algorithmic puzzles

anany levitin | maria levitin

ALGORITHMIC PUZZLES

Anany Levitin
and
Maria Levitin

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Oxford University Press, Inc., publishes works that further
Oxford University's objective of excellence
in research, scholarship, and education.

Oxford New York
Auckland Cape Town Dares Salaam Hong Kong Karachi
Kuala Lumpur Madrid Melbourne Mexico City Nairobi
New Delhi Shanghai Taipei Toronto

With offices in
Argentina Austria Brazil Chile Czech Republic France Greece
Guatemala Hungary Italy Japan Poland Portugal Singapore
South Korea Switzerland Thailand Turkey Ukraine Vietnam

Published by Oxford University Press, Inc.
198 Madison Avenue, New York, New York 10016
http://www.oup.com

Oxford is a registered trademark of Oxford University Press

Library of Congress Cataloging-in-Publication Data

Levitin, Anany.
Algorithmic puzzles / Anany Levitin, Maria Levitin.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-19-974044-4 (pbk.)
1. Mathematical recreations. 2. Algorithms.
I. Levitin, Maria. II. Title.

Contents

Preface in Questions and Answers

WHAT IS THIS BOOK ABOUT?

This book is a collection of algorithmic puzzles—puzzles that involve, explicitly or implicitly, clearly defined procedures for solving problems. It is a unique collection of such puzzles. The book includes some old classics, which have become a part of mathematics and computer science folklore. It also contains newer examples, some of which have been asked during job interviews at major companies.

The book has two main goals:

- To entertain a wide range of readers interested in puzzles
- To promote development of high-level algorithmic thinking (with no computer programming), supported by a carefully developed list of general algorithm design strategies and analysis techniques

Although algorithms do constitute the cornerstone of computer science and no sensible computer programming is possible without them, it is a common misconception to equate the two. Some algorithmic puzzles predate computers by more than a thousand years. It is true, however, that the proliferation of computers has made algorithmic problem solving important in many areas of modern life, from hard and soft sciences to art and entertainment. Solving algorithmic puzzles is the most productive and definitely most enjoyable way to develop and strengthen one's algorithmic thinking skills.

WHOM IS THIS BOOK FOR?

There are three large categories of readers who should be interested in this book:

- Puzzle lovers
- People interested in developing algorithmic thinking, including teachers and students
- People preparing for interviews with companies giving puzzles as well as people conducting such interviews

All we have to say to puzzle lovers is to reassure them that they could enjoy this collection as they would a collection not dedicated to any particular theme or type of puzzle. They will encounter a few all-time favorites, but, hopefully, will also find a number of little-known puzzle gems. No computing background or even an interest in it is assumed; such a reader can simply ignore references to specific algorithm design strategies and analysis techniques in the solutions given.

Algorithmic thinking has recently become somewhat of a buzz word among computer science educators, and with some justice: ubiquity of computers in today's world does make algorithmic thinking a very important skill for almost any student. Puzzles are an ideal vehicle for mastering this important skill for two reasons. First, puzzles are fun, and a person is normally willing to put more effort into solving them than in doing routine exercises. Second, algorithmic puzzles force a solver to think on a more abstract level. Even computer science students have a tendency to think about algorithmic problems in terms of a computer language they know instead of applying general design and analysis strategies. Puzzles can rectify this important deficiency.

The puzzles in this book can certainly be used for individual study. Together with the tutorials, they provide, in our view, a good introduction to main algorithmic ideas. They can also be used by teachers of computing courses—both at the college and secondary school level—as supplemental exercises and project topics. The book might also be of interest for problem-solving courses, especially those based on puzzles.

As to people preparing for interviews, they should find the book helpful in two ways. First, it contains many examples of puzzles

they may encounter, with complete solutions and comments. Second, the book also provides concise tutorials on algorithm design strategies and analysis techniques. After all, managers offering puzzles during interviews claim that they are more interested in the way an interviewee approaches a puzzle than in an actual solution to it. Showing an expertise in applying general design strategies and analysis techniques should then be a highly attractive way to impress the potential employer.

WHAT PUZZLES ARE INCLUDED IN THE BOOK?

Algorithmic puzzles constitute a small fraction among thousands of mathematical puzzles invented over the years. In selecting puzzles for the book, we have sought puzzles that satisfy the following criteria.

First, we wanted puzzles that illustrate some general principle in design or analysis of algorithms.

Second, we were looking for beauty and elegance, the subjectivity of those qualities notwithstanding.

Third, we wanted the puzzles to run a wide range of difficulty levels. Puzzle difficulty is hard to pinpoint; math professors have been occasionally stumped by puzzles easily solved by middle school students. Still, we have divided the book's puzzles into three sections—Easier Puzzles, Puzzles of Medium Difficulty, and Harder Puzzles—to give readers some help in gauging the puzzles' difficulty. Within each of these three sections, we have tried to order the puzzles in increasing level of difficulty as well. The puzzles in the Easier Puzzles section require only middle school mathematics. Although solutions to a few problems in the other two sections do use proofs by induction, high school mathematics should, in general, suffice for solving all the book's puzzles. In addition, topics such as binary numbers and simple recurrence relations are briefly reviewed in the second tutorial. This does not mean, of course, that all the puzzles in the book are easy. Some of them—especially those at the end of the last section—are truly

hard. But their difficulty does not lie in some sophisticated mathematics, and the reader should not be intimidated by them.

