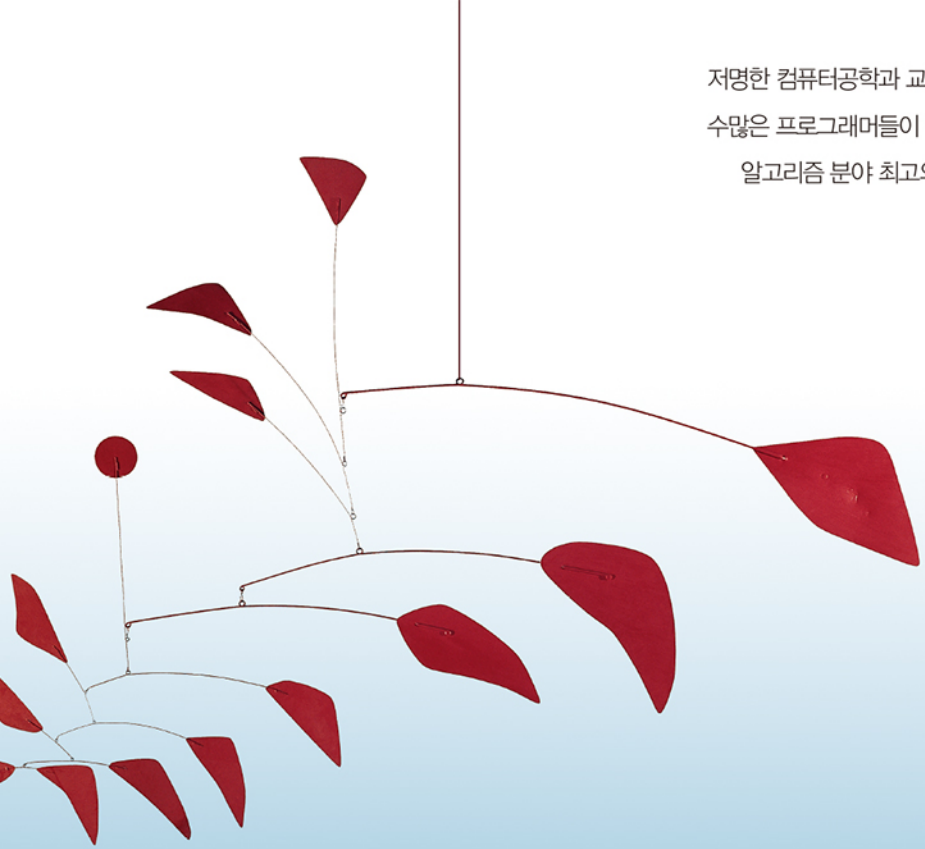
저명한 컴퓨터공학과 교수들과
수많은 프로그래머들이 극찬한
알고리즘 분야 최고의 명저



INTRODUCTION TO ALGORITHMS

THIRD EDITION

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein 지음

문병로(서울대학교 컴퓨터공학부 교수), **심규석**(서울대학교 전기·정보공학부 교수), **이충세**(충북대학교 전기전자·컴퓨터공학부 교수) 옮김

The MIT Press 한빛아카데미 Hanbit Academy, Inc.

저자 서문

알고리즘은 컴퓨터가 등장하기 전에도 존재했다. 그러다 컴퓨터가 대중화되어 더 많은 알고리즘이 생기면서 계산 분야의 핵심으로 자리 잡았다.

이 책은 컴퓨터 알고리즘의 최신 연구를 기반으로 한 종합 기본서다. 다양한 알고리즘을 상당히 높은 수준으로 다루지만, 모든 독자가 이해하기 쉽도록 알고리즘을 설계하고 분석했다. 또한 내용의 깊이나 수학적 정확성을 떨어뜨리지 않으면서도 알고리즘을 가능한 쉽게 설명하려고 노력했다.

각 장에서는 알고리즘과 설계 기법, 응용 분야, 관련 주제를 다룬다. 그리고 프로그래밍 경험이 조금이라도 있으면 누구나 이해할 수 있도록 알고리즘을 "의사코드(pseudocode)"로 작성했다. 이 책에는 알고리즘의 동작을 설명하는 그림이 244개 이상 수록되어 있으며, 효율적인 알고리즘 설계의 중요성을 강조하기 위해 모든 알고리즘 수행시간에 대한 면밀한 분석도 포함하고 있다.

이 책을 학부나 대학원 과정의 알고리즘이나 자료구조 과목에서 주로 활용할 수 있다. 또한 수학적 관점의 주제뿐만 아니라 알고리즘 설계에 대한 공학적 주제도 다루고 있어 전문가가 혼자 학습하기에도 어렵지 않을 것이다.

개정 3판에서는 내용이 전체적으로 새롭게 수정되었다. 전판에서 변화된 부분은 내용의 범위가 확장되고, 새로운 장이 추가됐으며, 의사코드가 보다 능동적인 표현이 가능하도록 개선되었다는 점이다.

이 책으로 가르치는 분들에게

이 책은 융통성 있으면서 동시에 완결성이 있도록 만들어졌다. 학부의 자료구조 과목에서 대학원의 알고리즘 과목에 이르기까지 다양한 과정에서 유용하게 사용할 수 있을 것이다. 또한 한 학기 강의 분량보다 많은 내용을 제공하여 뷔페처럼 강의에서 다루고 싶은 내용을 선택하여 가르칠 수 있다.

여러분은 이 책을 통해 꼭 필요한 장을 중심으로 강의 일정을 조정할 수 있다. 각 장의 필요한 내용을 해당 장에 모두 포함시켰기 때문에 "이 장을 학습하려면, 다른 장의 내용을 먼저 이해해야 하지 않을까"라는 걱정은 조금도 할 필요가 없다. 다소 쉬운 내용은 앞부분에, 조금 난해한 내용은 뒷부분에 배치하였으며 각 절의 마무리는 그 절의 내용이 자연스럽게 완성되도록 구성했다. 그러므로 학부 강의라면 그 장의 앞부분만 가르쳐도 좋을 것이다. 물론, 대학원 강의에서는 모든 장의 내용을 가르칠 수 있다.

이 책에는 957개의 연습문제와 158개의 종합문제가 포함되어 있다. 각 절의 끝에는 연습문제가, 그리고 각 장의 끝에는 종합문제가 실려 있다. 연습문제는 대개 해당 절의 기본 내용을 잘 이해하였는지를 알아보는 간단한 질문이다. 어떤 문제는 학생들 스스로 간단하게 체크만 하면 될 정도고, 어떤 문제는 상당히 실속 있는 어려운 문제므로 과제로 내기에 적당하다. 한편 종합문제는 종종 해당 장에서 다루지 않은 새로운 내용을 소개할 정도로 상당히 고심해서 만들 것들이다. 종합문제는 몇 개의 질문들로 구성되는데, 학생들이 단계별로 해결하면서 해답에 이를 수 있게 도와준다.

이전 판과 달리 개정 3판에서는 교재의 일부 연습문제와 종합문제의 해답을 http://mitpress.mit.edu/algorithms/에서 제공한다. 그러므로 여러분이 할당할 연습문제나 종합문제의 해답이 포함되어 있는지 이 사이트에서 확인하는 것이 좋다. 그리고 이 해답은 점점 늘어날 것이므로 가르칠 때마다 확인해야 할 것이다.

별표(⋆)가 붙은 절이나 연습문제는 학부생들보다는 대학원생들에게 적당한 내용임을 의미한다. 별표가 붙은 절이 별표가 없는 절보다 꼭 어려운 것은 아니지만, 고급 수학에 대한 이해가 다소 필요할 수 있다. 그리고 별표가 붙은 연습문제는 고등 교육 과정에서 배우는 배경 지식이나 어느 정도의 창의력이 필요할 수도 있다.

이 책으로 학습하는 학생들에게

이 책이 학생 여러분이 알고리즘 분야를 이해하는 데 즐거운 입문서가 되었으면 한다. 이 책에서 소개하는 모든 알고리즘이 이해하기 쉽고 재미있도록 노력했고, 익숙치 않거나 어려운 알고리즘의 경우, 잘 이해할 수 있도록 단계별로 서술했다. 또한 알고리즘 분석에 필요한 수학적 내용을 매우 주의 깊게 설명하려고 했다. 만약 이미 잘 알고 있는 주제라면 이를 소개하는 절은 대충 읽고 심화 내용을 바로 학습할 수도 있을 것이다.

이 책은 분량이 많기 때문에 강의 시간에 모두 배우기는 힘들겠지만, 지금 당장 강의 교재의 역할뿐만 아니라 나중에 전문가가 되어서도 수학 참고서나 공학 기술 안내서의 역할을 충분히 할 것이다.

이 책을 이해하려면 먼저 다음과 같은 사항을 알고 있어야 한다.

- 프로그램을 작성해 본 경험이 조금이라도 있어야 한다. 특히, 재귀적 처리 방식이나 배열과 연결 리스트와 같은 간단한 자료구조를 잘 이해하고 있어야 한다.
- 수학적 귀납법을 통한 증명에 어느 정도 익숙해야 한다. 이 책의 상당 부분이 기본적인 미적분학 내용에 기초하기 때문이다. 또한 I부와 VIII부에서 이 책을 이해하는 데 필요한 모든 수학적 기법을 설명한다.

연습문제와 종합문제의 해답을 제공해 달라는 요청이 많았다. http://mitpress.mit.edu/algorithms/에서 연습문제와 종합문제의 해답을 제공한다. 직접 풀이한 답과 웹 사이트에서 제공하는 비교해 보기를 권장한다. 단, 여러분이 직접 풀이한 답을 보낼 필요는 없다.

기술 전문가들에게

이 책은 알고리즘에 관한 다양한 주제를 다루고 있기 때문에 알고리즘에 관한 훌륭한 안내서가 될 수 있다. 각 장의 주제에 필요한 내용을 가능한 한 모두 담아내어 흥미로운 주제에 집중하여 학습할 수 있다.

이 책에서 다루는 대부분의 알고리즘은 대단히 실용적이다. 그래서 구현에 관한 관심사와 그 밖의 기술적 이슈도 자세히 설명했다. 또한 이론적으로 중요한 알고리즘은 실용적인 면에서 대안이 될 수 있는 방법을 설명했다.

이 책에 있는 특정 알고리즘을 완전히 구현하려면 의사코드를 익숙한 프로그래밍 언어로 바꾸어야 할 것이다. 이 책의 의사코드는 알고리즘을 분명하면서도 간결하게 표현할 수 있도록 설계되어 있다. 따라서 오류를 다루는 문제나 특정 프로그래밍 환경을 가정하는 소프트웨어 공학적 이슈는 따로 언급하지 않는다. 그리고 각 알고리즘의 본질이 흐려지지 않도록 특정 프로그래밍 언어의 특성을 배제한 채 간결하면서도 직접적인 방식으로 설명한다.

