론을 얻습니다. G를 H로 변환하는 데에는 최대 $O(V^2)$ 시간이 걸리므로, 방향 해밀턴 사이클 문제는 NP-난해입니다.

3Sat에서

3Sat에서 직접 환원하여 방향 해밀턴 사이클 문제가 NP-난해임을 증명할 수도 있습니다. n개의 변수 $x_1, x_2, \ldots, x_n$과 k개의 절 $c_1, c_2, \ldots, c_k$를 갖는 임의의 3CNF 논리식 Φ가 주어지면, Φ가 충족 가능할 때 그리고 오직 그때에만 해밀턴 사이클을 포함하는 방향 그래프 H를 다음과 같이 구성합니다.

각 변수 x_i에 대해 *변수 가젯*을 구성합니다. 변수 가젯은 $2k$개의 정점 $(i, 0), (i, 1), \ldots, (i, 2k)$로 이루어진 이중 연결 리스트이며, 이 정점들은 각 인덱스 j에 대해 간선 $(i, j-1) \to (i, j)$와 $(i, j) \to (i, j-1)$로 연결됩니다. 서로 인접한 변수 가젯 쌍마다 첫 번째 노드와 마지막 노드를 다음 간선들을 추가하여 연결합니다

$$(i, 0) \to (i+1, 0) \quad (i, 2k) \to (i+1, 0) \quad (i, 0) \to (i+1, 2k) \quad (i, 2k) \to (i+1, 2k)$$

이는 각 인덱스 i에 대해 적용됩니다. 또한 첫 번째 변수 가젯과 마지막 변수 가젯의 끝점들을 다음 간선들로 연결합니다

$$(n, 0) \to (1, 0) \qquad (n, 2k) \to (1, 0) \qquad (n, 0) \to (1, 2k) \qquad (n, 2k) \to (1, 2k).$$

그 결과로 얻는 그래프 G에는 정확히 2^n개의 해밀턴 사이클이 있으며, 이는 Φ의 n개 변수에 불 값을 할당하는 각 방법마다 하나씩 대응합니다. 구체적으로, 각 i에 대해 $x_i = \text{True}$이면 i번째 변수 가젯을 왼쪽에서 오른쪽으로 순회하고, $x_i = \text{False}$이면 오른쪽에서 왼쪽으로 순회합니다. 그림 12.19를 참고하십시오.

이제 각 절 c_j마다 그림 12.20에 나타낸 것처럼 여섯 개의 간선으로 변수 가젯에 연결되는 *절 정점* $[j]$를 추가하여 G를 더 큰 그래프 H로 확장합니다. c_j에 있는 각 양의 리터럴 x_i에 대해서는 간선 $(i, 2j-1) \to [j] \to (i, 2j)$를 추가하고, c_j에 있는 각 음의 리터럴 $\bar{x}_i$에 대해서는 간선 $(i, 2j) \to [j] \to (i, 2j-1)$를 추가합니다. 절 정점에 대한 이러한 연결은 대응하는 변수 할당이 Φ를 만족할 때, 그리고 오직 그때에만 G의 해밀턴 사이클을 H의 해밀턴 사이클로 확장할 수 있음을 보장합니다. 이제 모든 경우를 빠짐없이 분석하면 Φ가 충족 가능할 때, 그리고 오직 그때에만 H에 해밀턴 사이클이 존재함을 알 수 있습니다.

식 Φ를 그래프 H로 변환하는 데에는 $O(kn)$ 시간이 걸리며, 이는 식의 전체 길이에 대해 많아야 이차입니다. 따라서 방향 해밀턴 사이클 문제는 NP-난해라는 결론을 얻습니다.

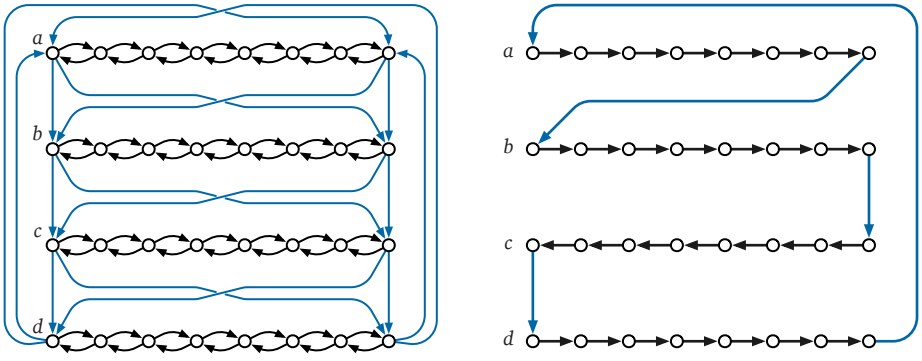


그림 12.19. 왼쪽: 변수가 4개이고 절이 4개인 임의의 논리식에 대해 G에 있는 변수 가젯과 연결자입니다. 오른쪽: 할당 $a = b = d =$ TRUE 및 $c =$ FALSE에 대응하는 G의 해밀턴 사이클입니다.

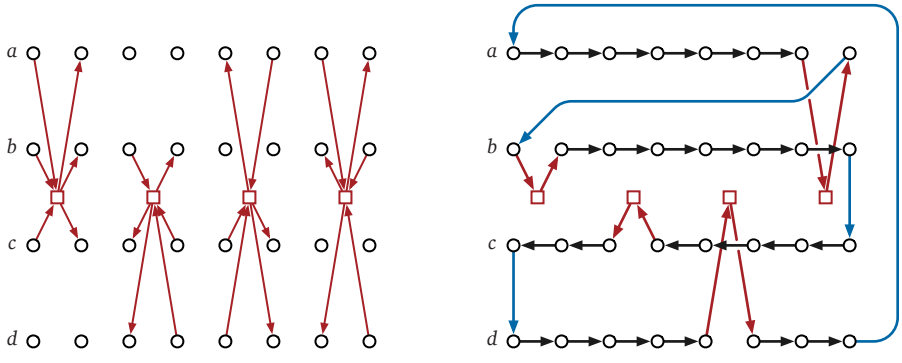


그림 12.20. 왼쪽: 논리식 $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$에 대한 절 가젯입니다. 오른쪽: 만족하는 할당 $a = b = d =$ TRUE 및 $c =$ FALSE에 대응하는 H의 해밀턴 사이클입니다.

변형과 확장

앞의 환원을 사소하게 수정하면 방향 그래프에서의 해밀턴 *경로* 문제 역시 NP-난해임을 알 수 있습니다. 그래프 G의 해밀턴 경로란 물론 G의 모든 정점을 정확히 한 번씩 방문하는 단순 경로입니다. 사실 해밀턴 사이클 문제에서 해밀턴 경로 문제로의 간단한 다항 시간 환원이 존재하며, 그 역방향의 환원도 존재합니다. 이 환원들의 세부 사항은 연습 문제로 남겨 두겠습니다.

앞의 두 환원은 모두 방향 그래프를 다루지만, 무방향 그래프에서 이에 대응하는 문제 역시 NP-난해합니다. 실제로 방향 해밀턴 사이클/경로 문제에서 무방향 해밀턴 사이클/경로 문제로 가는 비교적 간단한 환원이 존재합니다. 이번에도 이 환원의 세부 사항은 연습 문제로 남겨 두겠습니다.

마지막으로, 악명 높은 *외판원 문제*는 가중치가 있는 간선으로 이루어진 그래프에서 가장 짧은 해밀턴 사이클(또는 경로)을 찾는 문제입니다. 가장 짧은 사이클/경로를 찾는 것은 사이클/경로가 애초에 존재하는지 판별하는 것보다 분명히 더 어려우므로—모든 간선의 가중치가 1인 그래프를 생각해 보십시오!—TRAVELINGSALESMAN 역시 NP-난해입니다.

12.12 부분집합 합 (정점 덮개로부터)

다음으로 NP-난해임을 증명할 문제는 2장에서 다룬 SubsetSum 문제입니다. 양의 정수로 이루어진 집합 X와 정수 T가 주어질 때, 원소의 합이 T인 X의 부분집합이 존재하는지 판정하십시오.

이번에도 VertexCover로부터 환원합니다. 그래프 G와 정수 k가 주어지면, 양의 정수들의 집합 X와 정수 T를 계산해야 합니다. 이때 X에 합이 T인 부분집합이 존재하는 것은 G에 크기 k인 정점 덮개가 존재할 때이며, 오직 그때뿐이어야 합니다. 이 변환은 단 두 종류의 "가젯"만을 사용하며, 이들은 G의 정점과 간선을 나타내는 정수입니다.

G의 *간선*에 0부터 $E-1$까지 임의로 번호를 매깁니다. 집합 X는 각 간선 i에 대한 정수 $b_i := 4^i$와 함께 다음 정수를 포함합니다.

$$a_v := 4^E + \sum_{i \in \Delta(v)} 4^i$$

각 정점 v에 대해 이와 같으며, 여기서 $\Delta(v)$는 v를 끝점으로 갖는 간선들의 집합입니다. 달리 말하면, X의 각 정수를 4진법으로 쓴 $(E+1)$자리 수로 생각할 수 있습니다. E번째 자릿수는 그 정수가 정점을 나타내면 1이고, 그렇지 않으면 0입니다. 또한 각 $i < E$에 대해, i번째 자릿수는 그 정수가 간선 i 또는 그 끝점 중 하나를 나타내면 1이고, 그렇지 않으면 0입니다. 마지막으로, 대상 합을 다음과 같이 설정합니다.

$$T := k \cdot 4^E + \sum_{i=0}^{E-1} 2 \cdot 4^i.$$

이제 이 환원이 올바름을 증명합시다.

⇒ 먼저, G에 크기가 k인 정점 덮개 C가 있다고 가정합시다. 다음 부분집합을 생각해 봅시다

$$X' := \{a_v \mid v \in C\} \cup \{b_i \mid \text{간선} i \text{에 끝점이 정확히 하나 있습니다} C\}$$

X'의 원소들의 합을 4진법으로 쓰면 최상위 자릿수는 k이고 나머지 자릿수는 모두 2입니다. 따라서 X'의 원소들의 합은 정확히 T입니다.

⇐ 반면에, 합이 T가 되는 부분집합 $X' \subseteq X$가 존재한다고 가정합시다. 구체적으로 다음이 성립해야 합니다.

$$\sum_{v \in V'} a_v + \sum_{i \in E'} b_i = t$$

여기서 $V' \subseteq V$와 $E' \subseteq E$는 적절한 부분집합입니다. 앞에서와 마찬가지로, 이 4진수들을 더하면 처음 E개의 자리에서는 받아올림이 발생하지 않습니다. 각 i에 대해 X에서 i번째 자리가 1인 수는 세 개뿐이기 때문입니다. 각 간선 수 b_i는 합의 i번째 자리에 1을 하나만 더하지만, t의 i번째 자리는 2입니다. 따라서 G의 각 간선에 대해 그 끝점 중 적어도 하나는 V'에 속해야 합니다. 다시 말해, V'는 정점 덮개입니다. 한편, 4^E보다 큰 수는 정점 수뿐이며 $\lfloor T/4^E \rfloor = k$이므로, V'의 원소는 많아야 k개입니다. (사실 V'의 원소가 *정확히* k개라는 것도 어렵지 않게 알 수 있습니다.)

예를 들어, $V = \{u, v, w, x\}$이고 $E = \{uv, uw, vw, vx, wx\}$인 정점 네 개짜리 그래프 $G = (V, E)$가 주어지면, 집합 X에는 다음과 같은 4진 정수들이 포함될 수 있습니다:

$$\begin{aligned} a_u &:= 111000_4 = 1344 & b_{uv} &:= 010000_4 = 256 \\ a_v &:= 110110_4 = 1300 & b_{uw} &:= 001000_4 = 64 \\ a_w &:= 101101_4 = 1105 & b_{vw} &:= 000100_4 = 16 \\ a_x &:= 100011_4 = 1029 & b_{vx} &:= 000010_4 = 4 \\ & & b_{wx} &:= 000001_4 = 1 \end{aligned}$$

크기가 $k = 2$인 정점 덮개를 찾는다면 대상 합은 $T := 222222_4 = 2730$입니다. 실제로 정점 덮개 $\{v, w\}$는 부분집합 $\{a_v, a_w, b_{uv}, b_{uw}, b_{vx}, b_{wx}\}$에 대응하며, 이 부분집합의 합은 $1300 + 1105 + 256 + 64 + 4 + 1 = 2730$입니다.

이 환원은 분명히 다항 시간에 수행할 수 있습니다. VertexCover가 NP-난해임은 이미 증명했으므로, SubsetSum 역시 NP-난해임이 따라 나옵니다.

환원하는 자는 주의하십시오!

여기서 한 가지 미묘한 점을 강조해야 합니다. 300여 쪽 전, 3장에서 우리는 SubsetSum을 $O(nT)$ 시간에 푸는 동적 계획법 알고리즘을 개발했습니다. 이것은 다항 시간 알고리즘이 아닙니까? 우리가 방금 P=NP임을 증명한 것 아닙니까? 아니, 그럼 제 백만 달러는 어디 있습니까?!