Fourth, we have felt compelled to include a few puzzles because of their historical importance. Finally, we only included puzzles with clear statements and solutions devoid of any tricks such as intentional ambiguity, word play, and so on.

One more important comment needs to be made here. Many puzzles in this book can be solved by exhaustive search or backtracking. (These strategies are explained in the book's first tutorial.) It is *not* the approach the reader is expected to employ to solve the puzzles, unless explicitly stated otherwise. Therefore, we have excluded categories of puzzles such as Sudoku and cryptarithms, which have to be solved either by exhaustive search/backtracking or by some ingenious insight in the specific data given in the puzzle. We have also decided against inclusion of puzzles based on some physical objects that are not very easy to describe, such as the Chinese Rings and Rubik's Cube.

HINTS, SOLUTIONS, AND COMMENTS

The book contains hints, solutions, and comments for every puzzle. Puzzle books rarely include hints, but we see them as a valuable addition. Hints may provide a small push in a right direction, still leaving the reader with a chance to solve the puzzle. All the hints are collected at the end of the book in a separate section.

Solutions are provided to every puzzle. As a rule, they start with a short answer. This is done to provide the reader with the last opportunity to solve the puzzle on his or her own: if the reader's answer disagrees with the one in the book, the reader can stop reading the complete solution and try to solve the puzzle again.

Algorithms are described in free-style English, with no special formatting or pseudocode notations. The emphasis is on the ideas rather than insignificant details. Rewriting the solutions in a more formal manner may, in fact, provide useful exercises in their own right.

Most comments point out a general algorithmic idea that the puzzle and its solution illustrate. Occasionally, they also include references to similar puzzles in the book and elsewhere.

Many puzzle books do not indicate the puzzle sources. The reason usually given is that trying to find an author of a puzzle is akin to trying to find an author of a joke. While there is a lot of truth to this observation, we have decided to mention the earliest sources of the puzzles known to us. The reader should keep in mind, however, that we have not conducted anything close to an extensive search for the puzzles' origin; doing that would have resulted in a very different book.

WHAT ARE THE TWO TUTORIALS ABOUT?

The book includes two tutorials, with puzzle examples, on general strategies for algorithm design and techniques for algorithm analysis. Although almost all puzzles in the book can be solved without any knowledge of the topics discussed in these tutorials, there is no question that they can make solving the puzzles much easier and, importantly, more useful. Besides, solutions, comments, and a few hints use some special terminology explained in the tutorials.

The tutorials are written on the most elementary level possible to make them comprehensible for a wide variety of readers. A reader with a computer science degree will hardly find there anything new, except, possibly, for puzzle examples. At the same time, such a reader might use them as a concise refresher of the fundamental ideas in the design and analysis of algorithms.

WHY ARE THERE TWO INDICES IN THE BOOK?

In addition to a standard index, the book contains an index indicating puzzles based on a particular design strategy or a type of analysis. This index should help the reader to locate problems on a specific strategy or technique and can also serve as a list of additional hints.

We conclude with a hope that the readers will find the book both enjoyable and useful. We also hope they will share our delight in beauty of and amazing feats of human ingenuity behind many of the book's puzzles.

Anany Levitin
Maria Levitin
May 2011
algorithmicpuzzles.book@gmail.com

Acknowledgments

We would like to express our deep gratitude to the book's reviewers: Tim Chartier (Davidson College), Stephen Lucas (James Madison University), and Laura Taalman (James Madison University). Their enthusiastic support of the book's idea and specific suggestions about its contents have certainly been very helpful to us.

We are also thankful to Simon Berkovich of the George Washington University for several discussions of the puzzle topics and for reading a portion of the book's manuscript.

Our thanks go to all the people at the Oxford University Press and their associates who worked on the book. We are especially grateful to our editor, Phyllis Cohen, for her ceaseless efforts to make the book better. We are also thankful to the editorial assistant, Hallie Stebbins, the book's cover designer, Natalya Balnova, and the marketing manager, Michelle Kelly. We appreciate the work of Richard Camp, the book's copy editor, as well as the efforts of Jennifer Kowing and Kiran Kumar who supervised the book's production.

List of Puzzles

TUTORIAL PUZZLES

The list below contains all the puzzles in the book's two tutorials. The puzzles are listed in the order of their appearance. The page numbers indicate locations of the puzzles; their solutions are given directly in the tutorials following the puzzle statements.

MAIN SECTION PUZZLES

This list contains all 150 puzzles included in the main section of the book. The page numbers indicate locations of the puzzle, hint, and solution with comments, respectively.

148. **One Coin for Freedom** 69, 81, 234

149. **Pebble Spreading** 69, 81, 236

150. **Bulgarian Solitaire** 70, 81, 238

THE EPIGRAPH PUZZLE: WHO SAID WHAT?