이 책을 정규 교과 과정이 아닌 다른 용도로 사용해 강의자로부터 연습문제와 종합문제의 해답을 구할 수 없을 것이다. http://mitpress.mit.edu/algorithms/에서 연습문제와 종합문제의 해답 일부를 확인할 수 있다. 여러분이 구한 답을 보낼 필요는 없다.

동료들에게

이 책에서는 광범위한 참고문헌과 이 분야의 최신문헌에 대한 가이드를 제공한다. 각 장이 좀 더 자세한 내용과 역사적인 참고 내용을 담은 참고문헌으로 끝을 맺는다. 하지만 이 참고문헌이 관련 알고리즘에 대한 완벽한 참고를 제공하지는 않는다. 믿기 어렵겠지만, 두께에 대한 부담으로 흥미로운 알고리즘을 많이 포함시키지 못했다.

그동안 많은 학생이 연습문제와 종합문제 해답을 요청해왔지만, 스스로 해답을 찾으려고 노력하기 전에 모범 답안부터 보고싶은 충동을 사전에 없애기 위해 해답을 싣지 않기로 했다.

개정 3판에서의 변화

2판과 비교해 3판에서 변한 내용은 무엇일까? 1판에서 2판으로 변화된 정도와 비슷하다. 그리고 보는 관점에 따라 아주 큰 변화일 수도 있고 아주 작은 변화일 수도 있다.

목차를 대략 비교해 보면 2판에 있던 대부분의 장과 절이 3판에 포함되었다는 사실을 확인할 수 있을 것이다. 장 두 개와 절 하나를 없애는 대신 새로운 장 세 개와 절 두 개를 추가했다.

1판과 2판은 복합적인 구조로 유지하였다. 문제 영역이나 기술 한 분야로만 구성하기보다는 양쪽을 다 고려하여 구성하였다. 기술 분야 장으로는 분할-정복, 동적 프로그래밍, 그리디 알고리즘, 분할상환 분석, NP-완비성, 근사 알고리즘 등이 있다. 또한 정렬의 모든 부분, 동적인 집합에 대한 자료구조, 그래프 문제에 대한 알고리즘도 담고 있다. 독자가 알고리즘의 설계와 분석에 기술을 적용하는 방법을 알아야 하므로 연습문제는 기술을 적용하여 해결하는 방법을 거의 알려주지 않는다.

3판의 중요한 변화를 요약하면 다음과 같다.

- 엠데 보아스 트리와 멀티스레드 알고리즘 장을 새로 추가했고, 부록에 행렬의 기초 내용을 포함시켰다.
- 점화식 장을 다양한 분할-정복 기법을 다루는 장으로 변경했는데, 첫 두 절에서는 두 가지 문제를 분할-정복으로 해결하는 방법을 다룬다. 그리고 두 번째 절에서는 행렬의 곱에 대한 스트라센 알고리즘을 제시하는데 이것은 행렬 연산 장에서 옮겨 온 내용이다.
- 3판에서는 거의 다루지 않는 이항 힙과 정렬 네트워크 장을 삭제했다. 정렬 네트워크에 있던 핵심 아이디어는 종합문제 8-7에서 비교-교환 알고리즘에 대한 0-1 정렬 정리로 보여준다. 그리고 피보나치 힙을 더 이상 이항 힙이 선행되어 의존적이지 않게 다룬다.
- 동적 프로그래밍과 그리디 알고리즘을 수정했다. 동적 프로그래밍은 2판의 조립 라인 스케줄링 문제보다 재미있는 막대 자르기 문제로 시작한다. 게다가 2판보다 메모 방식을 좀 더 강조하여, 그래프의 부분 문제 개념을 동적 프로그래밍 알고리즘 수행시간을 이해하는 수단으로 도입했다. 그리디 알고리즘의 개방형 예인 활동 선택 문제를 2판보다 그리디 알고리즘에 더 직접적으로 이르게 하였다.
- 이진 검색 트리(레드-블랙 트리 포함)에서 노드를 삭제하는 방법이 삭제를 요청한 노드가 실제로 삭제 노드가 되는 것을 보장한다. 1판과 2판에서는 내용이 노드로

이동해 삭제 프로시저에 전달되기 때문에 다른 노드가 삭제될 수 있었다. 노드를 삭제하는 새로운 방법을 이용하면 프로그램의 다른 구성 요소가 트리에 있는 노드에 포인터를 유지하게 되어 삭제한 노드를 나타내는 비정상적인 포인터로 끝나는 실수로 끝나지 않을 것이다.

- 플로우 네트워크 장은 전적으로 간선에서 플로우하는 것을 기본으로 한다. 이런 접근은 1, 2판에서 사용한 네트워크 플로우보다 좀 더 직관적이다.
- 행렬의 기초 내용과 스트라센 알고리즘을 다른 장으로 옮기고 행렬 연산 장을 2판보다 축소했다.
- 크누스-모리스-프랫에 의한 스트링-매칭 알고리즘은 수정하였다.
- 여러 오류를 수정하였다. 이 오류들은 웹 사이트에 등록된 2판의 오류다.
- 많은 요청에 의해 의사코드의 문장을 바꾸었다. C, C++, 자바, 파이썬처럼 "="은 대입(assignment)을 나타내고 "=="은 같은지를 검사한다. 마찬가지로 키워드 **do**와 **then**을 제거하고 "//"를 행의 주석 기호로 사용한다. 그리고 의사코드는 객체지향보다 절차적으로 유지한다. 즉, 객체에 대해 메소드를 실행하기보다는 객체를 인수로 전달하는 프로시저 호출을 사용한다.
- 새로운 연습문제 100개 종합문제 28개를 실었다. 또한 참고문헌을 갱신하고 추가하였다.
- 마지막으로, 명확하고 생동감 있게 만들기 위해 책 전반적으로 문장, 단락, 절을 재작성하였다.

웹 사이트

http://mitpress.mit.edu/algorithms/을 이용하면 보충 자료를 얻을 수 있고, 또한 우리와 의사소통도 할 수 있다. 웹 사이트에는 오류 목록, 연습문제와 종합문제의 해답, 촌스러운 교수들의 농담이 연결되어 있으며 다른 내용이 더 추가될 수도 있다. 또한 웹 사이트는 오류를 보고하고 의견을 제시하는 방법도 알려준다.

이 책이 출판되기까지의 이야기

2판과 마찬가지로 3판에서도 LaTeX 2_ε로 작업했다. MathTime Pro 2 폰트를 사용하는 수학 타입셋을 가진 Times 폰트를 사용했다. Publish or Perish, Inc.의 Michael Spivak, Personal TeX, Inc.의 Lance Carnes, 그리고 Dartmout 대학의 Tim Tregubovd의 기술 지원에 감사드린다. 1, 2판과 마찬가지로 Windex를 이용해 색인을 컴파일했고, 참고문헌은 BibTeX을 이용해 만들었다. 그리고 이 책의 PDF 파일은 OS 10.5에서 동작하는 MacBook에서 만들었다.

MacDraw Pro를 이용해 3판의 그림을 그렸고, 그림에 있는 수학 수식은 LaTeX 2ε용 psfrag 패키지로 앉혔다. 불행히도 MacDraw Pro는 레거시 소프트웨어로 지난 10여 년간 시장에서 사용되지 않았는데, 다행히 OS 10.4 환경에서 동작하는 맥킨토시를 소수 가지고 있어 이를 이용해 MacDraw Pro를 대부분 실행했다. MacDraw Pro는 구식 환경에서도 컴퓨터 과학 교재의 그림을 그리는 데 다른 드로잉 소프트웨어보다 훨씬 사용이 쉬웠고, 결과물도 훌륭했다. [1] Mac을 언제까지 사용할지 알 수 없지만, 만약 애플 관계자가 이 책을 본다면 *MacDraw Pro OS-X 호환용 프로그램을 만들어 달라고 부탁하고 싶다!*

3판에 대한 감사의 글

MIT Press와 함께 일을 한지 20년 넘었고 우리는 무척 친밀하다. 특히 Ellen Faran, Bob Prior, Ada Brunstein, Mary Reilly의 도움과 지원에 감사드린다.

3판을 준비하는 동안, Dartmouth 대학의 전산학과, MIT 전산학과와 인공지능 연구실, Columbia 대학교의 산업공학과와 운영체제연구학과 등 다양한 장소에 떨어져 작업하였다. 물심 양면으로 지원과 충고를 아끼지 않은 대학과 동료들에게 감사드린다.

이번에도 Julie Sussman, P.P.A.가 기술적인 편집을 도와주었다. 저자들이 놓친 오류를 찾아주었고 여러 문장의 표현을 다듬어 주었다. 기술 편집에 대한 명예의 전당이 있다면 Julie가 당연히 뽑혀야 할 것이다. Julie에게 말로 표현할 수 없는 감사를 전한다. Priya Natarajan도 책이 발간되기 전에 오류를 찾아주었다. 이 책에서 오류를 남아 있다면 이는 Julie가 읽은 후에 추가된 것이므로 그 책임은 모두 저자들에게 있음을 밝힌다.

van Emde Boas의 트리의 내용은 Erik Demaine의 노트에서 인용했는데, 이 내용은 Michael Bender의 영향을 받은 것이다. 또한 3판에서는 Javed Aslam, Bradley Kuszmaul, 그리고 Hui Zha의 아이디어를 활용하였다.

멀티스레드 장은 Harald Prokop와 공동으로 작성한 노트를 기반으로 한다. 또한 이 자료는 MIT의 Bradley Kuszmaul와 Matteo Frigo가 소속된 Cilk 프로젝트의 영향을 받았다. 멀티스레드 의사코드를 설계하는 데 MIT의 Cilk extensions to C와 Cilk Arts의 Cilk++ extensions to C++로부터 영감을 얻었다.

1판과 2판을 읽은 많은 독자가 책의 오류를 알려주었고 이 책을 개선하는 방안을 제시해 주었는데, 이런 독자들에게 감사를 드린다. 보고해 준 모든 오류를 교정하였고

[1] Mac OS X에서 동작하는 여러 드로잉 프로그램을 MacDraw Pro와 비교했는데, 대다수가 치명적인 결함이 있었다. 유명한 다른 드로잉 프로그램을 이용해 이 책에 있는 그림을 그리기 위해 여러 차례 시도해보았는데, MacDraw Pro로 작업할 때보다 무려 5배의 시간이 걸렸고 결과물도 좋지 않았다. 그래서 MacDraw Pro를 사용하기로 했다.

최대한으로 독자들의 제안을 따랐다. 이런 독자가 많다는 데 즐거움을 느끼고 있으며 앞으로도 계속 의견에 귀를 기울일 것이다.