아쉽게도 삶은 그렇게 단순하지 않습니다. 실행 시간이 변수 n과 T에 대한 다항 함수인 것은 사실이지만, 진정한 다항 시간 알고리즘으로 인정받으려면 실행 시간이 *입력 크기*, 즉 입력을 표현하는 데 필요한 *비트* 수에 대한 다항 함수여야 합니다. X의 원소들과 대상 합 T의 *값*은 입력 비트 수보다 지수적으로 클 수 있습니다. 실제로 방금 설명한 환원은 원래 입력 그래프의 크기보다 지수적으로 큰 T 값을 만들어 내며, 이 때문에 우리의 동적 계획법 알고리즘은 지수 시간에 실행될 수밖에 없습니다.

이와 같은 알고리즘은 ***의사 다항 시간(pseudo-polynomial time)***에 실행된다고 하며, 이러한 알고리즘이 존재하는 NP-난해 문제를 ***약한 NP-난해(weakly NP-hard)*** 문제라고 합니다. 달리 말하면, 약한 NP-난해 문제란 모든 입력 수를 *단항*(1들의 합)으로 표현하면 다항 시간에 풀 수 있지만, 모든 입력 수를 *이진*으로 표현하면 NP-난해가 되는 문제입니다. 모든 입력 수를 단항으로 표현하더라도 NP-난해인 문제는 ***강한 NP-난해(strongly NP-hard)*** 문제라고 합니다. *강한* NP-난해 문제의 좋은 예로는 TravelingSalesman이 있으며, 이 문제는 입력 그래프가 완전 그래프이고 모든 간선의 가중치가 1 또는 2인 경우에도 여전히 NP-난해입니다.

12.13 기타 유용한 NP-난해 문제

말 그대로 수천 개의 문제가 NP-난해임이 증명되었습니다. 여기서는 환원을 유도하는 데 유용한 NP-난해 문제 몇 가지를 나열하겠습니다.[11] 이 문제들의 NP-난해성 증명을

[11] 언젠가 환원을 찾아야 할 날이 올지도 모르니, 제게는 작은 목록이 있습니다. 제게는 작은 목록이 있습니다. 하한을 증명하는 데 쓸 수 있는 헤라클레스급 문제들의 목록입니다. 빠른 해법은 존재하지 않습니다. 이 문제들은 우리의 증명을 도와줄 수 있습니다.

자세히 설명하지는 않겠지만, 대부분은 Garey와 Johnson의 고전인 *NP-완전성의 무시무시한 검은 책*에서 찾아볼 수 있습니다.[12] 지금까지 다룬 모든 문제와 다음 목록에 있는 문제 대부분은 1972년 Richard Karp가 발표한 기념비적인 논문 한 편에서 처음으로 NP-난해임이 증명되었습니다.[13]

- **PlanarCircuitSat**: 두 전선이 교차하지 않도록 평면에 매장할 수 있는 불 회로가 주어졌을 때, 회로가 True를 출력하게 하는 입력이 존재합니까? 이 문제는 각 교차점을 작은 게이트 조합으로 대체함으로써 일반 회로 충족 가능성 문제로부터의 환원을 통해 NP-난해임을 증명할 수 있습니다.

- **1-in-3Sat**: 3CNF 논리식이 주어졌을 때, 각 절이 True인 리터럴을 *정확히* 하나씩 포함하도록 변수에 값을 할당하는 방법이 존재합니까? 이 문제는 일반적인 3Sat으로부터의 환원을 통해 NP-난해임을 증명할 수 있습니다.

- **NotAllEqual3Sat**: 3CNF 논리식이 주어졌을 때, 모든 절이 적어도 하나의 True 리터럴*과* 적어도 하나의 False 리터럴을 포함하도록 변수에 값을 할당하는 방법이 존재합니까? 이 문제는 일반적인 3Sat으로부터의 환원을 통해 NP-난해임을 증명할 수 있습니다.

- **Planar3Sat**: 3CNF 불 논리식이 주어졌을 때, 절과 변수를 정점으로 하고 어떤 변수(또는 그 부정)가 어떤 절에 나타나면 그 둘을 간선으로 잇는 이분 그래프를 생각해 봅시다. 이 그래프가 평면 그래프이면 해당 3CNF 논리식도 평면이라고 부릅니다. Planar3Sat 문제는 평면 3CNF 논리식이 주어졌을 때 그 논리식에 만족하는 할당이 있는지 묻습니다. 이 문제는 PlanarCircuitSat으로부터의 환원을 통해 NP-난해임을 증명할 수 있습니다.[14]

- **Exact3DimensionalMatching** 또는 **X3M**: 집합 S와 *삼중항*이라고 부르는 S의 세 원소 부분집합들의 모음이 주어졌을 때, S를 정확히 덮는 서로소인 삼중항들의 부분 모음이 존재합니까? 이 문제에는 3이 들어 있으므로 3Sat으로부터의 환원을 통해 NP-난해임을 증명할 수 있습니다.

- **Partition**: n개의 정수로 이루어진 집합 S가 주어졌을 때, $A \cup B = S$, $A \cap B = \varnothing$ 이면서 다음 조건을 만족하는 부분집합 A와 B가 존재합니까?

$$\sum_{a \in A} a = \sum_{b \in B} b?$$

  이 문제는 SubsetSum으로부터의 간단한 환원을 통해 NP-난해임을 증명할 수 있습니다. SubsetSum과 마찬가지로 Partition 문제도 약한 의미에서만 NP-난해입니다.

- **3Partition**: $3n$개의 정수로 이루어진 집합 S가 주어졌을 때, 모든 부분집합의 합이 정확히 같도록 S를 서로소인 원소 세 개짜리 부분집합 n개로 분할할 수 있

[12] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., 1979. 그리고 네, 정말로 검은색입니다.

[13] 이 논문은 훗날 오프브로드웨이에서 *리처드 카프와 21명의 조수*들이라는 제목으로 공연되었고, Karp는 이 작품으로 마땅히 받을 만한 토니 튜링상을 수상했습니다.

[14] 놀랍게도 PlanarNotAllEqual3Sat은 다항 시간에 풀 수 있습니다!

습니까? 이름은 비슷하지만 이 문제는 Partition과 *매우* 다릅니다. 죄송하지만 이름을 지은 사람은 제가 아닙니다. 이 문제는 X3M으로부터의 환원을 통해 NP-난해임을 증명할 수 있는데, 이름에 3이 들어 있기 때문입니다. Partition과 달리 3Partition 문제는 *강한* NP-난해 문제입니다. 즉, 모든 입력 수가 n^3 이하이더라도 여전히 NP-난해입니다.

- **SetCover**: 집합들의 모음 $\mathbf{S} = \{S_1, S_2, \ldots, S_m\}$이 주어졌을 때, $\bigcup_i S_i$의 모든 원소를 포함하는 S_i들의 가장 작은 부분 모음을 찾으십시오. 이 문제는 VertexCover와 X3M을 모두 일반화한 것입니다.

- **HittingSet**: 집합들의 모음 $\mathbf{S} = \{S_1, S_2, \ldots, S_m\}$이 주어졌을 때, $\mathbf{S}$에 속한 모든 집합과 만나는 $\bigcup_i S_i$의 원소의 최소 개수를 구하십시오. 이 문제 역시 VertexCover의 일반화입니다.

- **LongestPath**: 음이 아닌 가중치를 갖는 가중 그래프 G(방향 그래프 또는 무방향 그래프)와 두 정점 u, v가 주어질 때, 그래프에서 u에서 v로 가는 가장 긴 단순 경로는 무엇입니까? 경로가 각 정점을 최대 한 번만 방문하면 그 경로를 *단순* 경로라고 합니다. 이 문제는 대응하는 해밀턴 경로 문제를 일반화한 것입니다. 물론 대응하는 *최단* 경로 문제는 다항 시간에 풀 수 있습니다.

- **SteinerTree**: 일부 정점이 표시된 가중치 무방향 그래프 G가 주어질 때, 표시된 모든 정점을 포함하는 G의 최소 가중치 부분 트리는 무엇입니까? *모든* 정점이 표시되어 있다면 최소 슈타이너 트리는 최소 신장 트리이며, 정확히 두 정점만 표시되어 있다면 최소 슈타이너 트리는 그 두 정점 사이의 최단 경로입니다. 이 문제는 VertexCover로부터의 환원을 통해 NP-난해임을 증명할 수 있습니다.

- **Max2Sat**: 각 절에 리터럴이 정확히 두 개씩 있는 논리곱 정규형 불 논리식이 주어졌을 때, True 리터럴을 하나 이상 포함하는 절의 개수를 최대화하는 변수 할당을 찾으십시오. 이 문제는 3Sat으로부터의 환원으로 NP-난해임을 증명할 수 있습니다 (그렇습니다, 이름에 3이 들어 있지 않은데도 말입니다). *모든* 절을 만족하는 할당이 존재하는지 묻는 더 단순한 결정 문제 2Sat은 실제로 다항 시간에 풀 수 있습니다.

- **MaxCut**: 무방향 그래프 $G = (V, E)$가 주어졌을 때, 끝점 중 정확히 하나만 S에 속하는 간선의 개수를 최대화하는 부분집합 $S \subset V$를 찾으십시오. 이는 G의 가장 큰 이분 부분 그래프를 찾는 것과 동치입니다. 이 문제는 Max2Sat으로부터의 환원을 통해 NP-난해임을 증명할 수 있습니다.

이처럼 무미건조하지만 유용한 문제들 외에도, 흥미로운 퍼즐과 1인용 게임 대부분이 NP-난해이거나 NP-난해한 일반화를 가진다는 사실이 증명되었습니다. (어떤 게임이나 퍼즐이 적어도 NP-난해하지 않다면 그것은 흥미롭지 않다고 주장할 수도 있습니다!) 다음은 여러분에게 익숙할 만한 몇 가지 예입니다:

- 지뢰 찾기(CircuitSat로부터 환원)[15]

[15] Richard Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer* 22(2):9–15, 2000. 다만 다음 문헌도 참고하십시오: Allan Scott, Ulrike Stege, and Iris van Rooij. Minesweeper may not be NP-complete but is hard nonetheless. *Mathematical Intelligencer* 33(4):5–17, 2011.

- 스도쿠(궁극적으로는 3Sat에서 환원)[16]
- 테트리스 (3Partition으로부터)[17]
- 클론다이크, 일명 "1인용 카드 게임(Solitaire)" (3Sat에서)[18]
- 팩맨(HamiltonianCycle로부터 환원)[19]
- 슈퍼 마리오 브라더스(3Sat로부터 환원)[20]
- 캔디 크러시 사가 (3Sat의 한 변형으로부터)[21]
- Threes/2048 (물론 3Sat로부터)[22]
- Trainyard(DominatingSet로부터; 연습문제 26 참조)[23]
- 최단 $n \times n \times n$ 루빅스 큐브 해법(3Sat에서 PlanarUndirectedHamCycle의 특수한 경우를 거쳐 환원).[24]
- 쿠키 클리커(Partition 또는 3Partition에서 환원)[25]

각주 예산이 한정되어 있는 탓에 이 목록은 불완전할 수밖에 없습니다.[26] 2019년 6월 현재, Ultimate Paperclips, Line Rider, Twister, Cards Against Humanity를 일반화한 게임이 NP-난해임을 보인 증명은 아직 아무도 발표하지 않았지만, 이는 시간문제일 뿐이라고 확신합니다.

---

[16]Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A(5):1052–1060, 2003. http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf.

[17]Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik J. Hoogeboom, Walter A. Kosters, and David Liben-Nowell. Tetris is hard, even to approximate. *International Journal of Computational Geometry and Applications* 14:41–68, 2004.

[18]Luc Longpré and Pierre McKenzie. The complexity of Solitaire. *Proceedings of the 32nd International Mathematical Foundations of Computer Science*, 182–193, 2007.

[19]Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014. http://giovanniviglietta.com/papers/gaming2.pdf.

[20]Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games Are (computationally) hard. *Theoretical Computer Science* 586:135–160, 2015. http://arxiv.org/abs/1203.1895.

[21]Luciano Gualà, Stefano Leucci, Emanuele Natale. Bejeweled, Candy Crush and other match-three games are (NP-)hard. *Proc. 2014 IEEE Conference on Computational Intelligence and Games*, 2014. http://arxiv.org/abs/1403.5830.

[22]Stefan Langerman and Yushi Uno. Threes!, Fives, 1024!, and 2048 are Hard. *제8회 Fun with Algorithms(알고리즘의 즐거움) 국제 학회 논문집*, 2016. https://arxiv.org/abs/1505.04274.

[23]Matteo Almanza, Stefano Leucci, Alessandro Panconesi. Trainyard는 NP-난해입니다. *제8회 Fun with Algorithms(알고리즘의 재미) 국제 학술대회 논문집*, 2016. https://arxiv.org/abs/1603.00928.

[24]Erik D. Demaine, Sarah Eisenstat, and Mikhail Rudoy. Solving the Rubik's Cube optimally is NP-complete. *Proc. 35th Symposium on Theoretical Aspects of Computer Science*, 2018. https://arxiv.org/abs/1706.06708.

[25]Erik D. Demaine, Hiro Ito, Stefan Langerman, Jayson Lynch, Mikhail Rudoy, and Kai Xiao. Cookie Clicker. 프리프린트, 2018년 8월. https://arxiv.org/abs/1808.07540.

[26]https://xkcd.com/1208/ 을 참고하십시오.