Match the following quotations with the authors listed below:

The man with a hammer sees every problem as a nail. Our age's great hammer is the algorithm.

Solving problems is a practical skill like, let us say, swimming. We acquire any practical skill by imitation and practice.

There is no better way to relieve the tedium than by injecting recreational topics into a course, topics strongly tinged with elements of play, humor, beauty, and surprise.

It is not knowledge, but the act of learning, not possession but the act of getting there, which grants the greatest enjoyment.

If I have perchance omitted anything more or less proper or necessary, I beg indulgence, since there is no one who is blameless and utterly provident in all things.

William Poundstone, the author of *How Would You Move Mount Fuji? Microsoft's Cult of the Puzzle: How the World's Smartest Companies Select the Most Creative Thinkers*

George Pólya (1887–1985), a prominent Hungarian mathematician, the author of *How To Solve It,* the classic book on problem solving

Martin Gardner (1914–2010), an American writer, best known for his "Mathematical Games" column in *Scientific American* and books on recreational mathematics

Carl Friedrich Gauss (1777–1855), a great German mathematician

Leonardo of Pisa a.k.a. Fibonacci (1170–c1250), a remarkable Italian mathematician, the author of *Liber Abaci* ("The Book of Calculation"), one of the most consequential mathematical book in history

1 Tutorials

GENERAL STRATEGIES FOR ALGORITHM DESIGN

The purpose of this tutorial is to briefly review a few general strategies for designing algorithms. While these strategies are not all applicable to every puzzle, taken collectively they provide a powerful tool kit. Not surprisingly, these strategies are also used for solving many problems in computer science. Therefore, learning to apply these strategies to puzzles can serve as an excellent introduction to this important field.

But before we embark on reviewing major algorithm design strategies, we need to make an important comment on two types of algorithmic puzzles. Every algorithmic puzzle has an input. An input defines an *instance* of the puzzle. The instance can be either specific (e.g., find a false coin among eight coins with a balance) or general (e.g., find a false coin among n coins with a balance). When dealing with a specific instance of a puzzle, the solver has no obligations beyond solving the instance given. In fact, it might be the case that other instances of the puzzle do not have the same solution or even do not have solutions at all. On the other hand, specific numbers in a puzzle's statement may be of no significance whatsoever. Then solving the general instance of the puzzle could be not only more satisfying but, on occasion, even easier. But whether a puzzle is presented by a specific instance or given in its most general form, it is almost always a good idea to solve a few small instances of it anyway. On rare occasions the solver might be misled by such investigation, but much more often it can provide useful insights into the puzzle given.

Exhaustive Search

Theoretically, many puzzles can be solved by *exhaustive search*—a problem-solving strategy that simply tries all possible

candidate solutions until a solution to the problem is found. Little ingenuity is typically required in applying exhaustive search. Therefore, puzzles are rarely offered to a person (as opposed to a computer) in the expectation that a solution will be found by applying this strategy. The most important limitation of exhaustive search is its inefficiency: as a rule, the number of solution candidates that need to be processed grows at least exponentially with the problem size, making the approach inappropriate not only for a human but often for a computer as well. As an example, consider the problem of constructing a *magic square* of order 3.

Magic Square Fill the 3 × 3 table with nine distinct integers from 1 to 9 so that the sum of the numbers in each row, column, and corner-to-corner diagonal is the same (Figure 1.1).

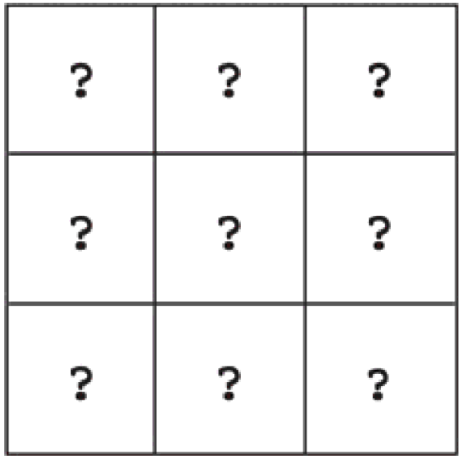


FIGURE 1.1 3 × 3 table to be filled with integers 1 through 9 to form a magic square.

How many ways are there to fill such a table? Let us think of the table as filled with one number at a time, starting with placing the 1 somewhere and ending with placing the 9. There are nine ways to place 1, followed by eight ways to place 2, and so on until the last number 9 is placed in the only unoccupied cell of the table. Hence, there are $9! = 9 \cdot 8 \cdot \ldots \cdot 1 = 362{,}880$ ways to arrange the

nine numbers in the cells of the 3 × 3 table. (We just used the standard notation, $n!$, called *n factorial*, for the product of consecutive integers from 1 to n.) Therefore, solving this problem by exhaustive search would imply generating all 362,880 possible arrangements of distinct integers from 1 to 9 in the table and checking, for each of the arrangements, whether all its row, column, and diagonal sums are the same. This amount of work is clearly impossible to do by hand.