마지막으로, 이 책을 준비하는 동안 우리들의 아내인 Nicole Cormen, Wendy Leiserson, Gail Rivest, Rebecca Ivry와 우리들의 자녀인 Ricky, Will, Debby, Katie Leiserson; Alex와 Christopher Rivest; 그리고 Molly, Noah, Benjamin Stein이 보여준 지원과 사랑에 감사한다. 가족들의 충고와 인내 덕분에 이 책의 출판이 가능했다. 이 책을 가족들에게 바친다.

토마스 코멘(THOMAS H. CORMEN) *레바논, 뉴 햄프셔에서*
찰스 레이서손(CHARLES E. LEISERSON) *캠브리지, 메사추세스에서*
로날드 르베스트(RONALD L. RIVEST) *캠브리지, 메사추세스에서*
클리포드 스타인(CLIFFORD STEIN) *뉴욕, 뉴욕에서*

2009년 2월

4 분할정복

2.3.1절에서 분할정복 체계의 한 가지 예로 병합 정렬이 동작하는 방법을 보았다. 분할정복에서는 각 재귀 호출 레벨 위에서 다음 세 가지 단계를 거치면서 재귀적으로 문제를 푼다.

분할: 현재의 문제와 동일하되 입력의 크기가 더 작은 다수의 부분 문제로 분할한다.

정복: 부분 문제를 재귀적으로 풀어서 정복한다. 부분 문제의 크기가 충분히 작으면 직접적인 방법으로 푼다.

결합: 부분 문제의 해를 결합해 원래 문제의 해가 되도록 만든다.

부분 문제가 재귀적으로 풀 수 있을 만큼 충분히 클 때 ***재귀 대상***이라고 한다. 부분 문제가 충분히 작아져 더 이상 재귀 호출을 할 수 없을 때 재귀가 "바닥을 쳤다"고 하고, ***베이스 케이스***까지 내려왔다고 이야기한다. 때로는 입력의 크기가 더 작은 완전히 동일한 부분 문제 외에 원래의 문제와 다른 부분 문제를 풀어야 할 때도 있다. 그런 부분 문제는 결합 단계의 일부로 간주한다.

이 장에서는 더 많은 분할정복 기반의 알고리즘을 살펴볼 것이다. 첫 번째 알고리즘은 최대 부분 배열 문제(maximum-subarray problem)를 푸는 알고리즘이다. 최대 부분 배열 문제는 배열을 입력으로 받아 가장 큰 합을 가지는 연속 부분 배열을 찾는 문제다. 다음으로는 $n \times n$ 행렬을 곱하는 두 가지 분할정복 알고리즘을 살펴볼 것이다. 하나는 $\Theta(n^3)$ 시간에 수행되는데, 정사각 행렬을 곱하는 일반적인 방법보다 나을 것이 없다. 그러나 다른 하나인 스트라센 알고리즘(Strassen's algorithm)은 $O(n^{2.81})$ 시간에 수행되며 이는 일반적인 방법보다 점근적으로 더 낫다.

점화식

점화식은 분할정복 체계와 관련이 많다. 점화식을 통해 분할정복 알고리즘의 수행시간을 자연스럽게 표현할 수 있기 때문이다. ***점화식***(recurrence)은 더 작은 입력에 대한 자신의 값으로 함수를 나타내는 방정식 또는 부등식이다. 예를 들어, 2.3.2절에서

MERGE-SORT 프로시저의 최악의 경우 수행시간 $T(n)$이 다음 점화식으로 표현될 수 있음을 보았다.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases} \tag{4.1}$$

이 점화식의 해는 $T(n) = \Theta(n \lg n)$이다.

점화식은 여러 형태가 될 수 있다. 예를 들어, 어떤 재귀 알고리즘은 부분 문제를 2/3와 1/3 분할처럼 비균등하게 나눌 수도 있다. 분할과 결합 단계에 선형 시간이 걸린다면 이 알고리즘에서 점화식 $T(n) = T(2n/3) + T(n/3) + \Theta(n)$을 이끌어 낼 수 있을 것이다.

부분 문제를 원래 문제 크기의 상수 분할로 제한할 필요는 없다. 예를 들어, 선형 검색(연습문제 2.1-3 참고)의 재귀 버전에서는 원래 문제보다 원소가 딱 한 개 적은 부분 문제가 하나만 존재한다. 각 재귀 호출에 걸리는 시간은 상수 시간과 새로운 재귀 호출에 필요한 시간을 더한 것이 되므로 점화식은 다음이 된다. $T(n) = T(n-1) + \Theta(1)$

이 장에서는 점화식을 푸는 세 가지 방법, 즉 해의 점근적 "Θ"나 "O" 한계를 얻는 방법을 제시한다.

- ***치환법***(substitution method)에서는 한계를 추측한 후 그 추측식이 옳음을 증명하기 위해 수학적 귀납법을 이용한다.
- ***재귀 트리 방법***(recursion-tree method)은 점화식을 각 노드가 재귀 호출의 해당 레벨에 따른 비용을 나타내도록 만든 트리로 변환하여 점화식을 푼다. 그리고 이를 위해 합의 한계를 구하는 기법을 이용한다.
- ***마스터 방법***(master method)은 다음과 같은 형식으로 된 점화식의 답을 제시해준다.

  $$T(n) = aT(n/b) + f(n) \tag{4.2}$$

  여기서 $a \geq 1, b > 1$이고 $f(n)$은 주어진 함수다. 이런 점화식은 자주 볼 수 있다. 식 (4.2) 형태의 점화식은 a개의 부분 문제를 만들어 내는 분할정복 알고리즘을 표현하고 있다. 각 부분 문제는 크기가 원래 문제의 $1/b$이고, 분할과 결합 과정은 합쳐서 $f(n)$ 시간이 걸린다.

  마스터 방법을 사용하려면 세 가지 경우에 대한 암기가 필요하지만 한 번 기억해 놓으면 간단한 점화식의 점근적 한계를 쉽게 알아낼 수 있다. 이제 최대 부분 배열 문제와 행렬 곱셈을 위한 분할정복 알고리즘의 수행시간을 알아내는데 마스터 방법을 사용할 것이다. 또한 이 책의 다른 부분에서도 분할정복 기반의 알고리즘에 마스터 방법을 사용한다.

때로는 $T(n) \leq 2T(n/2) + \Theta(n)$과 같이 등식이 아닌 부등식 형태의 점화식도 보게 될 것이다. 이런 점화식은 $T(n)$에 대한 상한만을 나타내기 때문에 해를 표현할 때 Θ-표기가 아닌 O-표기를 사용하게 된다. 부등식이 반대로 $T(n) \geq 2T(n/2) + \Theta(n)$인 경우에도 점화식이 $T(n)$에 대한 하한만 나타내기 때문에 해에 Ω-표기를 사용하게 된다.

점화식에서의 기술적 사항

점화식을 기술하고 해를 구하는 데 일부 기술적인 세부 사항은 무시할 것이다. 예를 들어, n이 홀수일 때 n개의 원소에 대해 MERGE-SORT를 호출한다면 $\lfloor n/2 \rfloor$과 $\lceil n/2 \rceil$ 크기의 부분 문제를 얻게 된다. n이 홀수면 $n/2$가 정수가 아니므로 두 부분 문제의 크기가 실제로는 $n/2$이 아니다. MERGE-SORT의 최악의 경우 수행시간을 나타내는 실제 점화식은 기술적을 다음과 같다.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 \end{cases} \tag{4.3}$$

한계 조건도 흔히 무시하는 세부 사항이다. 상수 크기의 입력에 대한 알고리즘의 수행시간은 상수므로 알고리즘의 수행시간에 대한 점화식은 일반적으로 충분히 작은 n에 대해 $T(n) = \Theta(1)$이다. 따라서 보통은 편의를 위해 점화식의 한계 조건에 대한 서술은 생략하고, 작은 n에 대해 $T(n)$이 상수라고 가정한다. 예를 들어, 점화식 (4.1)에서 작은 n에 대한 명시적인 값을 생략하고 다음과 같이 쓴다.

$$T(n) = 2T(n/2) + \Theta(n) \tag{4.4}$$

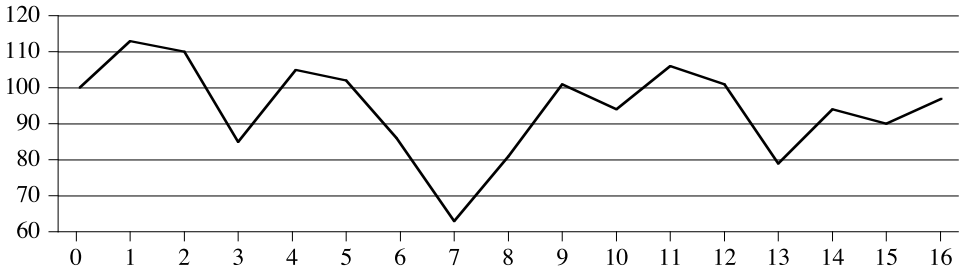
이렇게 하는 이유는 $T(1)$ 값의 변화가 점화식의 정확한 해를 바꾸긴 하지만 그 해는 대개 상수배 이상 바뀌지 않아 증가 차수가 바뀌지 않기 때문이다.

점화식을 서술하고 해를 구할 때는 주로 내림, 올림, 한계 조건을 생략한다. 이런 세부 사항 없이 풀고 난 후에 이를 고려할 필요가 있는지를 결정한다. 대부분의 경우에는 고려할 필요가 없지만 필요한 경우가 언제인지를 아는 것은 중요하다. 그리고 분할정복 알고리즘을 표현하는 많은 점화식에서 이 세부 사항이 점근적 한계에 영향을 끼치지 않음을 말해 주는 몇몇 정리와 경험이 도움이 될 것이다(정리 4.1 참고). 그러나 이 장에서는 점화식 해법의 미세한 부분을 보여주기 위해 이 세부 사항 중 몇 가지를 다룰 것이다.

4.1 최대 부분 배열 문제

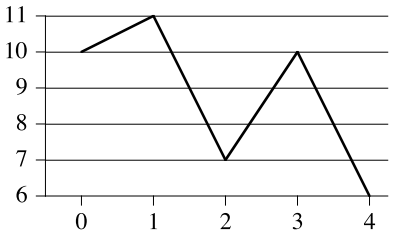
휘발성 화학물질 기업에 투자할 기회가 있다고 가정해 보자. 이 회사가 생산하는 화학물질처럼 휘발성 화학물질 기업의 주가도 상당히 휘발성이 강하다. 당신은 딱 한 번, 한 단위의 주식을 살 수 있고 나중에 그것을 팔 수 있다. 매매는 그 날의 거래가 끝난 후에 이루어진다. 이런 제약에 대한 보상으로, 당신은 미래에 주가가 얼마가 될 것인지 알 수 있다. 당신의 목표는 이익을 최대화하는 것이다. 그림 4.1은 17일 동안의 주가를 보여준다. 가격이 주당 $100인 0일에 시작해 언제든 한 번 주식을 살 수 있다. 물론 이익을 최대화하기 위해 "싸게 사서, 비싸게 팔고" 싶을 것이다. 즉, 가능한 한 가장 낮은 가격에 구입하고 이후에 가능한 한 가장 높은 가격에 파는 것이다. 불행히도 당신은 주어진 기간 중에 가장 낮은 가격에 사고 가장 높은 가격에 팔지 못할 수도 있다. 그림 4.1에서 7일의 가격이 가장 낮은데 이 날은 가장 가격이 높은 날인 1일 이후다.