12.14 올바른 문제 선택하기

어떤 문제가 NP-난해임을 증명할 때 가장 어려운 단계 중 하나는 환원의 출발점으로 삼을 가장 좋은 문제를 고르는 것입니다. 쿡-레빈 정리에 따르면 *어떤* NP-난해 문제에서든 문제 X로의 환원이 존재하면 모든 NP-완전 문제에서 문제 X로의 환원이 존재합니다. 하지만 어떤 문제는 다른 문제보다 다루기가 더 쉽습니다. 올바른 문제를 고르는 체계적인 방법은 없지만, 몇 가지 유용한 경험칙을 소개합니다.

- 문제가 객체에 비트를 할당하는 방법을 묻거나, 객체의 부분집합을 선택하도록 요구하거나, 객체를 서로 다른 두 부분집합으로 분할하도록 요구한다면, Sat이나 Partition의 어떤 변형으로부터 환원해 보십시오.
- 작은 고정 집합에서 뽑은 레이블을 객체에 배정하는 방법을 묻거나 객체를 적은 수의 부분집합으로 분할하는 방법을 묻는 문제라면, *k*Color나 3Color로부터의 환원을 시도해 보십시오.
- 객체들의 집합을 특정한 순서로 배열하라고 요구하는 문제라면 DirectedHamCycle이나 DirectedHamPath, 또는 TravelingSalesman으로부터 환원해 보십시오.
- 문제에서 어떤 제약 조건을 만족하는 *작은* 부분집합을 찾으라고 요구한다면, MinVertexCover로부터 환원해 보십시오.
- 문제가 어떤 제약 조건을 만족하는 *큰* 부분집합을 찾으라고 요구한다면, MaxIndSet이나 MaxClique 또는 Max2Sat으로부터 환원해 보십시오.
- 문제에서 객체들을 다수의 작은 부분집합으로 분할하라고 요구한다면, 3Partition으로부터 환원해 보십시오.
- 문제에 숫자 3이 자연스럽게 등장한다면 3Sat이나 3Color, X3M, 3Partition을 시도해 보십시오. (아니요, 농담이 아닙니다.)
- 다른 방법이 모두 실패하면 3Sat이나 심지어 CircuitSat까지 시도해 보십시오!

Tetris, SuperMarioBros, Trainyard에서 환원을 시도하는 것은 권하지 않습니다. 출발 문제로는 가능한 한 단순하면서도, 여러분의 문제를 풀기 어렵게 만드는 특징을 *하나라도* 담고 있는 문제를 고르는 것이 좋습니다.

12.15 사소한 실생활 예시

드래프트는 수천 년 동안 즐겨 온 보드 게임의 한 갈래입니다. 대부분의 미국인은 *체커* 또는 *잉글리시 드래프트*라고 불리는 변형에 익숙하지만, 전 세계적으로 가장 널리 퍼진 변형인 ***국제 드래프트*** 또는 ***폴란드 드래프트***는 16세기 네덜란드에서 유래했습니다. 전체 규칙은 위키백과를 참고하십시오. 여기서는 영미식 게임과의 중요한 차이점 몇 가지를 소개합니다:

- **플라잉 킹:** 체커와 마찬가지로, 상대편에 가장 가까운 줄에서 이동을 마친 말은 *킹*이 되어 뒤로 움직일 수 있게 됩니다. 그러나 체커와 달리, 국제 드래프트의 킹은

중간 칸이 비어 있거나 상대 말이 정확히 하나만 있는 한(그 말은 잡힙니다), 한 번의 차례에 대각선을 따라 원하는 거리만큼 이동할 수 있습니다.

- **최대 포획 강제:** 매 턴마다 수를 두는 플레이어는 상대편 말을 가능한 한 많이 잡아야 합니다. 이는 체커의 포획 강제 규칙과는 다릅니다. 체커의 규칙은 잡을 수 있으면 반드시 잡아야 한다는 것, 그리고 움직이는 말이 더 이상 잡을 수 없을 때에만 포획 수가 끝난다는 것만을 요구합니다. 다시 말해, 체커는 매 턴 상대편 말의 *국소적으로 극대인* 집합을 잡도록 요구하는 반면, 국제 드래프트는 *전역적으로 최대인* 포획을 요구합니다.
- **잡기의 세부 규칙:** 체커에서와 마찬가지로, 잡힌 말은 턴이 끝날 때에야 판에서 제거됩니다. 어떤 말이든 최대 한 번만 잡힐 수 있습니다. 따라서 상대 말을 뛰어넘으면 그 말은 판에 남아 있지만, 턴이 끝날 때까지 *다시 뛰어넘을 수는 없습니다.*

예를 들어, 아래에 보인 첫 번째 국면에서 각 원은 말을 나타내고, 이중 원은 킹을 나타냅니다. 흑은 표시된 첫 번째 수를 *반드시* 두어 백의 말 다섯 개를 잡아야 합니다. 다섯 개보다 많은 말을 잡는 것은 불가능하고, 다섯 개를 잡는 다른 수도 없기 때문입니다. 흑은 북동쪽으로도 남동쪽으로도 잡기를 더 이어 갈 수 없습니다. 잡힌 백의 말들은 흑의 차례가 끝날 때까지 판 위에 남아 있기 때문입니다. 그러면 백은 표시된 두 번째 수를 *반드시* 두어야 하며, 그 수로 게임에서 승리합니다.

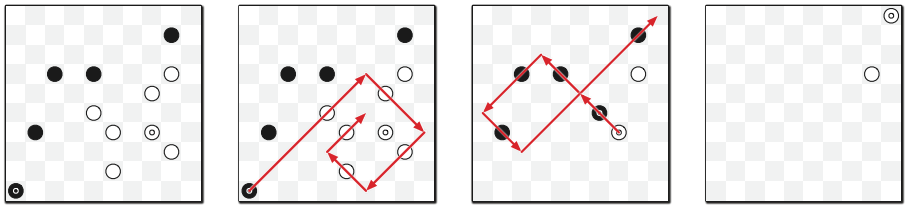


그림 12.21. 국제 드래프트에서 강제된(!) 두 수입니다. 이중 원은 킹입니다.

각 색깔의 말 20개씩을 가지고 10×10 판 위에서 진행되는 실제 게임은 계산적으로 자명합니다. 가능한 모든 판 배치에 대해 두 플레이어의 최적 수를 미리 계산해 두고, 그 결과를 상수 크기의 조회 테이블에 하드코딩할 수 있기 때문입니다. 물론 *큰* 상수이긴 하지만, 그래도 여전히 상수입니다!

하지만 국제 드래프트를 $n \times n$ 보드로 자연스럽게 일반화한 경우를 생각해 봅시다. 이 설정에서는 ***합법적인 수를 찾는 것조차 실제로 NP-난해입니다!*** 방향 그래프에서의 해밀턴 사이클 문제로부터의 다음 환원은 2010년에 밥 헌(Bob Hearn)이 발견했습니다.[27] 대부분의 2인용 게임에서 *최선의* 수를 찾는 것은 NP-난해(또는 그보다 더 어려운 문제)입니다. 제가 아는 한, *단지 규칙을 따르는 것*만으로도 NP-난해인 게임은 이것이 유일합니다. 게다가 이 게임은 수 세기 동안 수백만 명이 즐겨 온 실제 게임입니다!

n개의 정점을 가진 무방향 그래프 G가 주어지면, G에 해밀턴 사이클이 있을 때에만 그리고 그때에만 백이 한 번의 수로 특정 개수의 흑 말을 잡을 수 있도록 하는 국제 드래프트 판 배치를 구성합니다. G를 방향 그래프로 취급하여, 각 무방향 간선 uv 대신

[27] Theoretical Computer Science Stack Exchange를 참고하십시오: http://cstheory.stackexchange.com/a/1999/111.

두 호 $u \to v$와 $v \to u$를 둡니다. 정점에 1부터 n까지 임의로 번호를 매기십시오. 최종 드래프트 배치는 여러 가젯으로 이루어집니다.

- G의 정점들은 토끼 모양의 *정점 가젯*으로 나타내며, 이 가젯들은 수평선을 따라 같은 간격으로 배치됩니다. 각 호 $i \to j$는 정점 가젯 i의 "왼쪽 귀"에서 정점 가젯 j의 "오른쪽 귀"까지 이어지는 경로로 나타내며, 이 경로는 두 개의 대각선 선분으로 이루어집니다. 호 $i \to j$에 대한 경로는 $i < j$이면 정점 가젯들의 위쪽에, $i > j$이면 정점 가젯들의 아래쪽에 위치합니다.

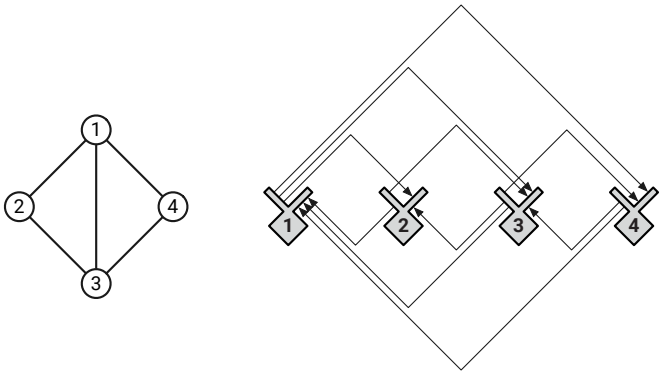


그림 12.22. 해밀턴 사이클에서 국제 드래프트로의 환원에 대한 개략적인 개요입니다.

- 각 정점 가젯의 대부분은 *금고*라고 부르는 다이아몬드 모양의 영역입니다. 금고의 벽은 잡을 수 없는 흑색 말 두 겹의 단단한 층으로 이루어져 있으며, 그림에서 이 말들은 회색 원으로 그려져 있습니다. 각 금고 안에는 잡을 수 있는 흑색 말이 N개 있는데, 여기서 N은 나중에 정할 어떤 큰 정수입니다. 백색 킹은 "오른쪽 귀"를 통해 금고에 들어가 내부의 말을 모두 잡은 다음 "왼쪽 귀"를 통해 나올 수 있습니다. 두 귀는 모두 복도이며, 이 복도의 벽도 마찬가지로 말 두 개 두께입니다. 호 경로가 끝나는 곳에는 틈이 있어서 백색 킹이 그곳으로 들어오고 나갈 수 있습니다. "귀"의 길이는 다른 가젯과 맞도록 쉽게 조정할 수 있습니다.
- 각 호 $i \to j$마다 *코너 가젯*을 하나씩 둡니다. 이 가젯은 정점 가젯 i를 떠나는 백색 킹이 정점 가젯 j 쪽으로 방향을 바꾸도록 해 줍니다.
- 마지막으로, 두 호 경로가 교차하는 곳마다 *교차 가젯*을 둡니다. 이 가젯은 백색 킹이 두 호 경로 중 어느 쪽이든 지나갈 수 있게 하지만, 한 호 경로에서 다른 호 경로로 갈아타는 것은 금지합니다.

백색 킹 하나가 금고 중 하나의 아래쪽 모서리에서 출발합니다. 모든 합법적인 수에서 이 킹은 호 경로 전체를 지나가는 것과 금고를 비우는 것을 번갈아 해야 합니다. 킹은 여러 가젯을 거꾸로 지나갈 수도 있습니다. 즉, 각 금고에 출구로 들어가고 입구로 나올 수 있습니다. 하지만 G의 해밀턴 사이클을 뒤집은 것 역시 G의 해밀턴 사이클이므로 거꾸로 걸어가도 문제가 없습니다.

G에 해밀턴 사이클이 있으면, 백색 킹은 다른 금고를 각각 방문한 뒤 시작 금고로 돌아옴으로써 최소 nN개의 흑색 말을 잡을 수 있습니다. 반면 G에 해밀턴 사이클이

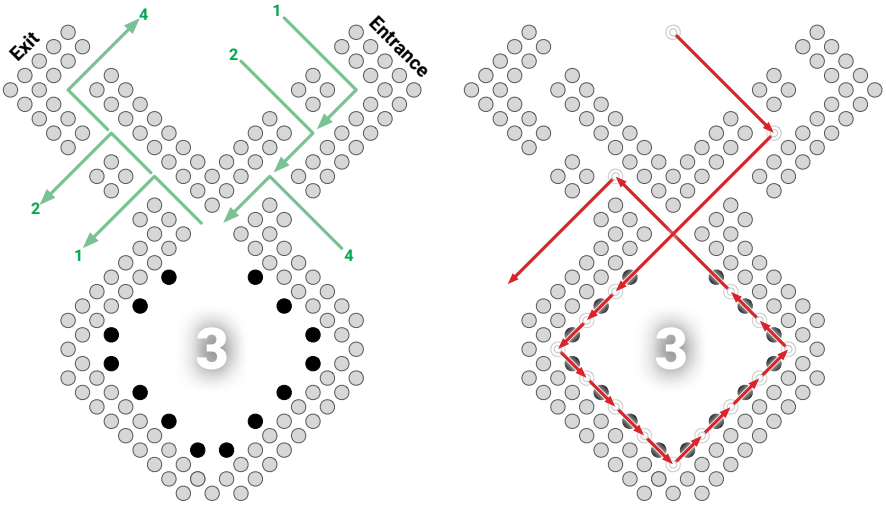


그림 12.23. 왼쪽: 입구 세 개와 출구 세 개가 있는 정점 가젯입니다. 오른쪽: 금고를 비우는 백색 킹입니다. 회색 원은 잡을 수 없는 흑색 기물입니다.