Actually, it is not difficult to solve this puzzle by proving first that the value of the common sum is equal to 15 and that 5 must be put at the center cell (see the *Magic Square Revisited* puzzle (#29) in the main section of the book). Alternatively, one can take advantage of several known algorithms for constructing magic squares of an arbitrary order $n \geq 3$, which are especially efficient for odd n's (e.g., [Pic02]). Of course, these algorithms are not based on exhaustive search: the number of candidate solutions the exhaustive search algorithm would have to consider becomes prohibitively large even for a computer for n as small as 5. Indeed, $(5^2)! \simeq 1.5 \cdot 10^{25}$, and hence it would take a computer making 10 trillion operations per second about 49,000 years to finish the job.

Backtracking

There are two major difficulties in applying exhaustive search. The first one lies in the mechanics of generating all possible solution candidates. For some problems, such candidates compose a well-structured set. For example, candidate arrangements of the first nine positive integers in the cells of the 3 × 3 table (see the *Magic Square* example above) can be obtained as *permutations* of these numbers, for which several algorithms are known. There are many problems, however, where solution candidates do not form a set with such a regular structure. The second, and more fundamental, difficulty lies in the number of solution candidates that need to be generated and processed. Typically, the size of this set grows at least exponentially with the problem size. Therefore, exhaustive search is practical only for very small instances of such problems.

Backtracking is an important improvement over the brute-force approach of exhaustive search. It provides a convenient method for generating candidate solutions while making it possible to avoid generating unnecessary candidates. The main idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows: If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for *any* remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with the next option for that component.

Typically, backtracking involves undoing a number of wrong choices—the smaller this number, the faster the algorithm finds a solution. Although in the worst-case scenario a backtracking algorithm may end up generating all the same candidate solutions as an exhaustive search, this rarely happens.

It is convenient to interpret a backtracking algorithm as a process of constructing a tree that mirrors decisions being made. Computer scientists use the term *tree* to describe hierarchical structures such as family trees and organizational charts. A tree is usually shown with its *root* (the only node without a parent) on the top and its *leaves* (nodes without children) on or closer to the bottom of the diagram. This is nothing but a convenient typographical convention, however. For a backtracking algorithm, such a tree is called a *state-space tree*. The root of a state-space tree corresponds to the start of a solution construction process; we consider the root to be on the zero level of the tree. The root's children—on the first level of the tree—correspond to possible choices of the first component of a solution (e.g., the cell to contain 1 in the magic square construction). Their children—the nodes on the second level—correspond to possible choices of the second component of a solution, and so on. Leaves can be of two kinds. The first kind—called *nonpromising nodes* or *dead ends*—correspond to partially constructed candidates that cannot

lead to a solution. After establishing that a particular node is nonpromising, a backtracking algorithm terminates the node (the tree is said to be *pruned*), undoes the decision regarding the last component of the candidate solution by backtracking to the parent of the nonpromising node, and considers another choice for that component. The second kind of a leaf provides a solution to the problem. If a single solution suffices, the algorithm stops; if other solutions need to be searched for, the algorithm continues searching for them by backtracking to the leaf's parent.

The following example is a perennial favorite for showing an application of backtracking to a particular problem.

The n-Queens Problem Place n queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same column, row, or diagonal.

For $n = 1$, the problem has a trivial solution, and it is easy to see that there is no solution for $n = 2$ and $n = 3$. So let us consider the 4-queens problem and solve it by backtracking. Since each of the four queens has to be placed in its own column, all we need to do is to assign a row for each queen on the board shown in Figure 1.2.

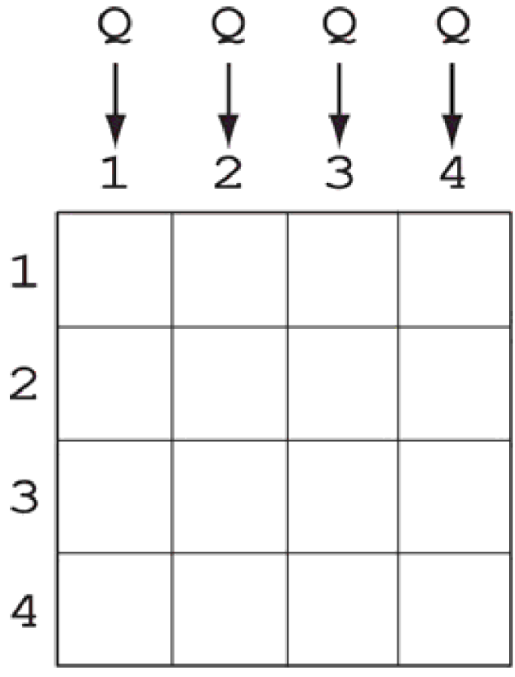


FIGURE 1.2 Board for the 4-queens problem.