가장 낮은 가격에 사거나 가장 높은 가격에 팔거나, 둘 중 하나만 해도 이익을 최대화할 수 있을 것이라고 생각할 수도 있다. 예를 들어, 그림 4.1에서 7일에 가장 낮은 가격에 사면 이익을 최대화할 수 있다. 이 전략이 항상 유효하다면 이익을 최대화하는 방법을 알아내는 것이 쉬워질 것이다. 가장 높은 가격과 가장 낮은 가격을 찾아 가장 높은 가격으로부터 왼쪽으로 가면서 이전의 가장 낮은 가격을 찾고, 가장 낮은 가격으로부터 오른쪽으로 가면서 이후의 가장 높은 가격을 찾은 후, 더 큰 차이를 보이는 쌍을 선택한다. 그림 4.2는 간단한 반례를 나타내고 있는데, 어떤 경우에는 최저 가격에 사거나 최대 가격에 팔지 않아도 최대 이익이 발생함을 보여준다.



날짜	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
가격	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
변동		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

그림 4.1 휘발성 화학물질 기업의 거래 종료 후 주가에 대한 17일 동안의 정보. 도표의 가로축은 날짜를, 세로축은 가격을 나타낸다. 표의 마지막 행은 전날과의 가격 차이다.



날짜	0	1	2	3	4
가격	10	11	7	10	6
변동		1	−4	3	−4

그림 4.2 최대 이익이 항상 최저 가격에서 시작하거나 최대 가격으로 끝나지 않음을 보여주는 예. 마찬가지로 가로축은 날짜를, 세로축은 가격을 나타낸다. 2일에 구입하고 3일에 판매함으로써 주당 $3의 최대 이익이 발생한다. 2일의 $7는 전체에서 최저 가격이 아니며, 3일의 $10는 전체에서 최고 가격이 아니다.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

maximum subarray

그림 4.3 최대 부분 배열 문제에서의 주가 변동값. 배열 A의 모든 연속 부분 배열 중에서 합이 가장 큰 부분 배열은 합이 43인 $A[8..11]$이다.

주먹구구 해

이 문제에 대한 주먹구구 해는 쉽게 얻을 수 있다. 매입일이 매도일보다 먼저 있는, 가능한 모든 매매일 쌍을 시도해 보는 것이다. n일의 기간 동안 그런 쌍이 $\binom{n}{2}$개 존재한다. $\binom{n}{2}$는 $\Theta(n^2)$이고, 각 매매일 쌍을 평가하는 데 최소한 상수 시간이 걸리므로 이 접근법은 $\Omega(n^2)$ 시간이 걸린다. 더 잘할 수 있을까?

변환

$o(n^2)$ 시간이 걸리는 알고리즘을 설계하기 위해 입력을 조금 다른 방식으로 보고자 한다. 찾고자 하는 것은 첫 날부터 마지막 날까지의 가격 차이를 최대로 하는 연속된 날짜다. 일별 가격을 보는 대신 일별 가격 변동을 고려해보자. i일의 가격 변동은 $i-1$일의 가격과 i일의 가격 차이를 의미한다. 그림 4.1 속 표의 마지막 행에 이런 일별 가격 변동이 나타나 있다. 그림 4.3처럼 이 행을 배열 A로 취급한다고 했을 때, 이제 비지 않고 연속된 A의 부분 배열 중에서 가장 큰 합을 가지는 것을 찾으려고 한다. 이런 연속 부분 배열을 ***최대 부분 배열***이라고 하자. 예를 들어, 그림 4.3의 배열에서 $A[1..16]$의 최대 부분 배열은 합이 43인 $A[8..11]$이다. 따라서 당신은 8일 직전(즉, 7일 거래 종료 후)에 주식을 사서 11일 거래 종료 후 팔고 싶을 것이다. 이익은 주당 $43이다.

언뜻 보기에 이런 변환은 쓸모없어 보인다. 여전히 n일의 기간 동안 $\binom{n-1}{2} = \Theta(n^2)$개의 부분 배열을 검사해야 한다. 연습문제 4.1-2는 부분 배열 하나의 비용을 계산하는 데 부분 배열의 길이에 비례하는 시간이 걸림에도 불구하고, $\Theta(n^2)$개의 부분 배열 합

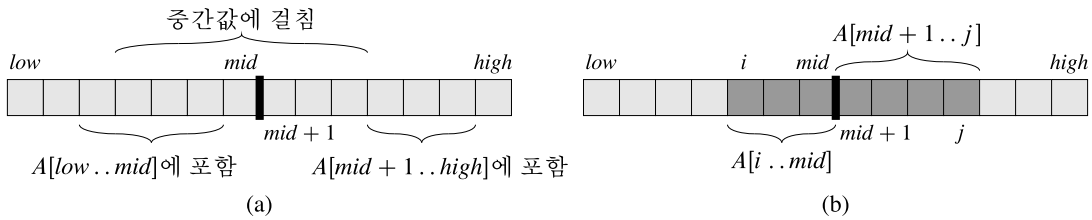


그림 4.4 (a) $A[low\,..\,high]$의 부분 배열이 가능한 위치: $A[low\,..\,mid]$에 완전히 포함되는 경우, $A[mid+1\,..\,high]$에 완전히 포함되는 경우, 중간값 mid에 걸쳐 있는 경우. (b) 중간값에 걸쳐 있는 $A[low\,..\,high]$의 모든 부분 배열은 두 부분 배열 $A[i\,..\,mid]$와 $A[mid+1\,..\,j]$로 이루어진다. 이 때 $low \le i \le mid$이고 $mid < j \le high$이다.

을 모두 구할 때 미리 계산된 부분 배열 합이 주어지면 각 부분 배열의 합을 구하는 데 $O(1)$의 시간이 걸리도록 만들 수 있다. 따라서 주먹구구 해를 구하는 데 $\Theta(n^2)$ 시간이 걸림을 보이는 문제다.

이제 최대 부분 배열 문제를 푸는 더 효율적인 방법을 찾아보자. 이 과정에서 "최대 부분 배열"이라는 용어보다는 "최대 부분 배열 중 하나"라는 용어를 자주 사용할 것인데, 이는 최대 합을 가지는 부분 배열이 하나 이상 존재할 수도 있기 때문이다.

최대 부분 배열 문제는 배열이 음수를 포함할 때만 흥미가 생긴다. 배열의 모든 항목이 음수가 아니라면 전체 배열이 최대 합이 되기 때문에 이런 최대 부분 배열 문제는 어렵지 않다.

분할정복을 이용한 해

분할정복 기법을 이용해 최대 부분 배열 문제를 푸는 방법을 생각해보자. 부분 배열 $A[low\,..\,high]$의 최대 부분 배열 중 하나를 찾는다고 가정하자. 분할정복에서는 부분 배열을 가능하면 같은 크기의 두 부분 배열로 나누는 것이 좋다. 즉, 부분 수열의 중간값(mid라고 하자)을 찾고 두 부분 수열 $A[low\,..\,mid]$와 $A[mid+1\,..\,high]$를 생각해보자. 그림 4.4(a)와 같이 $A[low\,..\,high]$의 모든 연속 부분 배열 $A[i\,..\,j]$는 다음 중 하나에 반드시 포함된다.

- 부분 배열 $A[low\,..\,mid]$에 완전히 포함되는 경우다. 따라서 $low \le i \le j \le mid$.
- 부분 배열 $A[mid+1\,..\,high]$에 완전히 포함되는 경우다. 따라서 $mid < i \le j \le high$.
- 중간값에 걸쳐 있는 경우다. 따라서 $low \le i \le mid < j \le high$.

따라서 $A[low\,..\,high]$의 최대 부분 배열은 반드시 이들 중 하나에 포함된다. 즉, $A[low\,..\,high]$의 최대 부분 배열은 $A[low\,..\,mid]$에 포함되거나, $A[mid+1\,..\,high]$에 포함되거나, 중간값에 걸쳐 있는 모든 부분 배열 중 가장 큰 합을 가져야 한다. $A[low\,..\,mid]$와 $A[mid+1\,..\,high]$의 최대 부분 배열은 재귀적으로 찾을 수 있다. 이 두 부분 문제는 최대 부분 배열 중 하나를 찾는 더 작은 문제기 때문이다. 이제 남은

것은 중간값에 걸쳐 있는 최대 부분 배열 중 하나를 찾고 세 부분 배열 중 합이 가장 큰 것을 선택하는 일이다.

중간값에 걸쳐 있는 최대 부분 배열 중 하나는 부분 배열 $A[low\,..\,high]$의 크기에 선형적으로 비례하는 시간에 쉽게 찾을 수 있다. 이 문제는 원래 문제보다 더 작은 동일한 문제가 *아니다*. 부분 배열이 중간값에 걸쳐야 한다는 제약이 생겼기 때문이다. 그림 4.4(b)와 같이 중간값에 걸쳐있는 모든 부분 배열은 두 개의 부분 배열 $A[i\,..\,mid]$와 $A[mid+1\,..\,j]$로 이루어져 있다. 이때 $low \le i \le mid$이고 $mid < j \le high$이다. 따라서 $A[i\,..\,mid]$와 $A[mid+1\,..\,j]$ 형태의 최대 부분 배열을 찾아 합치기만 하면 된다. 프로시저 FIND-MAX-CROSSING-SUBARRAY는 배열 A와 인덱스 $low, mid, high$를 입력으로 받아 중간값에 걸쳐있는 최대 부분 배열의 경계 인덱스 쌍과 최대 부분 배열의 합을 반환한다.