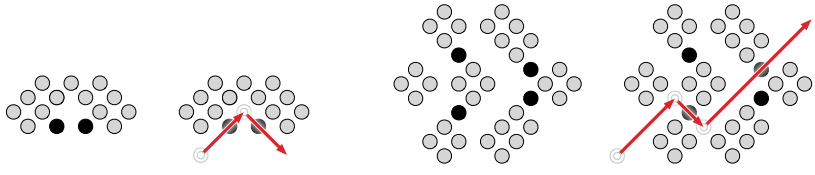


그림 12.24. 왼쪽: 모퉁이 가젯을 지나는 두 경로 중 하나입니다. 오른쪽: 교차 가젯을 지나는 두 경로 중 하나입니다.

없으면, 백색 킹은 시작 금고에 있는 말 중 최대 절반만 잡을 수 있으므로 전체적으로 최대 $(n-1/2)N + O(n^3)$개의 적 말을 잡을 수 있습니다. $O(n^3)$ 항은 모서리 가젯과 교차 가젯을 반영한 것입니다. 각 간선은 하나의 모서리 가젯과 최대 $n^2/2$개의 교차 가젯을 지나갑니다.

환원을 완성하기 위해 $N = n^4$로 설정합니다. 종합하면, $O(n^5)$개의 흑색 기물과 하나의 백색 킹이 놓인 $O(n^5) \times O(n^5)$ 크기의 판 배치를 얻습니다. 이 판 배치는 무차별 대입으로 다항 시간 안에 구성할 수 있음이 분명합니다. 그림 12.25는 이 구성의 완전한 예를 보여 줍니다.

다음의 관련 문제가 NP-난해인지는 아직 밝혀지지 않았습니다. 국제 드래프트의 $n \times n$ 판 배치가 주어졌을 때, 백은 한 차례에 흑의 말을 모두 잡아 게임에서 승리할 수 있습니까(그리고 그렇다면 반드시 그렇게 *해야만* 합니까)?

12.16 ♥얼룩말 너머로

P와 NP는 복잡도 클래스로 이루어진 거대한 계층 구조의 첫 두 단계에 불과합니다. 이 장(그리고 이 책)을 마무리하며, 흥미로운 클래스 몇 가지를 더 소개하겠습니다.

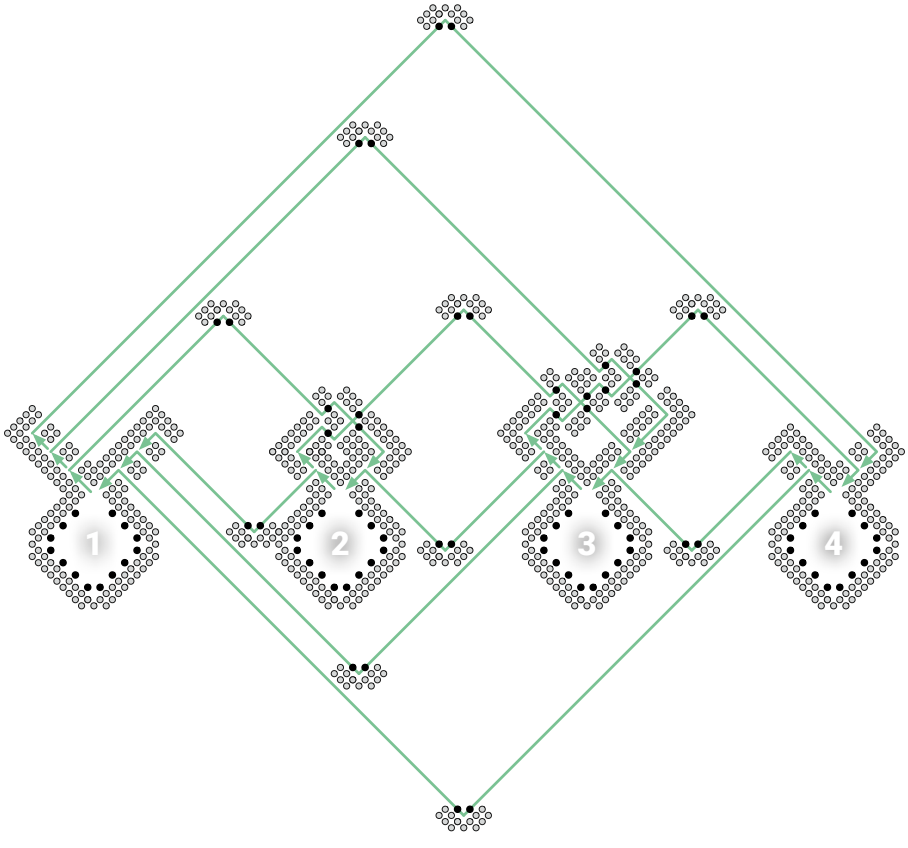


그림 12.25. 그림 12.22의 정점이 4개인 그래프에 대한 최종 체커 배치입니다. (초록색 화살표는 실제로는 배치에 포함되지 않습니다.)

다항 공간

PSPACE는 다항 *공간*을 사용하여 풀 수 있는 결정 문제들의 집합입니다. NP에 속하는 모든 문제(따라서 P에 속하는 모든 문제)는 PSPACE에도 속합니다. 일반적으로 NP ≠ PSPACE라고 여겨지지만, P ≠ PSPACE조차 증명한 사람은 아무도 없습니다.

다항식 *공간*을 사용하여 풀 수 있는 모든 문제 Π'에 대해 Π'에서 Π로 가는 다항식 *시간* 다대일 환원이 존재하면, 문제 Π를 ***PSPACE-난해***라고 합니다. 어떤 PSPACE-난해 문제가 NP에 속하면 PSPACE=NP이고, 마찬가지로 어떤 PSPACE-난해 문제가 P에 속하면 PSPACE=P입니다.

대표적인 PSPACE-난해 문제는 *양화 불 논리식*(quantified boolean formula) 문제, 즉 ***QBF***입니다. 자유 변수는 없지만 전칭 양화사나 존재 양화사를 몇 개든 포함할 수 있는 불 논리식 Φ가 주어졌을 때, Φ는 True와 동치입니까? 예를 들어, 다음 식은 QBF의 올바른 입력입니다:

$$\exists a : \forall b : \exists c : (\forall d : a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee (\exists e : \overline{(\bar{a} \Rightarrow e)} \vee (c \neq a \wedge e))).$$

Sat은 입력 논리식이 존재 한정기호(∃)만을 포함하는 QBF의 특수한 경우와 동치입

니다. QBF는 입력 논리식의 모든 한정기호가 맨 앞에 와야 하고, 한정기호가 ∃와 ∀ 사이에서 엄격하게 번갈아 나타나며, 한정된 명제가 각 절마다 정확히 세 개의 리터럴을 갖는 논리곱 정규형이어야 하는 경우에도 여전히 PSPACE-난해합니다. 예를 들면 다음과 같습니다.

$$\exists a : \forall b : \exists c : \forall d : \big((a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})\big)$$

이 제한된 버전의 QBF는 2인 전략 문제로도 표현할 수 있습니다. 두 플레이어 앨리스와 밥에게 자유 변수 $x_1, x_2, \ldots, x_n$을 갖는 3CNF 술어가 주어졌다고 가정합시다. 두 플레이어는 번갈아 가며 인덱스 순서대로 변수에 값을 할당합니다—앨리스가 x_1에 값을 할당하고, 밥이 x_2에 값을 할당하는 식입니다. 결국 앨리스는 인덱스가 홀수인 모든 변수에 값을 할당하고, 밥은 인덱스가 짝수인 모든 변수에 값을 할당합니다. 앨리스는 식을 True로 만들고자 하고, 밥은 False로 만들고자 합니다. 앨리스와 밥이 완벽하게 플레이한다고 가정할 때, 이 게임에서 누가 이깁니까? 놀랍지 않게도, 틱택토, 리버시, 체커, 바둑, 체스, 만칼라와 같은 대부분의 2인 게임[28]은—더 정확히 말하면, 이러한 상수 크기 게임을 임의의 보드 크기로 적절히 일반화한 것은—PSPACE-난해입니다.

또 다른 대표적인 PSPACE-난해 문제는 *NFA 전체성*입니다. 어떤 알파벳 Σ 위의 비결정적 유한 상태 오토마톤 M이 주어졌을 때, M은 Σ^*의 모든 문자열을 받아들입니까? 이와 밀접하게 관련된 문제인 *NFA 동치성*(주어진 두 NFA가 같은 언어를 받아들입니까?)과 *NFA 최소화*(주어진 NFA와 같은 언어를 받아들이는 가장 작은 NFA를 찾으십시오) 역시 PSPACE-난해이며, 정규 표현식에 관한 대응 문제들도 마찬가지입니다. (*결정적* 유한 상태 오토마톤에 관한 대응 문제들은 사실 다항 시간에 풀 수 있습니다.)

지수 시간

그다음으로 상당히 더 큰 복잡도 클래스인 ***EXP***(EXPTIME이라고도 합니다)는 지수 시간에, 즉 어떤 상수 $c > 0$에 대해 최대 2^{n^c} 단계를 사용하여 풀 수 있는 결정 문제들의 집합입니다. PSPACE에 속하는 모든 문제(따라서 NP에 속하는 모든 문제(따라서 P에 속하는 모든 문제))는 EXP에도 속합니다. 일반적으로 PSPACE ⊊ EXP라고 믿어지고 있지만, NP ≠ EXP조차 아무도 증명하지 못하고 있습니다. *지수* 시간에 풀 수 있는 임의의 문제 Π′에 대해 Π′에서 Π로 가는 *다항* 시간 다대일 환원이 존재하면, 문제 Π는 ***EXP-난해***하다고 합니다. 어떤 EXP-난해 문제가 PSPACE에 속한다면 EXP=PSPACE이며, 마찬가지로 어떤 EXP-난해 문제가 NP에 속한다면 EXP=NP입니다. P ≠ EXP라는 사실은 우리가 *확실히* 알고 있습니다. 특히 어떤 EXP-난해 문제도 P에 속하지 않습니다.

체커, 체스, 만칼라, 바둑처럼 흥미로운 2인용 게임 중 상당수는 자연스럽게 일반화하면 실제로 EXP-난해입니다. PSPACE-완전 게임과 EXP-난해 게임 사이의 경계는 상당히 미묘합니다. 예를 들어 체스(표준 8 × 8 게임)에서 무승부가 되는 경우는 세 가지입니다. 스테일메이트(둘 차례인 플레이어가 체크 상태는 아니지만 둘 수 있는 합법적인 수가 없는 경우), 같은 판 배치가 세 번 반복되는 경우, 그리고 기물을 잡거나

[28] 게임과 퍼즐의 계산 복잡도에 관해 알려진 결과를 잘 개관한 (그러나 불가피하게 낡아 가는) 자료로는 Erik Demaine과 Bob Hearn의 단행본 *Games, Puzzles, and Computation* (CRC Press, 2009)을 참고하십시오.

폰을 움직이지 않고 50수를 두는 경우입니다. 체스를 $n \times n$으로 일반화한 게임은 이 규칙들을 어떻게 일반화하느냐에 따라 PSPACE에 속하기도 하고 EXP-난해가 되기도 합니다. 만약 (예를 들어) 기물을 잡지 않는 수가 n^3번 이어진 뒤에 무승부를 선언한다면, 모든 게임은 다항식 개수의 수를 둔 뒤에는 반드시 끝나야 하므로, 주어진 어떤 배치에서든 가능한 모든 게임을 다항식 공간만으로 시뮬레이션할 수 있습니다. 반면에 기물을 잡지 않는 수에 관한 규칙을 완전히 무시하면, 그 결과로 나오는 게임은 지수 개수의 수만큼 이어질 수 있으므로, 다항식 공간만으로 반복되는 배치를 검출할 뚜렷한 방법이 없습니다. 실제로 이 버전의 $n \times n$ 체스는 EXP-난해입니다.

더 높이!

당연히 지수 시간조차도 이야기의 끝이 아닙니다. ***NEXP***는 *비결정적* 지수 시간에 풀 수 있는 결정 문제들의 클래스입니다. 동등하게 말하면, 결정 문제가 NEXP에 속할 필요충분조건은 모든 YES 인스턴스에 대해 그 사실을 지수 시간에 검증할 수 있는 *증명*이 존재하는 것입니다. ***EXPSPACE***는 지수 크기의 *공간*을 사용하여 풀 수 있는 결정 문제들의 집합입니다. 이처럼 더 큰 복잡도 클래스에도 어려운 문제가 있습니다. 예를 들어 정규 표현식의 문법에 교집합 연산자 $\cap$을 추가하면, 그러한 두 표현식이 같은 언어를 기술하는지 판정하는 문제는 EXPSPACE-난해입니다. EXPSPACE 너머에는 *이중* 지수 자원 한계를 갖는 복잡도 클래스(EEXP, NEEXP, EEXPSPACE)가 있고, 그다음에는 *삼중* 지수 자원 한계를 갖는 클래스(EEEXP, NEEEXP, EEEXPSPACE)가 있으며, 이런 식으로 끝없이 이어집니다.