We start with the empty board and then place queen 1 in the first possible position, which is in row 1 of column 1. Then we place queen 2, after trying unsuccessfully rows 1 and 2 of the second column, in the first acceptable position for it, which is square (3, 2), the square in row 3 and column 2. This proves to be a dead end because there is no acceptable position in the third column for queen 3. Therefore, the algorithm backtracks and puts queen 2 in the next possible position (4, 2). Then queen 3 is placed at (2, 3), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (2, 1). Queen 2 then goes to (4, 2), queen 3 to (1, 3), and queen 4 to (3, 4), which is a solution to the problem. The state-space tree of this search is given in Figure 1.3.

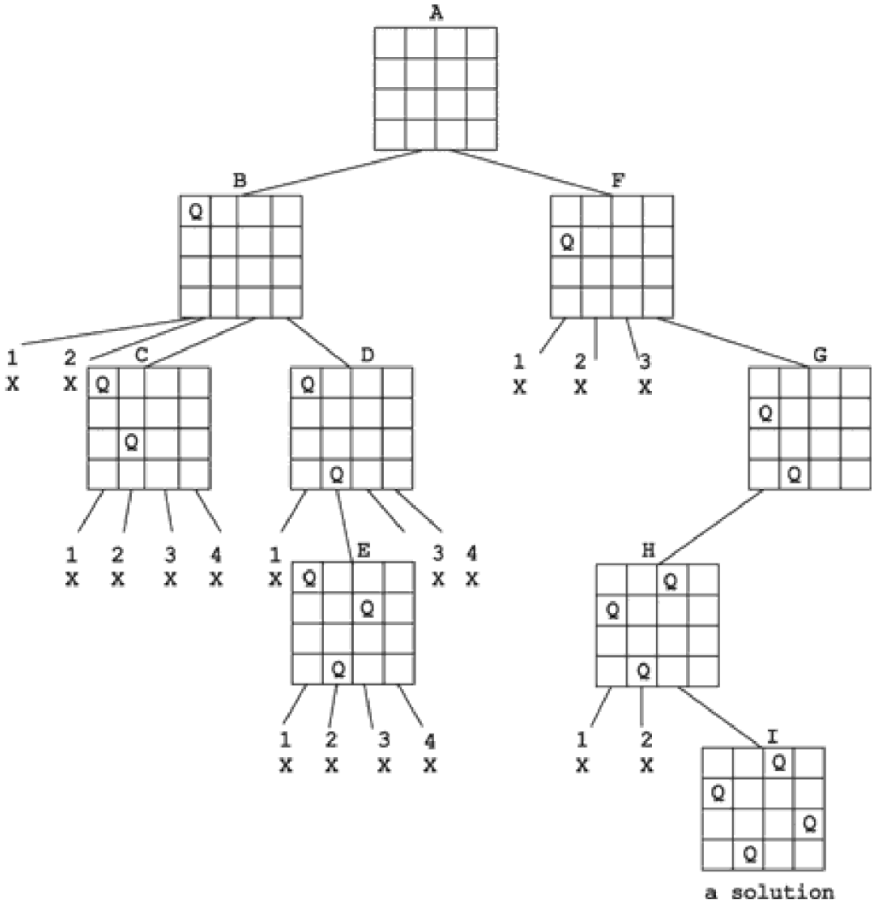


FIGURE 1.3 State-space tree of finding a solution to the 4-queens problem by backtracking. X denotes an unsuccessful attempt to place a queen in the indicated row. The letters above the nodes show the order in which the nodes are generated.

If other solutions need to be found (there is just one other solution to the 4-queens problem), the algorithm can simply resume its operations at the leaf at which it stopped. Alternatively, one can use the board's symmetry for this purpose.

How much faster is this solution by backtracking compared to exhaustive search? If we are to consider all possible placements of

four queens on four different squares of the 4 × 4 board, the number of such placements is

$$\frac{16!}{4!(16-4)!} = \frac{16 \cdot 15 \cdot 14 \cdot 13}{4 \cdot 3 \cdot 2} = 1820.$$

(The general formula for the number of ways to choose k different objects, the order of which is not of interest, from a given set of n different objects, called by mathematicians *combinations* of n objects taken k at a time and denoted by either $\binom{n}{k}$ or $C(n, k)$, is $\frac{n!}{k!(n-k)!}$.) If we consider only the placements with the queens in different columns, the total number of solution candidates decreases to $4^4 = 256$. And if we add to the latter the constraint that the queens must also be in different rows, the number of choices drops to 4! = 24. While the last number is quite manageable, it would not be the case for larger instances of the problem. For example, for a regular 8 × 8 chessboard, the number of such solution candidates is 8! = 40,320.

The reader might be interested to know that the total number of different solutions to the 8-queens problem is 92, twelve of which are qualitatively distinct, with the remaining 80 obtainable from the basic twelve by rotations and reflections. As to the general n-queens problem, it has a solution for every $n \geq 4$ but no convenient formula for the number of solutions for an arbitrary n has been discovered. It is known that this number grows very fast with the value of n. For example, the number of solutions for $n = 10$ is 724, of which 92 are distinct, while for $n = 12$ the respective numbers are 14,200 and 1787.