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4      sum = sum + A[i]
 5      if sum > left-sum
 6          left-sum = sum
 7          max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11      sum = sum + A[j]
12      if sum > right-sum
13          right-sum = sum
14          max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

프로시저는 다음과 같이 진행된다. 1–7행은 왼쪽 절반, $A[low\,..\,mid]$의 최대 부분 배열을 찾는다. 이 부분 배열은 $A[mid]$를 포함해야 하므로 3–7행의 **for** 루프가 인덱스 i를 mid에서 시작해 low까지 감소시킨다. 따라서 이 루프가 고려하는 모든 부분 배열은 $A[i\,..\,mid]$의 형태다. 1–2행은 지금까지 찾은 가장 큰 합을 저장하는 변수 *left-sum*과 $A[i\,..\,mid]$의 항목들의 합을 저장하는 변수 *sum*을 초기화한다. 5행에서 *left-sum*보다 큰 합을 가지는 부분 배열 $A[i\,..\,mid]$를 찾으면 6행에서 *left-sum*을 이 부분 배열의 합으로 갱신하고, 7행에서 변수 *max-left*에 인덱스 i를 저장한다. 8–14행은 오른쪽 절반, $A[mid+1\,..\,high]$에 대해 같은 과정을 행한다. 10–14행의 **for** 루프는 인덱스 j를 $mid+1$에서 시작해 $high$까지 증가시킨다. 따라서 이 루프가 고려하는 모든 부분 배열은 $A[mid+1\,..\,j]$의 형태다. 마지막으로, 15행은 중간값에 걸쳐 있는

최대 부분 배열의 경계를 표시하는 인덱스 *max-left*와 *max-right*, 그리고 부분 배열 $A[max\text{-}left\,..\,max\text{-}right]$의 합 $left\text{-}sum + right\text{-}sum$을 반환한다.

부분 배열 $A[low\,..\,high]$이 원소를 n개의 가지고 있다면($n = high - low + 1$이 되도록), FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$ 호출에 $\Theta(n)$ 시간이 걸린다고 할 수 있다. 두 개의 **for** 루프 각 반복당 $\Theta(1)$ 시간이 걸리므로, 합쳐서 반복이 몇 번 일어나는지 세기만 하면 된다. 3–7행의 **for** 루프에서는 $mid - low + 1$번의 반복이 일어나고, 10–14행의 **for** 루프에서는 $high - mid$번의 반복이 일어나므로 전체 반복 횟수는 다음과 같다.

$$\begin{aligned}(mid - low + 1) + (high - mid) &= high - low + 1 \\ &= n\end{aligned}$$

선형 시간이 걸리는 FIND-MAX-CROSSING-SUBARRAY 프로시저를 가지고, 최대 부분 배열 문제를 푸는 분할정복 알고리즘에 대한 의사코드를 작성할 수 있다.