이 복잡도 클래스들은 모두 포함 관계에 따라 순서를 매길 수 있습니다:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP} \subseteq \mathrm{NEXP} \subseteq \mathrm{EXPSPACE} \subseteq \mathrm{EEXP} \subseteq \mathrm{NEEXP} \subseteq \cdots$$

대부분의 복잡도 이론가들은 이 수열의 모든 포함 관계가 진포함이라고, 즉 이 복잡도 클래스들 중 어느 두 개도 서로 같지 않다고 굳게 믿습니다. 그러나 지금까지 증명된 가장 강력한 결과는 이 수열의 모든 클래스가 수열에서 *세* 단계 뒤에 있는 클래스에 진부분집합으로 포함된다는 것입니다. 예를 들어 $\mathrm{P} \neq \mathrm{EXP}$와 $\mathrm{PSPACE} \neq \mathrm{EXPSPACE}$는 증명되어 있지만, $\mathrm{P} \neq \mathrm{PSPACE}$인지 또는 $\mathrm{NP} \neq \mathrm{EXP}$인지는 증명되지 않았습니다.

점점 더 지수적으로 커지는 이 복잡도 클래스 계열의 극한은 어떤 상수 정수 k에 대해 $2 \uparrow^k n$ 꼴의 함수로 제한되는 시간 또는 공간을 사용하여 풀 수 있는 결정 문제들의 클래스 ***ELEMENTARY***입니다. 여기서

$$2 \uparrow^k n := \begin{cases} n & \text{만약} k = 0, \\ 2^{2 \uparrow^{k-1} n} & \text{그 외의 경우.} \end{cases}$$

예를 들어, $2 \uparrow^1 n = 2^n$이고 $2 \uparrow^2 n = 2^{2^n}$입니다.

모든 자연스러운 결정 가능 문제를 초등 시간 안에 풀 수 있다고 추측하고 싶을 수도 있지만, 실제로 이 추측은 틀렸습니다. 어떤 유한 알파벳 위의 (비어 있을 수도 있는) 문자열들을 연접(xy), 합집합($x + y$), 클레이니 폐포(x^*), 부정($\overline{x}$)으로 재귀적으로 결합하여 정의되는 ***일반화된 정규 표현식***을 생각해 봅시다. 예를 들어 일반화된 정규 표현식 $\overline{(0 + 1)^*00(0 + 1)^*}$는 $\{0, 1\}^*$에 속하는 문자열 중 0이 두 번 연속으로 나타나지

않는 문자열의 집합을 나타냅니다. 각 표현식을 동치인 NFA로 재귀적으로 변환하고, 각 NFA를 DFA로 변환한 다음, 그 DFA를 최소화하면 두 일반화된 정규 표현식이 같은 언어를 기술하는지를 알고리즘으로 판정할 수 있습니다. 그러나 이 알고리즘의 실행 시간은 비초등적인 한계 $2 \uparrow^{\Theta(n)} 2$를 가집니다. 직관적으로 말하면, 재귀적 부정의 각 층이 상태의 수를 지수적으로 늘릴 수 있기 때문입니다. 실제로 래리 스톡마이어(Larry Stockmeyer)는 1974년에 클레이니 페포를 금지하더라도 이 문제를 단지 초등 시간 안에는 풀 수 *없다*는 것을 증명했습니다.

연습문제

1. (a) 입력 집합의 크기를 n, 그 원소들의 절댓값의 합을 M이라고 할 때, PARTITION을 $O(nM)$ 시간에 해결하는 알고리즘을 설명하고 분석하십시오.

   (b) 이 알고리즘으로부터 P=NP가 따라 나오지 않는 이유는 무엇입니까?

2. BOXDEPTH라고 부르는 다음 문제를 생각해 봅시다. 평면 위에 축에 평행한 직사각형 n개의 집합이 주어질 때, 공통된 한 점을 포함하는 직사각형들의 부분집합 중 가장 큰 것의 크기는 얼마입니까?

   (a) BOXDEPTH에서 MAXCLIQUE로의 다항 시간 환원을 설명하십시오.

   (b) BOXDEPTH를 해결하는 다항 시간 알고리즘을 설명하고 분석하십시오. *[힌트: $O(n^3)$ 시간 알고리즘은 쉽게 찾을 수 있겠지만, $O(n \log n)$ 시간도 가능합니다.]*

   (c) 이 두 결과로부터 왜 P=NP가 따라 나오지 않습니까?

3. 불 논리식이 여러 개의 *항*(terms)의 *논리합*(OR)으로 이루어져 있고 각 항이 하나 이상의 리터럴의 논리곱(AND)이면, 그 불 논리식은 *논리합 정규형*(또는 *DNF*)이라고 합니다. 예를 들어, 다음 논리식은

$$(\overline{x} \wedge y \wedge \overline{z}) \vee (y \wedge z) \vee (x \wedge \overline{y} \wedge \overline{z})$$

   논리합 정규형입니다. DNF-SAT은 논리합 정규형인 불 논리식이 주어졌을 때 그 논리식이 충족 가능한지를 묻습니다.

   (a) DNF-SAT를 해결하는 다항 시간 알고리즘을 설명하십시오.

   (b) P=NP라는 다음 논증의 오류는 무엇입니까?

> 절마다 리터럴이 최대 세 개인 논리곱 정규형의 불 논리식이 주어졌고, 이 논리식이 충족 가능한지 알고 싶다고 가정합시다. 분배 법칙을 사용하면 논리합 정규형으로 된 동치인 논리식을 구성할 수 있습니다. 예를 들어,
>
> $$(x \vee y \vee \overline{z}) \wedge (\overline{x} \vee \overline{y}) \iff (x \wedge \overline{y}) \vee (y \wedge \overline{x}) \vee (\overline{z} \wedge \overline{x}) \vee (\overline{z} \wedge \overline{y})$$
>
> 이제 (a)번 문항의 알고리즘을 사용하여 그 결과로 얻은 DNF 논리식이 충족 가능한지를 다항 시간에 판정할 수 있습니다. 우리는 방금 3SAT을 다항 시간에 풀었습니다. 3SAT은 NP-난해이므로, P=NP라고 결론지을 수밖에 없습니다!

4. AllOrNothing3Sat 문제는 3CNF 불 논리식이 주어졌을 때, 모든 절이 세 개의 True 리터럴을 갖거나 세 개의 False 리터럴을 갖도록 하는 변수 할당이 존재하는지를 묻습니다.

   (a) AllOrNothing3Sat를 해결하는 다항 시간 알고리즘을 설명하십시오.

   (b) 하지만 3Sat은 NP-난해합니다! 그렇다면 왜 이 알고리즘이 존재한다는 사실로 P=NP임이 증명되지 않습니까?

5. (a) 임의의 가중 그래프 G가 주어지면 G에서 가장 짧은 해밀턴 사이클의 길이를 **다항 시간에** 판별할 수 있는 마법의 블랙박스가 주어졌다고 가정합시다. 이 마법의 블랙박스를 서브루틴으로 사용하여, 임의의 가중 그래프 G가 주어지면 G에서 가장 짧은 해밀턴 사이클을 계산하는 **다항 시간** 알고리즘을 설명하고 분석하십시오.

   (b) 임의의 그래프 G가 주어졌을 때 G의 가장 큰 완전 부분 그래프에 속한 정점의 수를 **다항 시간에** 알아낼 수 있는 마법의 블랙박스가 주어졌다고 가정합시다. 이 마법의 블랙박스를 서브루틴으로 사용하여, 임의의 그래프 G가 주어졌을 때 G의 최대 크기 완전 부분 그래프를 계산하는 **다항 시간** 알고리즘을 설명하고 분석하십시오.

   (c) 임의의 그래프 G가 주어졌을 때 G가 3-색칠 가능한지를 **다항 시간에** 판별할 수 있는 마법의 블랙박스가 주어졌다고 가정합시다. 이 마법의 블랙박스를 서브루틴으로 사용하여, 주어진 그래프의 올바른 3-색칠을 계산하거나 그러한 색칠이 존재하지 않음을 정확히 보고하는 **다항 시간** 알고리즘을 설명하고 분석하십시오. *[힌트: 마법의 블랙박스에 들어가는 입력은 그래프입니다. 오직 그래프뿐입니다. 정점과 간선뿐입니다. 그 밖에는 아무것도 없습니다.]*

   (d) 임의의 불 논리식 Φ가 주어지면 Φ가 충족 가능한지를 **다항 시간에** 판정할 수 있는 마법의 블랙박스가 주어졌다고 가정하십시오. 이 마법의 블랙박스를 서브루틴으로 사용하여, 주어진 불 논리식에 대해 만족하는 할당을 계산하거나 그러한 할당이 존재하지 않음을 올바르게 보고하는 **다항 시간** 알고리즘을 설계하고 분석하십시오.

   (e) 양의 정수로 이루어진 임의의 집합 X가 주어졌을 때, X를 $\sum A = \sum B$를 만족하는 두 집합 A와 B로 분할할 수 있는지를 **다항 시간 안에** 판별하는 마법의 블랙박스가 있다고 가정합시다. 이 마법의 블랙박스를 서브루틴으로 사용하여, 주어진 양의 정수 집합의 균등 분할을 계산하거나 그러한 분할이 존재하지 않는다고 올바르게 보고하는 **다항 시간** 알고리즘을 기술하고 분석하십시오.

   ♣♥(f) 임의의 일반화된 정규 표현식 R(연습문제 바로 앞에서 정의한 것)이 주어졌을 때, R과 일치하는 문자열이 하나라도 존재하는지를 **다항 시간 안에** 판정할 수 있는 마법의 블랙박스가 주어졌다고 가정합시다. 이 마법의 블랙박스를 서브루틴으로 사용하여, 주어진 일반화된 정규 표현식과 일치하는 문자열 하나를 찾거나 그러한 문자열이 존재하지 않음을 올바르게 보고하는 **다항 시간** 알고리즘을 기술하고 분석하십시오.

6. *숫자 3에는 특별한 무언가가 있습니다.*

   (a) 2Partition을 위한 다항 시간 알고리즘을 설명하고 분석하십시오. $2n$개의 양의 정수로 이루어진 집합 S가 주어졌을 때, 여러분의 알고리즘은 S의 원소들을 합이 모두 같은 서로소인 n개의 쌍으로 나눌 수 있는지를 다항 시간 안에 판별해야 합니다.

   (b) 2Color를 위한 다항 시간 알고리즘을 기술하고 분석하십시오. 무방향 그래프 G가 주어졌을 때, 여러분의 알고리즘은 G에 두 가지 색만 사용하는 올바른 색칠이 존재하는지를 다항 시간 안에 판별해야 합니다.

   (c) 2Sat를 위한 다항 시간 알고리즘을 설명하고 분석하십시오. 각 절에 정확히 *두 개의* 리터럴이 있는 논리곱 정규형의 불 논리식 Φ가 주어지면, 여러분의 알고리즘은 Φ에 만족하는 할당이 있는지를 다항 시간 안에 판정해야 합니다. *[힌트: 이 문제는 앞 장에서 설명한 주제들과 강하게 연결되어 있습니다.]*

7. *숫자 3에는 특별한 의미가 없습니다.*

   (a) 문제 12Partition은 다음과 같이 정의됩니다. 양의 정수 $12n$개로 이루어진 집합 S가 주어졌을 때, S의 원소들을 각각 12개의 원소를 가지며 합이 모두 같은 n개의 부분집합으로 나눌 수 있는지 판정하십시오. 12Partition이 NP-난해임을 증명하십시오. *[힌트: 3Partition으로부터 환원하십시오. 먼저 다중집합을 고려하는 편이 더 쉬울 수 있습니다.]*

   (b) 12Color 문제는 다음과 같이 정의됩니다. 무방향 그래프 G가 주어졌을 때, 모든 간선의 양 끝 정점이 서로 다른 색을 갖도록 각 정점을 열두 가지 색 중 하나로 칠할 수 있는지 판정하십시오. 12Color가 NP-난해임을 증명하십시오. *[힌트: 3Color로부터 환원하십시오.]*

   (c) 12SAT 문제는 다음과 같이 정의됩니다. 절마다 정확히 열두 개의 리터럴이 있는 논리곱 정규형의 불 논리식 Φ가 주어졌을 때, Φ에 만족하는 할당이 있는지 판정하십시오. 12Sat이 NP-난해임을 증명하십시오. *[힌트: 3Sat에서 환원하십시오.]*

8. 해밀턴 사이클 문제에는 방향 그래프에 대한 것과 무방향 그래프에 대한 것, 두 가지 버전이 있습니다. 이 장의 앞부분에서 *방향* 해밀턴 사이클 문제가 NP-난해라는 두 가지 증명을 찾을 수 있습니다.

   (a) *무방향* 해밀턴 사이클 문제에서 *방향* 해밀턴 사이클 문제로의 다항 시간 환원을 설명하십시오. 여러분의 환원이 올바름을 증명하십시오.

   (b) *방향* 해밀턴 사이클 문제를 *무방향* 해밀턴 사이클 문제로 바꾸는 다항 시간 환원을 설명하십시오. 그 환원이 올바름을 증명하십시오.

   (c) 이 두 환원 중 어느 것이 *무방향* 해밀턴 사이클 문제가 NP-난해임을 함의합니까?

9. (a) UndirectedHamiltonianCycle에서 DirectedHamiltonianCycle로의 다항 시간 환원을 설명하십시오.

   (b) DirectedHamiltonianCycle에서 UndirectedHamiltonianCycle로의 다항 시간 환원을 설명하십시오.

10. (a) HamiltonianPath 문제에서 HamiltonianCycle 문제로의 다항 시간 환원을 기술하십시오.