Many puzzles in this book can be solved by backtracking. For each of them, however, there is a more efficient algorithm the reader is expected to strive for. In particular, *The n-Queens Problem Revisited* (#140) in the main section of the book asks the reader to design a much faster algorithm for the n-queens problem.

Decrease-and-Conquer

The *decrease-and-conquer* strategy is based on finding a relationship between a solution to a given problem and a solution to its smaller instance. Once found, such a relationship leads naturally to a *recursive algorithm*, which reduces the problem to a sequence of its diminishing instances until it becomes small enough to be solved directly.[1] Here is an example.

Celebrity Problem A celebrity among a group of n people is a person who knows nobody but is known by everybody else. The task is to identify a celebrity by only asking questions to people of the following form: "Do you know this person?"

Assuming for simplicity that a celebrity is known to exist among a given group of n people, the problem can be solved by the following decrease-by-one algorithm. If $n = 1$, that one person is vacuously a celebrity by the definition. If $n > 1$, select two people from the group, say, A and B, and ask A whether he or she knows B. If A knows B, remove A from the remaining people who can be a celebrity; if A does not know B, remove B from this group. Then solve the problem recursively (i.e., by the same method) for the remaining group of $n - 1$ people who can be a celebrity.

As an easy exercise, the reader may want to solve the *Ferrying Soldiers* puzzle (#4) in the main section of the book.

In general, a smaller instance in the decrease-and-conquer paradigm need not necessarily be of size $n - 1$. Although *decrease-by-one* is the most common case of size reduction, there are examples of size reduction by a larger amount. We get a particularly fast algorithm if we manage reducing an instance size by a constant factor, for example, by half, on each iteration. A well-known example of such an algorithm arises in the following well-known game.

Number Guessing (*Twenty Questions*) Determine a selected integer in the range from 1to n, inclusive, by asking questions with yes/no answers.

The fastest algorithm for this problem asks a question that reduces the size of the set containing the answer by about half on each iteration. For example, the first question can be whether the selected number is greater than $\lceil n/2 \rceil$, which is the standard notation for $n/2$ rounded up to the nearest integer.[2] If the answer is "no," the selected number is among the integers 1 to $\lceil n/2 \rceil$; if the answer is "yes," the selected number is among the integers $\lceil n/2 \rceil + 1$ to n. In either case, the algorithm reduced the problem of size n to an instance of the same problem of about half the size of the original instance. Repeating this step until the instance size is reduced to 1 solves the problem.

Since this algorithm reduces the size of an instance (the range of the numbers that still can contain the selected number) by about half on each iteration, it works amazingly fast. For example, for $n = 1{,}000{,}000$, the algorithm requires no more than 20 questions! As fast as it is, an algorithm would be even faster if it could reduce the instance size by a larger factor, say, 3.

A Fake Among Eight Coins (#10) in the main section of the book provides another illustration of the *decrease-by-constant-factor* variation of the decrease-and-conquer strategy and can serve as a good exercise here.

It should be noted that sometimes it is easier to exploit a relationship between larger and smaller instances bottom up. This means solving first the puzzle for the smallest possible instance, then for the next larger one, and so on. This method is sometimes called the *incremental approach*. For a specific example, see the first solution of the *Rectangle Dissection* puzzle (#3) in the book's main section.

Divide-and-Conquer

The *divide-and-conquer* strategy is to partition a problem into several smaller subproblems (usually of the same or related type and ideally of about the same size), solve each of them, and then, if necessary, combine their solutions to get a solution to the original

problem. This strategy underlines many efficient algorithms for important problems in computer science. Surprisingly, there are not many puzzles solvable by divide-and-conquer algorithms. Here is a well-known example, however, that perfectly illustrates this strategy.

Tromino Puzzle Cover a $2^n \times 2^n$ board missing one square with right trominoes, which are L-shaped tiles formed by three adjacent squares. The missing square can be any of the board squares. Trominoes should cover all the squares except the missing one exactly with no overlaps.

The problem can be solved by a recursive divide-and-conquer algorithm that places a tromino at the center of the board in such a way that the problem's instance of size n is reduced to four instances of the same problem, each of size $n - 1$ (Figure 1.4). The algorithm stops after every 2×2 region with one missing square generated by it is covered with a single tromino.

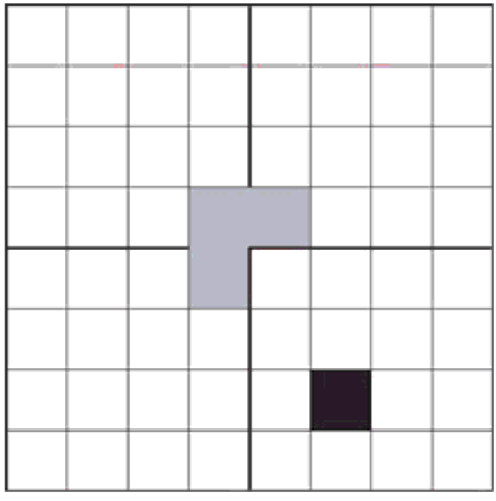


FIGURE 1.4 First step in tromino tiling of a $2^n \times 2^n$ board without one square by a divide-and-conquer algorithm.