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2      return (low, high, A[low])          // 베이스 케이스: 단 하나의 원소
 3  else mid = ⌊(low + high)/2⌋
 4      (left-low, left-high, left-sum) =
            FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5      (right-low, right-high, right-sum) =
            FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6      (cross-low, cross-high, cross-sum) =
            FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7      if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8          return (left-low, left-high, left-sum)
 9      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10          return (right-low, right-high, right-sum)
11      else return (cross-low, cross-high, cross-sum)
```

첫 호출 FIND-MAXIMUM-SUBARRAY$(A, 1, A.length)$는 $A[1\,..\,n]$의 최대 부분 배열 중 하나를 찾는다.

FIND-MAX-CROSSING-SUBARRAY와 유사하게, 재귀 프로시저 FIND-MAXIMUM-SUBARRAY는 최대 부분 배열의 경계를 표시하는 인덱스 쌍과 최대 부분 배열의 합을 반환한다. 1행은 원소가 하나밖에 없는 부분 배열, 즉 베이스 케이스에 대한 검사를 수행한다. 원소가 하나인 부분 배열은 단 하나의 부분 배열(자기 자신)을 가지므로 2행은 유일한 원소의 시작과 끝 인덱스 쌍, 그리고 원소의 값을 반환한다. 3–11행은

재귀 대상을 다룬다. 3행은 분할을 담당하여 중간값의 인덱스 mid를 계산한다. 부분 배열 $A[low..mid]$를 ***왼쪽 부분 배열***이라 하고 $A[mid+1..high]$를 ***오른쪽 부분 배열***이라 하자. 부분 배열 $A[low..high]$가 적어도 두 개의 원소를 포함하므로 왼쪽과 오른쪽 부분 배열은 각각 적어도 한 개의 원소를 가져야 한다. 4행과 5행은 왼쪽과 오른쪽 부분 배열에서 각각 재귀적으로 최대 부분 배열을 찾아서 정복한다. 6–11행은 결합을 담당한다. 6행은 중간값에 걸쳐 있는 최대 부분 배열을 찾는다(6행이 푸는 부분 문제는 크기가 더 작지만 원래 문제와 동일한 문제가 아니므로 결합으로 간주한다는 사실을 상기하자). 7행은 왼쪽 부분 배열이 최대 합을 가진 부분 배열을 포함하는지 검사하고, 8행은 최대 부분 배열을 반환한다. 반대로 9행은 오른쪽 부분 배열이 최대 합을 가진 부분 배열을 포함하는지 검사하고, 10행은 최대 부분 배열을 반환한다. 만약 왼쪽과 오른쪽 부분 배열 둘 다 최대 합을 가지는 부분 배열을 포함하지 않으면, 최대 부분 배열은 중간값에 걸쳐 있어야 하고 11행이 이를 반환한다.

분할정복 알고리즘 분석하기

다음으로 재귀 프로시저 FIND-MAXIMUM-SUBARRAY의 수행시간을 표현하는 점화식을 만들어 보자. 2.3.2절에서 병합 정렬을 분석할 때 한 것처럼 모든 부분 문제의 크기가 정수가 되도록 원래 문제의 크기가 2의 거듭제곱이라고 가정하자. n개의 원소를 가지는 부분 배열에 대한 FIND-MAXIMUM-SUBARRAY의 수행시간을 $T(n)$으로 표시한다. 먼저 1행은 상수 시간에 수행된다. 베이스 케이스인 $n = 1$일 때는 쉽다. 2행은 상수 시간에 수행되므로 다음과 같다.

$$T(1) = \Theta(1) \tag{4.5}$$

재귀 대상은 $n > 1$인 경우다. 1행과 3행은 상수 시간에 수행된다. 4행과 5행에서 푸는 각 부분 문제는 $n/2$개의 원소로 이루어진 부분 배열에 관한 것이기 때문에(원래 문제의 크기가 2의 거듭제곱이라고 가정했으므로 $n/2$는 정수다), 각각을 푸는 데 $T(n/2)$ 시간이 걸린다. 두 개의 부분 문제(왼쪽 부분 문제와 오른쪽 부분 문제)를 풀어야 하므로 4행과 5행은 $2T(n/2)$ 수행시간을 차지한다. 이미 보았듯이 6행에서 FIND-MAX-CROSSING-SUBARRAY 호출은 $\Theta(n)$ 시간이 소요된다. 7–11행은 단 $\Theta(1)$ 시간이 걸린다. 따라서 이 재귀 대상의 점화식은 다음과 같다.

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n) \end{aligned} \tag{4.6}$$

식 (4.5)와 (4.6) 을 통해 FIND-MAXIMUM-SUBARRAY의 수행시간 $T(n)$에 대한 다음 점화식을 얻을 수 있다.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases} \tag{4.7}$$

이 점화식은 병합 정렬의 점화식 (4.1)과 동일하다. 4.5절에서 살펴볼 마스터 방법을 사용하면 이 점화식의 해가 $T(n) = \Theta(n \lg n)$임을 알 수 있다. 또한 해가 $T(n) = \Theta(n \lg n)$임을 이해하기 위해 그림 2.5의 재귀 트리를 참고하는 것도 좋다.

이와 같이 분할정복 기법이 주먹구구식 방법보다 점근적으로 더 빠른 알고리즘을 만들어 낼 수 있음을 알게 되었다. 병합 정렬과 최대 부분 배열 문제를 통해 분할정복 기법이 얼마나 강력한 힘을 발휘할 수 있는지에 대한 느낌을 조금 갖게 된 것이다. 분할정복 기법은 종종 점근적으로 가장 빠르게 푸는 알고리즘을 만들어내지만, 그보다 더 빠른 알고리즘을 만들 수도 있다. 연습문제 4.1-5와 같이 최대 부분 배열 문제를 푸는 선형 시간 알고리즘이 존재하며 이 알고리즘은 분할정복 기법을 사용하지 않는다.

연습문제

4.1-1

A의 모든 원소가 음수일 때 FIND-MAXIMUM-SUBARRAY가 반환하는 값은 무엇인가?

4.1-2

최대 부분 배열 문제를 푸는 주먹구구식 방법에 대한 의사코드를 작성하라. 작성한 프로시저는 $\Theta(n^2)$ 시간에 수행되어야 한다.

4.1-3

당신의 컴퓨터를 사용해 최대 부분 배열 문제를 푸는 주먹구구식 알고리즘과 재귀 알고리즘을 둘 다 구현하라. 재귀 알고리즘이 주먹구구식 알고리즘을 넘어설 때의 교차 지점이 되는 문제의 크기 n_0은 무엇인가? 그리고 문제의 크기가 n_0보다 작을 때 주먹구구식 알고리즘을 사용하도록 재귀 알고리즘의 베이스 케이스를 수정하라. 이로 인해 교차 지점이 달라지는가?

4.1-4

최대 부분 배열 문제를 수정해 빈 부분 수열을 결과로 허용한다고 가정하자. 빈 부분 배열의 합은 0이다. 빈 부분 수열을 허용하지 않는 알고리즘이 빈 부분 배열을 결과로 허용하도록 하려면 어떻게 수정해야 할까?

15 동적 프로그래밍

동적 프로그래밍(dynamic programming)은 분할정복 기법과 같이 부분 문제의 해를 결합해 문제를 해결한다(여기서 "프로그래밍"은 컴퓨터 코드를 쓰는 것이 아니라 테이블을 이용한 방법을 일컫는다). 2장에서 보았듯이, 분할정복 알고리즘은 문제를 서로 겹치지 않는(disjoint) 부분 문제로 분할하고, 해당 부분 문제를 재귀적으로 해결한 후, 해결 결과를 결합하여 원래의 문제를 해결한다. 반면, 동적 프로그래밍은 부분 문제가 서로 중복될 때, 즉 부분 문제가 다시 자신의 부분 문제를 공유할 때 적용할 수 있다. 이 경우, 분할정복 알고리즘은 공유되는 부분 문제를 반복적으로 해결하여 일을 필요 이상으로 더 많이 하게 된다. 동적 프로그래밍 알고리즘을 이용하면 모든 부분 문제를 한 번만 풀어 그 해를 테이블에 저장함으로써 각 부분 문제를 풀 때마다 다시 계산하는 일을 피할 수 있다.

일반적으로 ***최적화 문제***(optimization problem)에 동적 프로그래밍을 적용한다. 이런 문제는 다양한 해를 가질 수 있다. 각 해는 값을 가져 이 중 최적(최소 또는 최대)의 값인 해를 찾기를 원한다. 이런 해를 그 문제에 대한 *유일한* 최적해라 하지 않고 *한 개*의 최적해라 한다. 최적의 값을 가지는 해가 여러 개 존재할 수 있기 때문이다. 동적 프로그래밍 알고리즘을 개발할 때는 다음 4단계를 따른다.

1. 최적해의 구조의 특징을 찾는다.
2. 최적해의 값을 재귀적으로 정의한다.
3. 최적해의 값을 일반적으로 상향식(bottom-up) 방법으로 계산한다.
4. 계산된 정보들로부터 최적해를 구성한다.

1–3단계는 주어진 한 문제에 대한 동적 프로그래밍 해의 기초가 된다. 4단계는 최적해 자체는 필요 없고 최적해의 값만 필요할 경우 생략할 수 있다. 4단계를 수행할 경우, 최적해를 쉽게 구성하기 위해 3단계에서 부가적인 정보를 유지하기도 한다.

이후 절에서는 동적 프로그래밍 방법을 이용해 몇 가지 최적화 문제를 풀어본다. 15.1절에서는 막대 하나를 더 작은 막대 여러 개로 나누어 총 가치가 최대가 되도록 하는

길이 i	1	2	3	4	5	6	7	8	9	10
가격 p_i	1	5	8	9	10	17	17	20	24	30

그림 15.1 막대 가격의 예. i 인치의 각 막대는 회사에 p_i 달러의 수익을 가져온다.

문제를 살펴본다. 15.2절에서는 일련의 행렬을 곱할 때 스칼라 곱의 전체 횟수를 최소화하는 곱 순서를 살펴본다. 동적 프로그래밍에 대한 이런 예가 주어졌을 때 어떤 문제에 동적 프로그래밍이 적용되기 위해 필요한 두 가지 주요한 특징에 대해 15.3절에서 논의한다. 그런 다음 15.4절에서는 두 개의 시퀀스에서 최장 공통 부분 시퀀스(Longest Common Subsequence)를 동적 프로그래밍을 이용해 찾는 방법을 보여준다. 마지막으로 15.5절에서는 검색할 키들의 분포가 주어졌을 때 동적 프로그래밍을 이용해 최적인 이진 검색 트리(binary search tree)를 만들어본다.

15.1 막대 자르기

첫 번째 예제에서는 동적 프로그래밍을 이용해 강철 막대의 자를 곳을 결정하는 간단한 문제를 해결한다. 강철 회사는 기다란 강철 막대를 사서 더 작게 나누어 판매한다. 강철 막대를 자르는 데 비용이 들지 않는다. 강철 회사의 경영진은 이익이 최대가 되도록 막대를 자르는 방법을 알고 싶어한다.

$i = 1, 2, \ldots$에 대해 강철회사는 길이 i 인치의 막대에 대해서 달러로 가격 p_i를 청구한다고 가정한다. 막대 길이는 언제나 단위가 인치다. 그림 15.1은 가격 표의 예다.

막대-자르기 문제는 다음과 같다. 길이가 n 인치의 막대와 $i = 1, 2, \ldots, n$에 대한 가격 p_i의 표가 주어지면, 해당 막대를 잘라서 판매하여 얻을 수 있는 최대 수익 r_n을 결정하라. 길이가 n인 막대의 가격 p_n이 충분히 비싸면 최적해는 자르지 않은 것일 수도 있음에 주목하라.

$n = 4$일 때를 고려해보자. 그림 15.2는 자르지 않는 것을 비롯해 길이가 4인 막대를 자르는 방법을 모두 보여준다. 4인치 막대를 2인치 두 개로 나누면 최적의 수익으로 $p_2 + p_2 = 5 + 5 = 10$이 발생함을 알 수 있다.

길이가 n인 막대는 2^{n-1}개의 다른 방법으로 나눌 수 있는데, 맨 왼쪽 끝부터 $i = 1, 2, \ldots, n-1$에 대한 모든 i 인치마다 자르거나 자르지 않거나 하는 독립적인 선택을 할 수 있기 때문이다.[1] 보통 덧셈 개념을 이용해 조각의 분해를 나타내, $7 = 2 + 2 + 3$

[1] 감소하지 않는 크기 순으로 잘라 조각을 만들어야 한다면 가능한 방법이 더 작을 수 있다. $n = 4$에 대해서는 그림 15.2의 (a), (b), (c), (e), (h)와 같은 5가지 방법만 고려하게 될 것이다. 가능한 방법의 개수

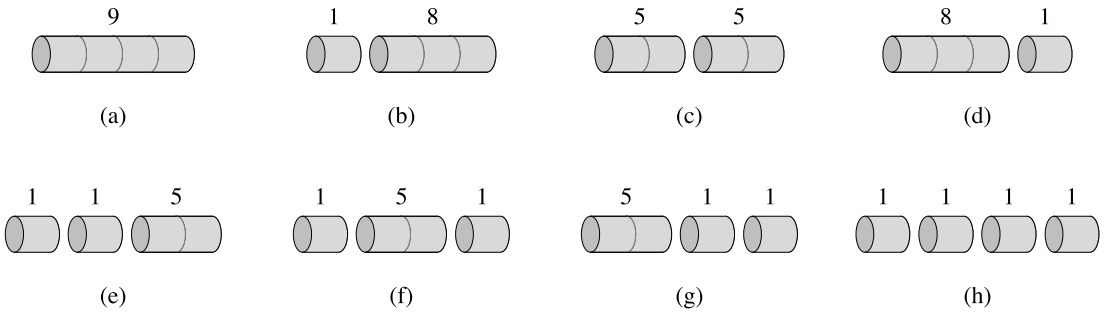


그림 15.2 길이가 4인 막대를 자르는 8가지 방법. 각 조각 위에는 그림 15.1의 가격표에 따른 가격을 표시했다. 가장 좋은 전략은 (c)와 같이 길이가 2가 되게 둘로 나누는 방법으로, 총 가격이 10이다.

은 길이가 7인 막대를 길이가 2인 두 조각과 길이가 3인 한 조각으로 나누었음을 나타낸다. 만약 최적해가 막대를 $1 \le k \le n$에 대해 k개의 조각으로 나누면, 막대를 조각 길이 $i_1, i_2, \ldots, i_k$로 나누는 다음의 최적의 분해는

$$n = i_1 + i_2 + \cdots + i_k$$

다음과 같은 최대의 해당 수익을 제공한다.

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

예제에서는 $i = 1, 2, \ldots, 10$에 대한 최적의 수익 r_i를 해당되는 최적의 분해를 이용한 면밀한 조사를 통해 결정할 수 있다.

r_1=1 $1 = 1$ (자르지 않음)
r_2=5 $2 = 2$ (자르지 않음)
r_3=8 $3 = 3$ (자르지 않음)
r_4=10 $4 = 2 + 2$분해로부터
r_5=13 $5 = 2 + 3$분해로부터
r_6=17 $6 = 6$ (자르지 않음)
r_7=18 $7 = 1 + 6$ 또는 $7 = 2 + 2 + 3$분해로부터
r_8=22 $8 = 2 + 6$분해로부터
r_9=25 $9 = 3 + 6$분해로부터
r_{10}=30 $10 = 10$ (자르지 않음)

는 ***파티션 함수***(partition function)라고 부르는데 이는 대략적으로 $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$과 같다. 이 양은 2^{n-1}보다 작지만 n에 대한 다항식보다는 훨씬 크다. 하지만 이런 방향으로의 논의는 더 이상 진행시키지 않을 것이다.

좀 더 일반적으로 말하면, $n \geq 1$에 대한 r_n값을 더 작은 막대로부터의 최대 수익을 이용해 나타낼 수 있다.

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right) \tag{15.1}$$

첫 번째 인자 p_n은 자르지 않고 길이가 n인 막대를 그대로 파는 것에 해당한다. max에 대한 나머지 $n-1$ 인자는 막대를 맨 처음에 $i = 1, 2, \ldots, n-1$에 대해 길이 i와 길이 $n-i$가 되게 둘로 나누고 두 조각을 더 최적으로 잘라 두 조각으로부터 r_i와 r_{n-i}의 수익을 얻는 것 중에서 얻어진 최대 수익에 해당한다. 그런데 어떤 i값이 수익을 최대화하는지 미리 알 수 없으므로 i에 대한 모든 가능한 값을 고려해서 수익을 최대로 하는 값을 뽑아야 한다. 막대를 자르지 않고 그대로 파는 것이 수익이 더 크다면 i값을 선택하지 않을 수도 있다.

크기가 n인 원래 문제를 풀기 위해 종류가 같은 더 작은 크기의 문제를 푼다. 첫 번째 자르기를 하고 나면 두 조각을 막대-자르기 문제의 독립적인 예로 생각해도 된다. 전체적인 최적해는 두 부분 문제의 각각에 대해 수익을 최대화하는 해를 이용한다. 막대 자르기 문제는 ***최적 부분 구조***(optimal substructure)를 가졌다고 한다. 그러므로 하나의 문제에 대한 최적해는 각각을 독립적으로 풀 수 있는 연관된 부분 문제들의 최적해를 이용한다.

막대 자르기 문제를 재귀 구조로 만드는 좀 더 간단한 방법은 맨 왼쪽 끝부터 길이가 i인 첫 번째 조각과 길이가 $n-i$인 오른쪽 나머지로 조각으로 나누는 것이다. 이때 첫 번째 조각을 제외한 나머지를 더 분해할 수 있다. 따라서 길이가 n인 막대의 모든 분해를 이런 식으로 첫 번째 조각과 나머지의 어떤 분해로 볼 수 있다. 그렇게 할 때는 막대를 전혀 자르지 않는 방법의 해는 첫 번째 조각이 $i = n$의 크기와 p_n의 수익을 가지고 나머지는 0의 크기와 $r_0 = 0$의 수익을 가지는 것으로 만들 수 있다. 그러므로 다음의 좀 더 간단한 식 (15.1)을 얻을 수 있다.

$$r_n = \max_{1 \leq i \leq n} \left(p_i + r_{n-i}\right) \tag{15.2}$$

이 공식에서 최적해는 두 개보다는 단 한 개(나머지 부분)의 관련된 부분 문제만 가지고 만든다.

하향식 재귀의 구현

다음 프로시저는 식 (15.2)가 함축하고 있는 계산을 단순한 하향식 재귀 방법으로 구현한 것이다.

```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

프로시저 CUT-ROD는 가격이 들어 있는 배열 $p[1..n]$과 정수 n을 입력 받아 길이가 n인 막대에 대해 가능한 최대 수익을 리턴한다. $n = 0$이면 수익이 없으므로 2행에서 CUT-ROD는 0을 리턴한다. 3행에서 최대 수익 q를 $-\infty$로 초기화하고, 4–5행에서 **for** 루프가 정확히 $q = \max_{1 \le i \le n}(p_i + \text{CUT-ROD}(p, n-i))$를 계산한다. 그런 다음 6행에서 이 값을 리턴한다. n에 대한 단순한 수학적 귀납법은 식 (15.2)를 이용해 이 답이 원하는 정답 r_n임을 증명할 수 있다.