    (b) HamiltonianCycle 문제에서 HamiltonianPath로의 다항 시간 환원을 설명하십시오. *[힌트: 다항 시간 환원은 블랙박스 서브루틴을 두 번 이상 호출할 수 있지만, 반드시 그래야 하는 것은 아닙니다.]*

11. CNFSat의 다음과 같은 미묘한 변형들을 생각해 봅시다. 각 문제에서 입력은 논리곱 정규형으로 된 불 논리식 Φ이며, 목표는 Φ를 만족하는 할당이 존재하는지 판정하는 것입니다.

    (a) Φ의 모든 절이 리터럴을 ***최대*** 세 개 포함하고, 각 변수가 최대 ***세*** 개의 절에 나타난다고 가정합시다. CNFSat의 이 변형이 NP-난해임을 증명하십시오.

    (b) Φ의 모든 절이 ***정확히*** 세 개의 리터럴을 포함하고, 각 변수가 최대 ***네*** 개의 절에 나타난다고 가정합시다. 3Sat의 이 변형이 NP-난해임을 증명하십시오. *[힌트: (a)번 문제를 먼저 푸십시오.]*

    ♥(c) Φ의 각 절이 *임의 개수의* 리터럴을 포함할 수 있지만, 각 변수는 최대 **두** 개의 절에만 나타난다고 가정합시다. CNFSat의 이 변형 문제를 푸는 다항 시간 알고리즘을 설명하십시오.

    ♥(d) Φ의 모든 절이 ***정확히*** 세 개의 리터럴을 포함하고, 각 변수가 최대 ***세*** 개의 절에 나타난다고 가정합시다. Φ가 반드시 충족 가능함을 증명하십시오. (따라서 3Sat의 이 변형은 완전히 자명합니다!)

12. (a) NotAllEqual3Sat가 NP-난해임을 증명하십시오.

    (b) 1-in-3Sat가 NP-난해임을 증명하십시오.

13. *배타적 논리합 논리곱 정규형*(XCNF)의 불 논리식은 여러 *절*의 논리곱(And)이며, 각 절은 여러 리터럴의 *배타적* 논리합입니다. 즉, 절은 참인 리터럴을 홀수 개 포함할 때, 그리고 그때에만 참입니다. XCNF-Sat 문제는 주어진 XCNF 논리식이 충족 가능한지를 묻습니다. XCNF-Sat에 대한 다항 시간 알고리즘을 설명하거나, XCNF-Sat이 NP-난해임을 증명하십시오. *[힌트: 둘 다 하려고 하지 마십시오.]*

♥14. Majority3Sat이라고 하는 3Sat의 다음 변형을 생각해 봅시다. 3Sat과 마찬가지로 Majority3Sat의 입력은 절마다 리터럴이 정확히 세 개인 논리곱 정규형의 불 논리식 Φ입니다. Majority3Sat은 모든 절이 *적어도 두 개의* True 리터럴을 포함하도록 하는 Φ의 변수에 대한 할당이 존재하는지를 묻습니다.

Majority3Sat을 다항 시간에 푸는 알고리즘을 기술하거나, Majority3Sat이 NP-난해임을 증명하십시오. *[힌트: 둘 다 하려고 하지 마십시오.]*

♠♥15. 임의의 부분집합 $X \subseteq \{0, 1, 2, 3\}$에 대하여, 제가 ***X-3Sat***이라고 부를 다음 문제를 생각해 봅시다. 입력은 각 절에 정확히 세 개의 리터럴이 있는 논리곱 정규형의 불 논리식 Φ입니다. 이 문제는 Φ의 각 절에서 True인 리터럴의 개수가 집합 X에 속하도록 하는 Φ의 변수에 대한 할당이 존재하는지를 결정하는 것입니다. 예를 들면 다음과 같습니다:

- $\{1, 2, 3\}$-3Sat는 표준 3Sat 문제입니다.
- $\{0, 3\}$-3Sat은 AllOrNothing3Sat과 같습니다. (연습문제 4를 참고하십시오.)
- $\{1, 2\}$-3Sat은 보통 NotAllEqual3Sat이라고 부릅니다. (연습문제 12(a)를 참고하십시오.)
- $\{1\}$-3Sat은 보통 1-in-3Sat이라고 부릅니다. (연습문제 12(b)를 참고하십시오.)
- $\{1, 3\}$-3Sat은 보통 XCNF-3Sat이라고 부릅니다. (연습 문제 13을 참조하십시오.)
- $\{2, 3\}$-3Sat은 보통 Majority3Sat이라고 부릅니다. (연습문제 14를 참고하십시오.)

P $\neq$ NP라고 가정할 때, X-3Sat 문제를 다항 시간에 풀 수 있는 부분집합 $X \subseteq \{0, 1, 2, 3\}$를 빠짐없이 모두 나열하십시오. *[힌트: 16가지 경우마다 서로 다른 논증을 제시하지는 마십시오.]*

16. 다음 문제들이 NP-난해임을 증명하십시오.

(a) 무방향 그래프 G가 주어졌을 때, G에 정확히 17개를 제외한 모든 정점을 방문하는 단순 경로가 존재합니까?

(b) 무방향 그래프 G가 주어졌을 때, G에 모든 노드의 차수가 23 이하인 신장 트리가 존재합니까?

(c) 무방향 그래프 G가 주어졌을 때, G에 잎이 최대 42개인 신장 트리가 존재합니까?

(d) 무방향 그래프 $G = (V, E)$가 주어졌을 때, E의 간선 중 양 끝점이 모두 S에 속하는 간선이 최대 374개가 되도록 하는 정점 부분집합 $S \subseteq V$ 중 가장 큰 것의 크기는 얼마입니까?

(e) 무방향 그래프 $G = (V, E)$가 주어졌을 때, S에 속한 각 정점의 S 안의 이웃이 최대 473개가 되도록 하는 정점 부분집합 $S \subseteq V$ 중 가장 큰 것의 크기는 얼마입니까?

(f) 무방향 그래프 G가 주어졌을 때, 두 끝점의 색이 같은 간선이 최대 31337개가 되도록 G의 정점을 서로 다른 세 가지 색으로 칠할 수 있습니까?

17. 다음에 제시하는 최소 신장 트리 문제의 변형들이 NP-난해임을 증명하십시오.

(a) 그래프 G가 주어졌을 때, G의 *지름이 최대인* 신장 트리를 계산하십시오. (트리 T의 지름은 T에서 최장 경로의 길이입니다.)

(b) 간선에 가중치가 있는 그래프 G가 주어졌을 때, G의 최소 가중치 *깊이 우선* 신장 트리를 계산하십시오.

(c) 가중치가 있는 간선을 가진 그래프 G와 G의 정점들의 부분집합 S가 주어졌을 때, 모든 리프가 S에 속하는 최소 가중치 신장 트리를 계산하십시오.

(d) 간선에 가중치가 있는 그래프 G와 정수 ℓ이 주어졌을 때, 리프가 최대 ℓ개인 최소 가중치 신장 트리를 계산하십시오.

(e) 가중치가 있는 간선을 가진 그래프 G와 정수 Δ가 주어졌을 때, 모든 노드의 차수가 최대 Δ인 최소 가중치 신장 트리를 계산하십시오.

18. (a) 그림 12.26(a)의 가젯을 사용하여, 주어진 *평면* 그래프가 3-색칠 가능한지 판정하는 문제가 NP-난해임을 증명하십시오. *[힌트: 먼저 가젯이 3-색칠 가능함을 보이고, 그다음 평면 임베딩에 있는 모든 교차를 가젯으로 적절히 대체하십시오.]*

(b) (a)와 그림 12.26(b)의 가젯을 사용하여, *최대 차수가 4인* 평면 그래프가 3-색칠 가능한지 판정하는 문제가 NP-난해임을 증명하십시오. *[힌트: 차수가 4보다 큰 정점을 모두 가젯들의 모음으로 대체하되, 어떤 정점의 차수도 4를 넘지 않도록 가젯들을 연결하십시오.]*

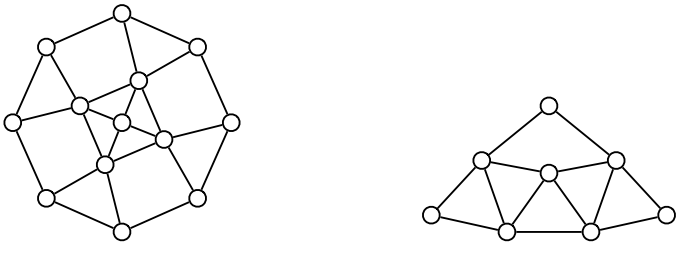


그림 12.26. (a) 평면 3-색칠 가능성을 위한 가젯. (b) 차수 4 평면 3-색칠 가능성을 위한 가젯.

19. PlanarCircuitSat가 NP-난해임을 증명하십시오. *[힌트: 교차하는 전선을 위한 가젯을 구성하십시오.]*

20. (a) 3Sat에서 4Sat으로의 다항 시간 환원을 설명하십시오.

(b) 4Sat에서 3Sat으로의 다항 시간 환원을 설명하십시오.

♥21. 4Color에서 3Color로 가는 직접적인 다항 시간 환원을 서술하십시오. (이는 반대 방향보다 훨씬 어렵습니다.)

22. *도미노*는 두 개의 정사각형으로 나뉜 1×2 직사각형이며, 각 정사각형에는 정수가 하나씩 적혀 있습니다.[29] 도미노의 *올바른 배열*에서는 맞닿은 끝의 숫자가 서로 일치하도록 도미노들이 끝과 끝을 이어 한 줄로 늘어섭니다.

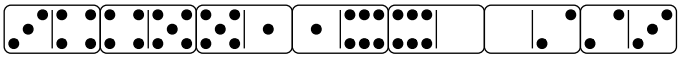


그림 12.27. 0부터 6까지의 모든 정수가 각각 두 번씩 나타나는 올바른 도미노 배치

다음 각 문제에 대해 다항 시간 알고리즘을 설명하거나, 해당 문제가 NP-난해임을 증명하십시오:

(a) 임의의 도미노 모음 D가 주어졌을 때, D에 있는 *모든* 도미노를 규칙에 맞게 배열하는 방법이 존재합니까?

(b) 도미노들의 임의의 주머니 D가 주어졌을 때, 1부터 n까지의 모든 정수가 정확히 두 번씩 나타나도록 D의 도미노들을 올바르게 배열하는 방법이 존재합니까?

♥(c) 임의의 도미노 주머니 D가 주어졌을 때, D에서 도미노를 꺼내 올바른 배치를 만들 때 사용할 수 있는 도미노의 최대 개수는 얼마입니까?

23. *조약돌 게임(pebbling)*은 무방향 그래프 G 위에서 하는 1인용 게임이며, 각 정점에는 0개 이상의 *조약돌*이 놓여 있습니다. 한 번의 *조약돌 이동*은 정점 v에서 조약돌 두 개를 제거하고 v의 임의의 이웃 정점에 조약돌 하나를 추가하는 것입니다. (당연히 정점 v에는 이동하기 전에 조약돌이 적어도 두 개 있어야 합니다.) PebbleDestruction 문제는 그래프 $G = (V, E)$와 각 정점 v의 조약돌 개수 $p(v)$가 주어졌을 때, 조약돌을 하나만 남기고 모두 제거하는 일련의 조약돌 이동이 존재하는지를 묻습니다. PebbleDestruction이 NP-난해임을 증명하십시오.

24. 그래프 G의 5-색칠이란 G의 각 정점에 집합 $\{0, 1, 2, 3, 4\}$에서 "색"을 하나씩 배정하는 함수로서, 모든 간선 uv에 대해 정점 u와 v에 서로 다른 "색"이 배정되도록 하는 함수임을 상기하십시오. 인접한 정점들에 배정된 색이 서로 다를 뿐만 아니라 $1 \pmod 5$보다 크게 차이 나는 경우, 그 5-색칠을 *신중한* 색칠이라고 합니다. 주어진 그래프에 신중한 5-색칠이 존재하는지 판정하는 문제가 NP-난해임을 증명하십시오. *[힌트: 표준 5Color 문제로부터 환원하십시오.]*

25. (a) 무방향 그래프 G의 정점 부분집합 S에 대해, S의 각 정점이 S의 다른 정점 *최대 하나*와만 인접하면 S를 *반독립*이라고 합니다. 주어진 무방향 그래프에서 가장 큰 반독립 집합의 크기를 구하는 문제가 NP-난해임을 증명하십시오.

[29] 이 정수들은 보통 주사위와 똑같이 점으로 표시됩니다. 표준 도미노에서 각 면의 점 개수는 0에서 6 사이이지만, 각 면에 점이 최대 9개, 심지어 12개까지 있는 세트도 살 수 있습니다. 여기서는 임의의 정수 레이블을 허용하겠습니다. 표준 도미노 세트에는 가능한 레이블의 비순서쌍마다 도미노가 정확히 하나씩 들어 있지만, 우리는 문제의 입력이 이러한 성질을 가진다고 가정하지 *않습니다*.

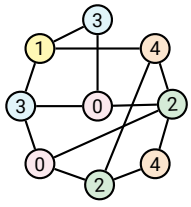


그림 12.28. 세심하게 칠한 5-색칠입니다.