The reader may want to finish the tiling of the 8×8 square in Figure 1.4 by this algorithm as a quick but useful exercise.

Most divide-and-conquer algorithms solve smaller subproblems recursively because, as in the above example, they represent smaller instances of the same problem. This need not always be the case, however. For some problems involving boards, in particular, a board may need to be divided into subboards that are not necessarily smaller versions of the board given. For such examples, see *2n-Counters Problem* (#37) and *Straight Tromino Tiling* (#78) in the main section of the book.

One more comment needs to be made about the divide-and-conquer strategy. Although some people consider decrease-and-conquer (discussed above) as a special case of divide-and-conquer, it is better to consider it as distinct design strategy. The crucial difference between the two lies in the number of smaller subproblems that need to be solved on each step: several in divide-and-conquer algorithms, and just one in decrease-and-conquer algorithms.

Transform-and-Conquer

The *transform-and-conquer* is a well-known approach to problem solving that is based on the idea of transformation. A problem is solved in two stages. First, in the transformation stage, it is modified or transformed into another problem that, for one reason or another, is more amenable to solution. Then, in the second, conquering stage, it is solved. In our realm of algorithmic problem solving, one can identify three varieties of this strategy. The first variety—called *instance simplification*—solves a problem by first transforming an instance given into another instance of the same problem with some special property that makes the problem easier to solve. The second variety—called *representation change*—is based on the transformation of a problem's input to a different representation that is more conducive to an efficient algorithmic solution. The third variety of the transformation strategy is *problem reduction*, in which an instance of a given problem is transformed into an instance of a different problem altogether.

As our first example, let us consider a puzzle-like problem from Jon Bentley's book *Programming Pearls* [Ben00, pp. 15–16].

Anagram Detection Anagrams are words that are composed of the same letters; for example, the words "eat," "ate," and "tea" are anagrams. Devise an algorithm to find all sets of anagrams in a large file of English words.

An efficient algorithm for this problem works in two stages. First, it assigns each word a "signature" obtained by sorting its letters (representation change) and then sorts the file in alphabetical order of the signatures (sorting data is a special case of instance simplification) to put anagrams next to each other.

As an exercise, the reader is invited to solve the *Number Placement* puzzle (#43), which exploits the same idea.

Another occasionally useful type of representation change is to employ binary or ternary representation of the problem's input. Just in case the reader is unfamiliar with this important topic, here is a one-paragraph introduction. In the decimal positional system, which most of the world has been using for the last eight hundred years, an integer is represented as a combination of powers of 10, for example, $1069 = 1 \cdot 10^3 + 0 \cdot 10^2 + 6 \cdot 10^1 + 9 \cdot 10^0$. In the binary and ternary systems, an integer is represented as a combination of powers of 2 and 3, respectively. For example, $1069_{10} = 10000101101_2$ because $1069 = 1 \cdot 2^{10} + 0 \cdot 2^9 + 0 \cdot 2^8 + 0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$, and $1069_{10} = 1110121_3$ because $1069 = 1 \cdot 3^6 + 1 \cdot 3^5 + 1 \cdot 3^4 + 0 \cdot 3^3 + 1 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0$. While a decimal number is composed of some of 10 digits (0 through 9), there are only two possible digits in a binary number (0 and 1), and there are three possible digits (0, 1, and 2) in a ternary number. Every decimal integer has a unique representation in either of these systems, which can be found by repeatedly dividing the integer by 2 and 3, respectively. The binary system is particularly important because it has proved to be most convenient for computer implementation.

As an example of a puzzle that takes advantage of the binary system, consider an instance of the problem mentioned in W. Poundstone's book [Pou03, p. 84].

Cash Envelopes You have one thousand $1 bills. How can you distribute them among 10 envelops so that any amount between $1 and $1000, inclusive, can be given as some combination of these envelopes? No change is allowed, of course.

Let us put 1, 2, 2^2,... , 2^8 dollar bills in the first nine envelopes and $1000 - (1 + 2 + \cdots + 2^8) = 489$ dollar bills in the tenth envelope. Any amount A smaller than 489 can be obtained as a combination of the powers of 2: $b_8 \cdot 2^8 + b_7 \cdot 2^7 + \cdots + b_0 \cdot 1$, where the coefficients b_8, b_7,... , b_0 are either 0 or 1. (These coefficients compose A's representation in the binary system. The largest integer a nine digit binary number can represent is $2^8 + 2^7 + \cdots + 1 = 2^9 - 1 = 511$.) Any amount A between 489 and 1000, inclusive, can be represented as $489 + A'$ where $0 \le A' \le 511$; hence, it can be obtained as the contents of the tenth envelope and a combination of the first nine, the latter given by the binary representation of A'. Note that the solution to the puzzle is not unique for some amounts A.

A good exercise for the reader would be to solve the two versions of the *Bachet's Weights* puzzle (#115), which take advantage of the binary and a variation of the ternary system, respectively.