좋아하는 프로그래밍 언어로 CUT-ROD를 작성하여 컴퓨터에서 수행해보면 입력 크기가 어느 정도 커지면 프로그램의 수행시간이 아주 오래 걸림을 발견할 것이다. $n = 40$이면 적어도 몇 분 이상, 많게는 한 시간 넘게 걸릴 가능성이 매우 높다. 사실 n을 1씩 증가시킬 때마다 프로그램 수행시간은 두 배가 될 것이다.

CUT-ROD가 매우 비효율적인 이유는 무엇일까? 문제는 CUT-ROD가 같은 인자를 가지고 자기 자신을 계속 재귀적으로 호출하는 데 있다. 이것은 같은 부분 문제를 반복해서 풀려고 한다. 그림 15.3은 $n = 4$일 때 어떤 일이 발생하는지를 보여준다. CUT-ROD(p, n)은 $i = 1, 2, \ldots, n$에 대해 CUT-ROD$(p, n-i)$를 호출한다. 같은 방식으로, CUT-ROD(p, n)은 각 $j = 0, 1, \ldots, n-1$에 대해 CUT-ROD(p, j)를 호출한다. 이 과정이 재귀적으로 전개되면 총 일의 양은 n에 대해 매우 폭발적으로 증가하는 함수로 표시된다.

CUT-ROD의 수행시간을 분석하기 위해 $T(n)$이 CUT-ROD의 두 번째 인자가 n일 때 호출된 총 횟수를 나타낸다고 하자. 이 표현은 재귀 트리에서 루트가 n으로 표시된 서브 트리에 있는 노드의 개수와 같다. 이 횟수는 그 루트에서의 최초 호출을 포함한다. 그러므로 $T(0) = 1$이고 $T(n)$은 다음과 같다.

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \tag{15.3}$$

여기서 1은 루트에서의 호출이고, $T(j)$는 $j = n - i$일 때 CUT-ROD$(p, n-i)$에 대한 호출에 의해 이루어지는(재귀 호출을 포함해) 호출 횟수다. 연습문제 15.1-1에서 다음과 같음을 보일텐데, 이와 같이 CUT-ROD의 총 수행시간은 n의 지수 함수에 비례한다.

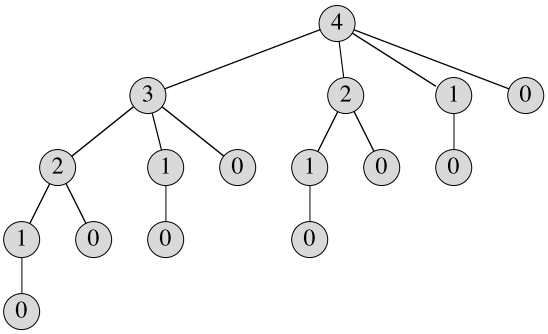


그림 15.3 $n = 4$에 대해 호출한 CUT-ROD(p, n)으로부터 발생하는 재귀 호출을 보여주는 재귀 트리. 각 노드 레이블은 해당 부분 문제의 크기 n을 표시하는데, 이는 레이블 s를 가진 부모로부터 레이블 t를 가진 자식 노드로의 간선은 최초로 $s - t$의 크기의 조각을 잘라내고 t의 크기의 나머지 부분 문제를 남기는 것에 해당한다. 루트로부터 리프 하나까지의 경로는 길이가 n인 막대를 나누는 2^{n-1}개 방법 중 하나에 해당한다. 일반적으로 재귀 트리는 2^n개의 노드와 2^{n-1}개의 리프를 가진다.

$$T(n) = 2^n \tag{15.4}$$

돌이켜 보면, 이 지수 함수에 비례하는 수행시간은 별로 놀랍지가 않다. CUT-ROD는 길이가 n인 막대를 자르는 데 가능한 2^{n-1} 방법을 명백하게 모두 고려한다. 재귀 트리는 리프가 2^{n-1}개고 각 리프는 막대를 자르는 가능한 방법 중 하나를 나타낸다. 루트로부터 리프 하나로의 단순 경로에 있는 레이블은 각 자르기를 하기 전에 남아 있는 각 오른쪽 조각의 크기를 알려준다. 다시 말해, 레이블은 막대의 오른쪽 끝부터 따질 때 자르는 지점에 해당한다.

동적 프로그래밍을 이용한 최적의 막대 자르기

이제 동적 프로그래밍을 이용해 CUT-ROD를 효율적인 알고리즘으로 바꾸는 방법을 보인다.

동적 프로그래밍 방법은 다음과 같이 동작한다. 초보적인 재귀해는 같은 부분 문제를 반복하여 풀어 비효율적임을 관찰했으므로, 각 부분 문제를 단 *한 번*만 풀고 그 해를 저장하도록 처리한다. 그러면 이 부분 문제의 해를 나중에 다시 구할 때 다시 계산하지 않고 저장된 값을 참조만 한다. 그래서 동적 프로그래밍은 계산 시간을 절약하기 위해 부가적인 메모리를 더 사용한다. 이는 ***시간-메모리 트레이드-오프***(time-memory trade-off)의 한 예로 작용한다. 지수 함수에 비례하는 시간의 해는 다항식에 비례하는 시간으로 변환될 수도 있듯이 극적으로 시간을 절약할 수 있다. 관련된 각각의 *다른* 부분 문제의 개수가 입력 크기에 대해 다항식에 비례하고 각 부분 문제를 다항식에 비례하는 시간에 풀 수 있다면, 동적 프로그래밍 방법은 다항식에 비례하는 시간에 수행된다.

동적 프로그래밍 방법을 구현하는 데 보통 두 가지 방법이 존재한다. 막대 자르기 예를 통해 두 가지 방법을 모두 설명하겠다.

첫 번째 방법은 ***메모하기를 이용한 하향식***이다.[2] 이 방법에서 프로시저를 자연스럽게 재귀적으로 쓰지만 각각의 부분 문제의 결과를 보통 배열이나 해시 테이블에 저장해놓도록 수정한다. 그리고 이 프로시저는 이 부분 문제를 이전에 풀었는지 알아보기 위해 확인을 한다. 이전에 풀었다면 이 레벨에서 이후에 계산하는 것을 절약하면서 그 저장된 결과를 리턴한다. 그리고 이전에 풀지 않았다면 이 프로시저는 통상적인 방법으로 값을 계산한다. 그러면 이 재귀 프로시저가 ***메모되었다***라고 말한다. 이 프로시저는 이전에 이미 계산된 결과를 "기억한다."

두 번째 방법은 ***상향식 방법***이다. 이 방법은 부분 문제의 "크기"라는 자연스러운 개념을 따르는데, 특정 부분 문제를 푸는 것이 "더 작은" 부분 문제를 푸는 것에만 의존한다. 부분 문제를 크기별로 정렬한 후 가장 작은 것을 첫 번째로 하여 크기 순으로 푼다. 특정 부분 문제를 풀 때는 그 해에 영향을 미치는 더 작은 부분 문제를 모두 풀어 그 해를 저장해놓는다. 그러므로 각 부분 문제를 한 번만 풀고 새로 등장하는 부분 문제에 필요한 모든 부분 문제는 이미 다 풀어놓은 상태가 된다.

이런 두 가지 방법은 하향식 방법이 실제로 모든 부분 문제를 조사하려고 재귀적으로 호출하지 않는 경우를 제외하고는, 똑같은 점근적인 수행시간을 가진 알고리즘을 만든다. 상향식 방법은 프로시저 호출에 대한 오버헤드가 더 작아 종종 훨씬 더 나은 상수 비율(constant factor)을 가진다.

다음은 메모하기가 더해진 하향식 CUT-ROD 프로시저의 의사코드다.

```
MEMOIZED-CUT-ROD(p, n)
1  r[0..n]을 새로운 배열이라 한다.
2  for i = 0 to n
3      r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

[2] 메모하기의 영어 표현은 *memorization*이 아니라 *memoization*인데 이것은 스펠링을 잘못 쓴 것이 아니다. *Memoization*의 어원은 *memo*인데, 이는 나중에 볼 수 있도록 값을 기록하는 것으로 이 기술이 만들어져 있기 때문이다.

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

메인 프로시저 MEMOIZED-CUT-ROD는 보조 배열 $r[0..n]$을 $-\infty$값으로 초기화하는데 여기서 $-\infty$는 "아직 모름"을 표시하는 편리한 하나의 선택이다(이미 계산된 수익 값은 언제나 음수가 아니다). 그러고 나서 헬퍼 프로시저인 MEMOIZED-CUT-ROD-AUX를 호출한다.

프로시저 MEMOIZED-CUT-ROD-AUX는 이전 프로시저 CUT-ROD에 메모하기가 더해진 버전이다. 맨 먼저 1행에서 원하는 값이 이미 계산되었는지 확인한다. 이미 계산되어 있으면 2행에서 해당 값을 리턴한다. 계산되어 있지 않으면 3–7행에서 원하는 값 q를 통상적인 방법으로 계산하고 8행에서 그 값을 $r[n]$에 저장한 후 9행에서 그 값을 리턴한다.

상향식 버전은 더 간단하다.

```
BOTTOM-UP-CUT-ROD(p, n)
1  r[0..n]을 새로운 배열이라 한다.
2  r[0] = 0
3  for j = 1 to n
4      q = −∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```

상향식 동적 프로그래밍 방법에서 BOTTOM-UP-CUT-ROD는 부분 문제의 자연스러운 순서를 사용한다. $i < j$면 크기 i의 부분 문제는 크기 j의 부분 문제보다 "작다." 그러므로 이 프로시저는 크기가 $j = 0, 1, \ldots, n$인 부분 문제를 크기 순으로 푼다.

프로시저 BOTTOM-UP-CUT-ROD의 1행은 새로운 배열 $r[0..n]$을 만드는데 여기에 부분 문제의 결과를 저장하고, 길이가 0인 막대는 수익이 없으므로 2행에서 $r[0]$을 0으

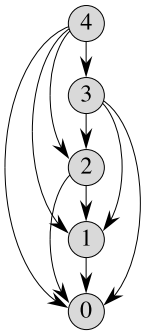


그림 15.4 $n = 4$일 때 막대 자르기 문제에 대한 부분 문제 그래프. 정점 레이블은 해당하는 부분 문제의 크기를 말한다. 방향성 간선(x, y)는 부분 문제 x를 풀 때 부분 문제 y에 대한 해가 필요함을 표시한다. 이 그래프는 그림 15.3에 있는 트리의 축소 버전으로, 같은 레이블의 모든 노드는 정점 하나로 합쳐지고 모든 간선이 부모에서 자식으로 간다.