(b) 무방향 그래프 G의 정점 부분집합 S는 S에 속한 각 정점이 S의 다른 정점 중 *최대 374개*와 인접할 때 *대충-독립*이라고 합니다. 주어진 무방향 그래프에서 정점들로 이루어진 가장 큰 대충-독립 집합의 크기를 구하는 문제가 NP-난해임을 증명하십시오.

(c) 무방향 그래프 G의 정점 부분집합 S에 대해, G에서 두 끝점이 모두 S에 속하는 간선이 최대 374개이면 S를 *거의 독립*이라고 합니다. 주어진 무방향 그래프에서 가장 큰 거의 독립 집합의 크기를 구하는 문제가 NP-난해임을 증명하십시오.

26. $G = (V, E)$를 그래프라고 합시다. G의 *지배 집합*이란 정점들의 부분집합 S로서, G의 모든 정점이 S에 속하거나 S에 속한 어떤 정점과 인접한 것을 말합니다. DominatingSet 문제는 그래프 G와 정수 k가 입력으로 주어졌을 때, G가 크기 k인 지배 집합을 포함하는지를 묻습니다. 이 문제가 NP-난해임을 증명하십시오.

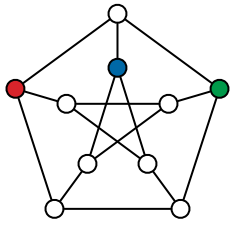


그림 12.29. 페테르센 그래프에서 크기가 3인 지배 집합입니다.

27. 무방향 그래프 G의 정점 부분집합 S에 속한 모든 정점 삼중쌍 $u, v, w \in S$에 대해 세 간선 uv, uw, vw 중 적어도 하나가 G에 *존재하지 않으면*, S를 *삼각형이 없는* (triangle-free) 부분집합이라고 합니다. 주어진 무방향 그래프에서 삼각형이 없는 가장 큰 정점 부분집합의 크기를 구하는 문제가 NP-난해임을 증명하십시오.

28. RectangleTiling 문제는 다음과 같이 정의됩니다. 큰 직사각형 하나와 그보다 작은 직사각형 여러 개가 주어질 때, 작은 직사각형들을 큰 직사각형 안에 빈틈이나 겹침 없이 배치할 수 있는지 판정하십시오.

(a) RectangleTiling이 NP-난해임을 증명하십시오.

(b) RectangleTiling이 *강한* NP-난해임을 증명하십시오.

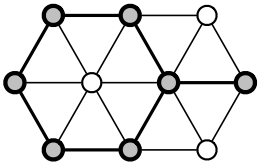


그림 12.30. 정점 7개로 이루어진 삼각형 없는 부분집합입니다. 이 그래프에서 이런 부분집합 중 가장 큰 것은 아닙니다.

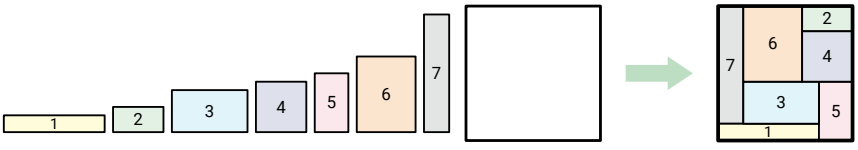


그림 12.31. RectangleTiling 문제의 양성 인스턴스입니다.

29. (a) 그래프 G의 정점 부분집합 B에 대해, G에서 B에 속한 모든 정점을 제거했을 때 남는 부분 그래프에 해밀턴 경로가 없으면 B를 ***Burr 집합***이라고 합니다. 주어진 그래프에서 가장 작은 Burr 집합을 찾는 문제가 NP-난해임을 증명하십시오.

    (b) 그래프 G의 정점 부분집합 S에 대하여, G에서 S의 모든 정점을 제거하고 남은 부분 그래프가 해밀턴 경로를 *포함한다면* S를 ***스카일러 집합***이라고 합니다. 주어진 그래프에서 가장 작은 스카일러 집합을 찾는 문제가 NP-난해임을 증명하십시오.

30. (a) 그래프 G의 *토니안 경로*(tonian path)란 G의 정점 중 적어도 절반을 지나는 경로입니다. 주어진 그래프에 토니안 경로가 존재하는지 판별하는 문제가 NP-난해임을 보이십시오.

    (b) 그래프 G에서 *토니안 사이클*이란 G의 정점 중 적어도 절반을 지나는 사이클입니다. 그래프에 토니안 사이클이 있는지 판정하는 문제가 NP-난해임을 보이십시오. *[힌트: (a)번을 이용하십시오. 이용하지 않아도 됩니다.]*

31. G를 간선에 가중치가 있는 무방향 그래프라고 합시다. *무거운 해밀턴 사이클*이란 G의 각 정점을 정확히 한 번씩 지나는 사이클 C 가운데, C에 속한 간선들의 가중치 합이 G의 모든 간선의 가중치 합의 절반보다 큰 사이클을 말합니다. 그래프에 무거운 해밀턴 사이클이 있는지 판정하는 문제가 NP-난해임을 증명하십시오.

32. 다음 각 문제에 대해 다항 시간 알고리즘을 설명하거나, 그 문제가 NP-난해임을 증명하십시오.

    (a) 무방향 그래프 G에서 *이중 오일러 투어*란 G의 모든 간선을 정확히 두 번씩 지나는 닫힌 보행입니다. 그래프 G가 주어졌을 때, G에 이중 오일러 투어가 존재합니까?

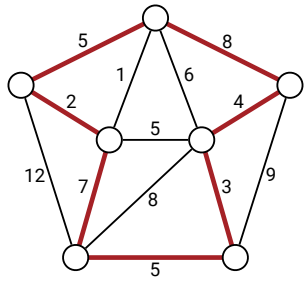


그림 12.32. 무거운 해밀턴 사이클. 이 사이클의 총 가중치는 34이고, 그래프의 총 가중치는 67입니다.

(b) 무방향 그래프 G에서 *이중 해밀턴 순회*란 G의 모든 정점을 정확히 두 번씩 방문하는 닫힌 보행입니다. 그래프 G가 주어졌을 때, G에 이중 해밀턴 순회가 존재합니까?

(c) 무방향 그래프 G에서 *이중 해밀턴* ***회로***란 G의 모든 정점을 정확히 두 번씩 방문하면서 *G의 각 간선은 최대 한 번만 지나는* 닫힌 보행입니다. 그래프 G가 주어졌을 때, G에 이중 해밀턴 회로가 존재합니까?

(d) 무방향 그래프 G에서의 *삼중 오일러 투어*란 G의 모든 간선을 정확히 세 번씩 지나는 닫힌 보행입니다. 그래프 G가 주어졌을 때, G에 삼중 오일러 투어가 존재합니까?

(e) 무방향 그래프 G에서 *삼중 해밀턴 순회*란 G의 모든 정점을 정확히 세 번씩 방문하는 닫힌 보행입니다. 그래프 G가 주어졌을 때, G에 삼중 해밀턴 순회가 존재합니까?

33. 이 연습 문제에서는 VertexCover에서 SteinerTree로의 특정 환원이 올바르다는 것을 증명해야 합니다. 주어진 무방향 그래프 $G = (V, E)$에서 가장 작은 정점 덮개를 찾으려 한다고 가정합시다. 새로운 그래프 $H = (V', E')$를 다음과 같이 구성합니다.

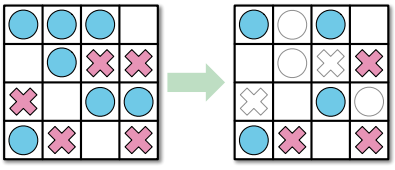
- $V' = V \cup E \cup \{z\}$
- $E' = \{ve \mid v \in V$가 끝점인 $e \in E\} \cup \{vz \mid v \in V\}$입니다.

이와 동등하게, G의 각 간선에 새로운 정점을 하나씩 넣어 세분한 다음, G의 원래 정점을 모두 새로운 *꼭대기(apex)* 정점 z에 연결하여 H를 구성할 수도 있습니다.

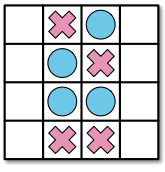
G에 크기가 k인 정점 덮개가 존재하는 것은 H의 부분 트리 중 정점이 $k+|E|+1$개이고 $E \cup \{z\}$의 모든 정점을 포함하는 것이 존재하는 것과 동치임을 증명하십시오.

34. 다음과 같은 1인용 솔리테어 게임을 생각해 봅시다. 이 퍼즐은 $n \times m$ 크기의 정사각형 칸으로 이루어진 격자로 구성되며, 각 칸은 비어 있거나, 빨간 돌이 놓여 있거나, 파란 돌이 놓여 있을 수 있습니다. 퍼즐의 목표는 주어진 돌 중 일부를 제거하여 남은 돌이 다음 두 조건을 만족하게 하는 것입니다: (1) 모든 행에 돌이

적어도 하나 있어야 하고, (2) 어떤 열에도 두 색의 돌이 함께 있어서는 안 됩니다. 돌의 초기 배치에 따라서는 이 목표를 달성하는 것이 불가능한 경우도 있습니다.



풀 수 있는 퍼즐과 그 퍼즐의 여러 해 중 하나입니다.



풀 수 없는 퍼즐입니다.

빨간 돌과 파란 돌의 초기 배치가 주어졌을 때 퍼즐을 풀 수 있는지 판정하는 문제가 NP-난해임을 증명하십시오.

35. 다음 각 게임에는 정사각형 칸으로 이루어진 $n \times m$ 격자가 있으며, 각 칸은 비어 있거나 돌이 하나 놓여 있습니다. 한 번의 수로 임의의 열 하나에 있는 돌을 모두 제거할 수 있습니다.

(a) 격자의 각 행에 돌이 *최대* 하나만 남도록 비울 수 있는 열들의 *가장 작은* 부분집합을 찾는 문제가 NP-난해임을 증명하십시오.

(b) 격자의 각 행에 돌이 *적어도* 하나씩 남도록 비울 수 있는 열들의 부분집합 중 *가장 큰* 것을 찾는 문제가 NP-난해임을 증명하십시오.

♥(c) 열의 부분집합 중 *어느* 하나라도 비워서 격자의 각 행에 돌이 *정확히* 하나씩 남게 할 수 있는지 판정하는 문제가 NP-난해임을 증명하십시오.

36. 제프는 학생들을 행복하게 해 주려고 노력합니다. 강의 첫 시간에 그는 자신이 융통성을 발휘할 수 있는 영역에서 가능한 여러 강의 방침을 나열한 설문지를 나누어 줍니다. 모든 학생은 가능한 각 강의 방침에 대해 "적극 찬성", "대체로 중립", "적극 반대" 중 하나로 응답하도록 요청받습니다. 각 학생은 최대 다섯 개의 질문에만 "적극 찬성" 또는 "적극 반대"로 응답할 수 있습니다. 제프의 학생들은 매우 너그럽기 때문에, 각 학생은 자신의 강한 방침 선호 중 적어도 하나가 관철되면 (그리고 오직 그 경우에만) 행복합니다. 행복한 학생의 수를 최대화하도록 강의 방침을 정하는 다항 시간 알고리즘을 기술하거나, 이 문제가 NP-난해임을 보이십시오.

37. 여러분은 지역 커뮤니티 극장에서 올릴 뮤지컬의 안무를 맡았고, 이제 공연 마지막을 장식할 화려한 대형 넘버의 최종 포즈를 정할 차례입니다. ("Streetcar!") 여러분은 노래가 끝날 때 공연에 출연하는 n명의 출연진이 모두 길게 한 줄로 늘어서서 팔을 쭉 뻗고 최고의 스피릿 핑거를 뽐내도록 하기로 결정했습니다.

감독은 마지막 피날레 동안 모든 출연진이 양팔을 모두 위로 들거나 모두 아래로 내려야 한다고 선언했습니다. 누가 위를 가리키고 누가 아래를 가리킬지 정하는 것이 여러분의 임무입니다. 게다가 감독은 자신의 섬세한 예술적 기질을 거스를 배치들의 목록도 여러분에게 건넸습니다. 각각의 금지된 배치는 출연진의

부분집합에 팔 위치를 짝지은 것입니다. 예를 들면 다음과 같습니다. "네드, 아푸, 스미더스가 팔을 아래로 내리고 있는 동안 마지는 팔을 위로 들 수 없습니다."

허용 가능한 팔 위치 배치를 찾는 문제가 NP-난해임을 증명하십시오.

38. 다음번에 여러분이 파티에 가면, 손님 중 한 명이 다 같이 삼자 멈블티펙(Three-Way Mumbletypeg)을 한 판 하자고 제안할 것입니다. 삼자 멈블티펙은 세 팀과 칼 한 자루가 필요한, 기술과 손재주를 겨루는 게임입니다. 삼자 멈블티펙 공식 규칙(1625년 신성 로마 삼자 멈블티펙 공의회에서 확정됨)에 따르면 (1) 각 팀에는 *반드시* 한 명 이상이 있어야 하고, (2) 같은 팀에 속한 두 사람은 누구든 *반드시* 서로 아는 사이여야 하며, (3) 게임을 지켜보는 모든 사람은 *반드시* 세 팀 중 한 팀에 속해야 합니다. 물론 정말 *즐거운* 파티일 것이므로 아무도 자리를 뜨고 싶어 하지 않을 것입니다. 파티에는 서로 모르는 사이인 사람 쌍이 여럿 있을 것입니다. 파티 주최자는 알고리즘에 관한 모든 일에서 여러분이 발휘한 뛰어난 솜씨에 대한 흥미진진한 이야기를 들은 터라, 칼을 갈면서 파티 참석자 중 어느 쌍이 서로 아는 사이인지 적힌 목록을 여러분에게 건네고 팀을 정해 달라고 부탁할 것입니다.