로 초기화한다. 3–6행에서는 $j = 1, 2, \ldots, n$에 대해 크기 j의 증가 순으로 각 부분 문제를 푼다. 특정 크기 j의 문제를 풀기 위해 이용하는 방법은 6행에서 크기 $j - i$의 부분 문제를 풀기 위해 재귀 호출을 하는 대신 배열 엔트리 $r[j - i]$를 직접 참조하는 것만 제외하고 CUT-ROD에서 사용한 방법과 같다. 7행에서는 크기 j의 부분 문제에 대한 해를 $r[j]$에 저장한다. 마지막으로 8행에서 $r[n]$을 리턴하는데 이는 최적의 값 r_n이다.

상향식과 하향식 버전은 똑같은 점근적인 수행시간을 가진다. 프로시저 BOTTOM-UP-CUT-ROD의 수행시간은 이중으로 중첩된 루프 구조므로 $\Theta(n^2)$이다. 5–6행의 안쪽 **for** 루프의 반복 횟수는 등차 수열을 형성한다. 이에 해당하는 하향식 버전 MEMOIZED-CUT-ROD의 수행시간도 알아보기에 조금 어려울지는 모르지만 역시 $\Theta(n^2)$이다. 이전에 풀었던 부분 문제를 풀려고 하는 재귀 호출은 즉시 리턴하므로 MEMOIZED-CUT-ROD는 각각의 부분 문제를 한 번만 푼다. 이는 $0, 1, \ldots, n$의 크기에 대한 부분 문제를 푼다. 크기 n의 부분 문제를 풀기 위해 6–7행의 **for** 루프는 n번 반복한다. 그래서 MEMOIZED-CUT-ROD의 모든 재귀 호출에 대해 **for** 루프의 총 반복 횟수는 BOTTOM-UP-CUT-ROD의 안쪽 **for** 루프처럼 등차 수열을 형성하고 $\Theta(n^2)$이다 (사실 여기서 총계 분석(aggregate analysis)의 형식을 이용한다. 총계 분석은 17.1절에서 자세히 살펴볼 것이다).

부분 문제 그래프

동적 프로그래밍 문제에 대해 생각할 때는 관련된 부분 문제의 집합과 부분 문제가 서로 어떻게 연관되어 있는지를 이해해야 한다.

문제에 대한 ***부분 문제 그래프***는 이런 정보를 정확하게 나타내준다. 그림 15.4는 $n = 4$일 때 막대-자르기 문제에 대한 부분 문제 그래프를 보여준다. 이는 방향성 그래프고 각각의 다른 문제에 대해 정점 하나를 가진다. 부분 문제 x의 최적해를 결정하는

것이 부분 문제 y에 대한 최적해를 직접적으로 고려하면 부분 문제 그래프는 부분 문제 x에 대한 정점에서 부분 문제 y에 대한 정점으로 방향성 간선을 가진다. 예를 들어, 부분 문제 x에 대한 해를 구하는 하향식 재귀 프로시저가 부분 문제 y에 대한 해를 구하기 위해 자기 자신을 직접 호출하면, 부분 문제 그래프는 x에서 y로의 간선을 포함한다. 부분 문제 그래프를 하향식 재귀 방법에 대한 재귀 트리의 "축소된" 또는 "합쳐진" 버전으로 생각할 수 있는데, 여기서 같은 부분 문제에 대한 모든 정점을 정점 하나로 합치고 부모에서 자식으로의 모든 간선을 표시한다.

동적 프로그래밍의 상향식 방법은 주어진 부분 문제 x를 풀기 전에 부분 문제 x에 인접한 부분 문제 y를 먼저 푸는 순서로 부분 문제 그래프의 정점을 고려한다(B.4절에서 보았듯이 인접 관계(adjacency relation)는 꼭 대칭적이 아닐 수도 있다). 22장의 용어를 사용해 상향식 동적 프로그래밍 알고리즘에서 부분 문제 그래프의 정점을 "위상 정렬의 역순" 또는 "전치 그래프의 위상 정렬순"으로 고려한다(22.4절 참고). 다시 말해, 어떤 부분 문제든지 필요로 하는 모든 부분 문제들의 해를 구한 후 고려한다. 유사하게, 같은 관점에서 보면 동적 프로그래밍에 대한 (메모하기를 이용한) 하향식 방법을 부분 문제 그래프의 깊이-우선 검색으로 볼 수 있다(22.3절 참고).

부분 문제 그래프 $G = (V, E)$의 크기는 동적 프로그래밍 알고리즘의 수행시간 결정을 도와준다. 모든 부분 문제를 한 번만 풀기 때문에 수행시간은 각각의 부분 문제를 풀기 위해서 필요한 시간들의 합이다. 일반적으로 한 개의 부분 문제에 대한 해를 계산하기 위한 시간은 부분 문제 그래프에 있는 해당 정점의 차수(degree)(진출 간선 수)에 비례하고, 부분 문제의 개수는 부분 문제 그래프에 있는 정점 개수다. 이 경우에는 동적 프로그래밍의 수행시간이 정점과 간선의 개수에 선형으로 비례한다.

해의 재구성

막대 자르기 문제에 대한 동적 프로그래밍 해는 최적해의 값을 리턴하지만, 실제 조각의 크기 리스트는 리턴하지 않는다. 이 동적 프로그래밍 방법을 각각의 부분 문제에 대한 계산된 최적 *값*뿐만 아니라 그 최적 값에 이르는 *선택*도 기록하도록 확장할 수있다. 이런 정보와 함께 최적해를 쉽게 출력할 수있다.

각 막대의 크기 j에 대해 최대 수익 r_j뿐만 아니라 잘라내는 첫 번째 조각의 최적 크기 s_j까지 계산하는 BOTTOM-UP-CUT-ROD의 확장된 버전은 다음과 같다.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  r[0..n]과 s[1..n]은 새로운 배열이다.
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r과 s
```

이 프로시저는 1행에서 배열 s를 생성하고, 크기 j의 부분 문제의 해를 구할 때, 잘라낸 첫 번째 조각의 최적 크기 i를 보관하기 위해서 8행에서 $s[j]$를 변경하는 것만 제외하고는 BOTTOM-UP-CUT-ROD와 유사하다.

다음 프로시저는 가격표 p와 막대 크기 n을 입력으로 받아서 최적의 첫 번째 막대 크기들의 배열 $s[1..n]$을 계산하기 위해 EXTENDED-BOTTOM-UP-CUT-ROD를 호출한 후 길이 n의 막대의 최적 분할에서의 모든 조각 크기의 리스트를 출력한다.

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      s[n]을 출력한다.
4      n = n - s[n]
```

막대 자르기 예제에서 EXTENDED-BOTTOM-UP-CUT-ROD$(p, 10)$에 대한 호출은 다음의 배열을 리턴한다.

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

PRINT-CUT-ROD-SOLUTION$(p, 10)$에 대한 호출은 10을 출력하지만, $n = 7$과 함께 호출하면 앞에서 주어진 r_7에 대한 첫 번째 최적 분할에 해당하는 1과 6을 출력한다.

연습문제

15.1-1

식 (15.4)가 식 (15.3)과 초기 조건 $T(0) = 1$로부터 얻어짐을 보여라.

15.1-2

다음의 "그리디(greedy)" 전략이 막대를 자르는 데 항상 최적의 방법을 결정하지 못함을 반례(counterexample)를 통해 보여라. 길이가 i인 막대의 **밀도**를 p_i/i로, 즉 인치당 값으로 정의하라. 길이가 n인 막대에 대한 그리디 전략은 $1 \le i \le n$에 대해 최대 밀도를 갖는 길이가 i인 첫 번째 조각을 잘라낸다. 그런 다음 길이가 $n-i$인 나머지 조각에 그리디 전략을 계속 적용한다.

15.1-3

막대 자르기 문제를 변형하여 각 막대에 대한 가격 p_i 외의 각 자르기가 특정한 비용 c를 발생한다고 생각하자. 하나의 해에 대한 수익은 조각들의 가격 총 합에서 자르는 비용을 빼 주어야 한다. 이렇게 변형된 문제를 풀기 위한 동적 프로그래밍 알고리즘을 보여라.

15.1-4

값뿐만 아니라 실제 해도 함께 리턴하도록 MEMOIZED-CUT-ROD를 수정하라.

15.1-5

피보나치 수는 재귀식 (3.22)에 의해 정의된다. n번째 피보나치 수를 계산하기 위한 $O(n)$ 시간의 동적 프로그래밍 알고리즘을 만들라. 그리고 부분 문제 그래프를 그려라. 그래프에서 정점과 간선의 수는 몇 개인가?

15.2 행렬-체인 곱셈

다음에 소개할 동적 프로그래밍은 행렬 곱셈 문제를 해결하는 알고리즘이다. 행렬 곱셈을 위한 다음과 같은 n개의 행렬 체인(순서) $\langle A_1, A_2, \ldots, A_n \rangle$이 주어지고 행렬의 곱을 계산하려고 한다.

$$A_1 A_2 \cdots A_n \tag{15.5}$$

INTRODUCTION TO ALGORITHMS

THIRD EDITION

초판 때부터 전 세계 여러 대학에서 교재로 뿐만 아니라 전문가들의 표준 참고서로 활용되어 온 책의 세 번째 판이다. 매우 다양한 알고리즘을 다루면서도 상당히 심도 있게 설명하여 정밀함과 포괄성이라는 두 가지 측면을 균형 있게 만족시켜 준다. 또한 각 장이 독립적으로 완결된 형태를 갖춰 필요한 내용을 찾아 참고하기 편하고, 모든 알고리즘이 프로그래밍 경험이 있으면 누구라도 이해할 수 있는 의사코드로 작성되어 있어 이론과 실전이라는 두 마리 토끼를 함께 잡을 수 있다.

특히 개정 3판에서는 많은 변화를 통해 완성도가 한층 강화되었다. 먼저 반 엠데 보아스 트리와 멀티스레드를 다루는 장이 추가되고, 점화식이 분할정복 장으로 정비되었다. 그리고 동적 프로그래밍과 그리드 알고리즘에 개선된 방법이 추가되었고, 플로우 네트워크에도 새로운 개념이 도입되었다. 이외에도 전체 내용이 다듬어지고 갱신되었는데, 특히 연습문제와 종합문제에 더 다양한 응용 문제가 추가되었을 뿐만 아니라 이에 대한 모범답안이 웹 사이트를 통해 제공된다.

프로그래밍 일반 / 알고리즘

93000
9 791156 641131

정가 47,000원

ISBN 979-11-5664-113-1