파티 참석자들을 규칙에 맞는 삼자 멈블티펙(Three-Way Mumbletypeg) 팀 세 개로 나눌 수 있는지 판정하는 다항 시간 알고리즘을 기술하고 분석하거나, 이 문제가 NP-난해임을 증명하십시오.

39. 여러분이 참석한 파티는 아주 즐겁게 진행되고 있지만, 이제 *알고리즘 행진(アルゴリズムこうしん)*을 위해 줄을 설 시간입니다! 이 춤은 원래 일본의 코미디 콤비 이츠모 코코카라(いつもここから)가 어린이 텔레비전 프로그램 피타고라 스위치(ピタゴラスイッチ)를 위해 만든 것입니다. 알고리즘 행진은 한 줄로 늘어선 사람들이 추며, 줄에 선 각 사람은 바로 앞사람보다 한 마디 늦게 정해진 일련의 동작을 시작합니다. 따라서 이 행진은 "Row Row Row Your Boat"나 "Frère Jacques" 같은 음악의 돌림노래 또는 카논을 춤으로 옮긴 것이라고 할 수 있습니다.

올바른 예절에 따르면, 사소한 실수 때문에 서로 모르는 사람들 사이에 끔찍하게 민망한 일이 생기지 않도록 각 행진 참가자는 줄에서 바로 앞에 선 사람을 알고 있어야 합니다. 파티에 온 사람들 가운데 누가 서로 아는 사이인지를 모두 담은 목록이 주어진다고 가정합시다. 알고리즘 행진에 참가할 수 있는 파티 참석자의 최대 인원을 구하는 문제가 NP-난해임을 증명하십시오. 일반성을 잃지 않고 파티에 닌자는 없다고 가정해도 됩니다.

♣40. 비결정적 유한 상태 오토마타와 정규 표현식에 관한 다음 문제들이 NP-난해임을 증명하십시오:

(a) 알파벳 $\Sigma = \{0, 1\}$ 위의 NFA M이 주어졌을 때, M이 받아들이지 *않는* 문자열이 Σ^*에 존재합니까?

(b) 알파벳 $\Sigma = \{0, 1\}$ 위의 *비순환* NFA M이 주어졌을 때, M이 받아들이지 *않는* Σ^*의 문자열 중 *가장 짧은* 문자열의 길이는 얼마입니까?

(c) 알파벳 $\Sigma = \{0, 1\}$ 위의 정규 표현식 R이 주어졌을 때, R이 매치하지 *않는* 문자열이 Σ^*에 존재합니까?

(d) 알파벳 $\Sigma = \{0, 1\}$ 위의 *스타 없는(star-free)* 정규 표현식 R이 주어졌을 때, R이 매칭하지 *않는* Σ^*의 문자열 중 *가장 짧은* 문자열의 길이는 얼마입니까?

(사실 문제 (a)와 (c)는 PSPACE-완전이며, 이 문제들이 PSPACE에 속한다는 것을 증명하는 것조차 자명하지 않습니다.)

♣41.

(a) 다음 문제를 해결하는 다항 시간 알고리즘을 설명하십시오: 알파벳 $\Sigma = \{0, 1\}$ 위의 NFA M이 주어졌을 때, M이 *실제로* 받아들이는 문자열이 Σ^*에 존재합니까?

(b) 다음 문제를 해결하는 다항 시간 알고리즘을 설명하십시오: 알파벳 $\Sigma = \{0, 1\}$ 위의 정규 표현식 R이 주어졌을 때, R이 *매치하는* 문자열이 Σ^*에 존재합니까?

(c) 모든 정규 언어의 여집합은 역시 정규 언어입니다. 그렇다면 이 두 알고리즘과 문제 40의 NP-난해성 결과를 함께 사용해도 P=NP임이 증명되지 않는 이유는 무엇입니까?

42. 카론은 최근에 죽은 n명의 사람들을 아케론강 건너 하데스로 실어 날라야 합니다. 이들 중 어떤 쌍은 철천지원수여서, 카론이 함께 있지 않는 한 강의 어느 쪽에서도 함께 있을 수 없습니다. (두 원수만 남겨지면 한 사람이 다른 사람의 입에서 오볼로스 동전을 훔쳐 가고, 동전을 빼앗긴 사람은 영원토록 유령이 되어 아케론강 기슭을 떠돌게 됩니다. 이것은 그저 아주 나쁜 일이라고만 해 둡시다.) 나룻배에는 카론을 포함하여 한 번에 최대 k명까지 탈 수 있으며, 나룻배는 카론만 몰 수 있습니다.[30]

[30]이 문제는 잘 알려진 늑대-염소-양배추 퍼즐을 일반화한 것입니다. 이 퍼즐이 처음 등장한 것으로 알려진 문헌은 주목할 만한 중세 필사본 *Propositiones ad Acuendos Juvenes* [*젊은이들을 단련하기 위한 문제들*]입니다.

> XVIII. Propositio De Homine et Capra et Lvpo.
>
> 어떤 사람이 늑대와 염소와 양배추 한 다발을 강 건너로 옮겨야 했습니다. 그런데 그는 그것들 가운데 단 둘만 실을 수 있는 배밖에 찾을 수 없었습니다. 또한 그는 이 모든 것을 조금도 상하지 않게 건너편으로 옮기라는 명을 받았습니다. 할 수 있는 이는 말해 보십시오. 그는 어떻게 그것들을 상하지 않게 하고 건널 수 있었습니까?
>
> **풀이.** 앞의 경우와 비슷한 방식으로, 저라면 먼저 염소를 데리고 건너가고 늑대와 양배추는 강가에 남겨 두겠습니다. 그런 다음 돌아와서 늑대를 건네 옮기겠습니다. 늑대를 강 건너편에 내려놓은 뒤에는 염소를 배에 태워 다시 데리고 돌아오겠습니다. 염소를 내려놓고 나서는 양배추를 건너편으로 실어 나르겠습니다. 그리고 다시 노를 저어 돌아와 염소를 태우고 건너가겠습니다. 이렇게 하면 무엇도 잡아먹히는 화를 입지 않고 무사히 강을 건널 수 있습니다.

고전 라틴어 실력이 조금 녹슨 몇 안 되는 독자들을 위해 영어 번역을 싣습니다:

> XVIII. 사람과 염소와 늑대의 문제.
>
> 한 남자가 늑대 한 마리, 염소 한 마리, 양배추 한 다발을 강 건너편으로 옮겨야 했습니다. 그런데 그는 자신의 배가 [남자를 포함하여 한 번에] 두 [개체]의 무게만 견딜 수 있다는 것을

카론이 n명 전원을 (뭐, 이미 죽었다는 점만 빼면) 무사히 아케론강 건너편으로 실어 나를 수 있는지 판정하는 문제가 NP-난해임을 증명하십시오. 카론의 문제의 입력은 정수 k와 n, 그리고 서로 적대 관계인 쌍들을 나타내는 정점 n개짜리 그래프 G로 이루어집니다. 출력은 True 또는 False입니다.

풀이를 고전 라틴어로 작성하지 마십시오.

알게 되었습니다. 그리고 그는 모든 것을 무사히 건너편으로 옮겨야 했습니다. 할 수 있다면 말해 보십시오. 그들은 어떻게 무사히 강을 건널 수 있었습니까?

풀이. [앞의 문제와] 비슷한 방식으로, 저는 먼저 염소를 데리고 건너가 늑대와 양배추는 반대편 강둑에 남겨 두겠습니다. 그다음 늑대를 데리고 건너가서, 늑대는 강가에 내려놓고 염소를 다시 데리고 돌아오겠습니다. 그다음 염소는 남겨 두고 양배추를 싣고 건너가겠습니다. 그리고 다시 노를 저어 건너와 염소를 데려가겠습니다. 이렇게 하면 잡아먹힐 위험 없이 무사히 강을 건널 수 있을 것입니다.

*Propositiones*의 저자로 가장 유력한 인물은 다작으로 이름난 8세기 잉글랜드 학자 요크의 앨퀸(Alcuin of York)입니다. 앨퀸이 이 논고의 저자라는 증거는 다소 정황적입니다. 그러나 그가 샤를마뉴와 주고받은 서신을 통해 그가 황제에게 "재미 삼아 풀어 볼 간단한 산술 문제" 몇 가지를 보냈다는 사실은 분명히 알 수 있습니다. 대부분의 현대 학자들은 설령 앨퀸이 실제로 *Propositiones*를 저술했다 하더라도 모든 문제를 직접 고안한 것은 아니며, 오히려 그보다 더 이전의 출처들에서 수집했을 것이라고 봅니다.

어떤 것들은 결코 변하지 않습니다.

잘 고른 예시는 평범한 사람의 마음에 진리를 새기는 데 거의 필수적이며, 어떤 교사도 수업 준비의 이 부분을 소홀히 해서는 안 됩니다.

— Howard Crosby (c.1880)

한 번 보여 주는 것이 백 마디 말보다 낫습니다.

— 앨런 와츠(중국 속담을 잘못 인용함), *선의 길(The Way of Zen)* (1957)

이것을 대수롭지 않은 문제로 여기거나, 그림을 빼는 것의 유일한 이점이 지면 절약뿐이라고 생각하지 마십시오. 교과서의 그림은 실제로 학습 과정을 방해합니다.

— Neville Martin Gwynne, *Gwynne's Grammar* (2013)

이미지 출처

표지를 포함하여 이 책에 실린 모든 그림은 아래에 열거한 것을 제외하면 모두 저자의 창작물입니다. 별도로 명시하지 않는 한, 열거한 작품은 모두 퍼블릭 도메인에 속합니다.

- 그림 0.1 (5쪽) — Biblioteca nazionale Braidense (Milano)
  http://atena.beic.it/webclient/DeliveryManager?pid=2953344
- 그림 0.2 (5쪽) — Internet Archive
  https://archive.org/details/archimedisopera05eutogoog/page/n377
- 그림 1.16(45쪽) — 인터넷 아카이브
  https://archive.org/details/p1rcrationsm00lucauoft/page/162
- 그림 1.25 (61쪽) — 티나 에릭슨(Tina Erickson)이 그린 저자의 크레용 초상화(2000)를 바탕으로 하였으며, 작가의 허락을 받아 수록하였습니다.
- 그림 5.1 (188쪽) — Wikimedia Commons
  https://commons.wikimedia.org/wiki/File:Tabula_Peutingeriana_-_Miller.jpg
- 그림 5.2 (189쪽) — 예일 로스쿨 도서관(Yale Law Library)이 크리에이티브 커먼즈 라이선스로 공개한 "법률 나무(Legal Trees)" 갤러리
  https://www.flickr.com/photos/yalelawlibrary/albums/72157621954683764
- 그림 5.3 (189쪽) — Internet Archive
  https://archive.org/details/A077240124/page/n261

- 연습문제 5.20(216쪽)과 8.22(304쪽) — 저자가 직접 만든 퍼즐로, Jason Batterson과 Shannon Rogers의 *Beast Academy Math: Practice 3A*(2012)에서 착안했습니다.
  https://beastacademy.com/pdf/3A/printables/AngleMazes.pdf
  https://www.beastacademy.com/resources/printables.php
- 그림 10.1(328쪽) — T[homas] E. Harris, F[rank] S. Ross. Fundamentals of a method for evaluating rail net capacities. The RAND Corporation, Research Memorandum RM-1517, 1955년 10월 24일. 퍼블릭 도메인에 속하는 미국 정부 저작물입니다.
  http://www.dtic.mil/dtic/tr/fulltext/u2/093458.pdf

1. 할 말이 있어야 합니다.
2. 그 말을 하십시오.
3. 다 말했으면 멈추십시오.
4. 논문에 알맞은 제목을 붙이십시오.

— John Shaw Billings, “An Address on Our Medical Literature”, International Medical Congress, London (1881)

있잖습니까, 저는 책을 한 권 쓸 수도 있습니다.
그리고 그 책은 황소도 기절시킬 만큼 두꺼울 것입니다.

— 로리 앤더슨(Laurie Anderson), “Let X=X”, *Big Science* (1982)

판권 정보

이 책은 TeXShop(버전 4.27)으로 편집했으며, pdfLaTeX(MacTeX-2018)로 조판했습니다. 조판에는 `memoir` 문서 클래스(`madsen` 장 스타일, `komalike` 머리글 스타일, `Ruled` 페이지 스타일 적용), `amsmath`, `babel`, `enumitem`, `imakeidx`, `mathdesign`, `microtype`, `standalone`을 비롯한 여러 표준 패키지, 그리고 부끄러울 만큼 많은 사용자 정의와 TeX-hAXing을 사용했습니다. 본문은 Bitstream Charter, Ἀρτεμισία, Roboto, `Inconsolata` 서체로 조판했습니다. 이미지 출처에 따로 밝힌 경우를 제외하면, 모든 그림은 저자가 OmniGraffle Pro로 직접 그려 PDF 파일로 내보낸 뒤 `graphicx` LaTeX 패키지를 사용하여 삽입했습니다.

저희 프로그램의 일부는 기계적으로 복제된 것이며, 이것으로 오늘 방송을 모두 마칩니다.