Finally, many problems can be solved by transforming them into questions about graphs. A *graph* can be thought of as a finite collection of points in the plane with lines connecting some of them. The points and lines are called, respectively, *vertices* and *edges* of the graph. Edges may have no directions on them or may be directed from one vertex to another. In the former case, the graph is said to be *undirected*; in the later case, it is called a *directed graph*, or a *digraph,* for short. In applications to puzzles and games, vertices of a graph typically represent possible states of

the problem in question, and edges indicate permitted transitions between the states. One of the graph's vertices represents an initial state, while another represents a goal state of the problem. (There might be several vertices of the latter kind.) Such a graph is called a *state-space graph.* Thus, the transformation just described reduces the problem to the question about a path from the initial-state vertex to a goal-state vertex.

As a specific example, let us consider a smaller instance of a very old and well-known puzzle.[3]

Two Jealous Husbands There are two married couples who need to cross a river. They have a boat that can hold no more than two people at a time. To complicate matters, both husbands are jealous and require that no wife can be in the presence of the other man without her husband being present. Can they cross the river under such constraints?

A state-space graph for this puzzle is shown in Figure 1.5, where H_i, W_i denote the husband and wife of couple i ($i = 1, 2$), respectively; the two bars || denote the river; the boat's location, which defines the direction of the next trip, is shown by the gray oval. (For the sake of simplicity, the graph does not include crossings that differ by obvious index substitutions such as starting with the first couple H_1W_1 crossing the river instead of the second couple H_2 W_2.) The vertices corresponding to the initial and final states are shown in bold.

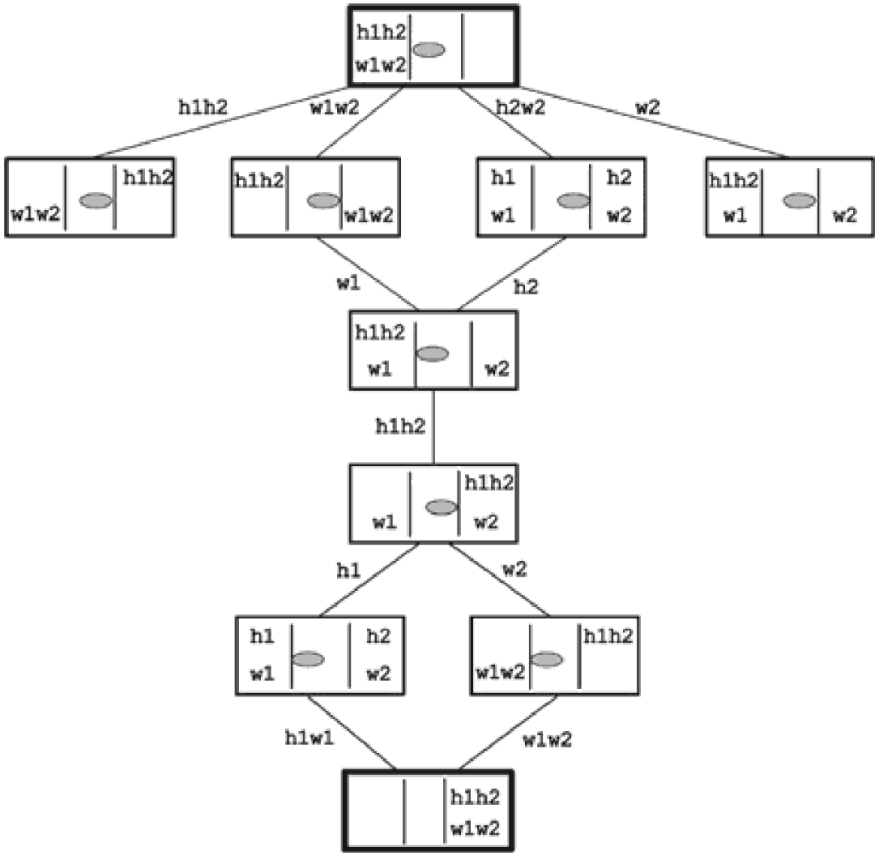


FIGURE 1.5 State-space tree for the *Two Jealous Husbands* puzzle.

There are four shortest paths from the initial-state vertex to the final-state vertex, each five edges long, in this graph. Specified by their edges, they are as follows:

$$
\begin{array}{lllll}
W_1W_2 & W_1 & H_1H_2 & H_1 & H_1W_1 \\
W_1W_2 & W_1 & H_1H_2 & W_2 & W_1W_2 \\
H_2W_2 & H_2 & H_1H_2 & H_1 & H_1W_1 \\
H_2W_2 & H_2 & H_1H_2 & W_2 & W_1W_2
\end{array}
$$

Hence, there are four (to within obvious symmetric substitutions) optimal solutions to this problem, each requiring five river crossings.

The *Missionaries and Cannibals* puzzle (#49) can be used as another exercise of this kind.

Two notes need to be made about solving puzzles via a graph representation. First, the creation of a state-space graph for more sophisticated puzzles can pose an algorithmic problem in its own right. In fact, the task might be infeasible because of a very large number of states and transformations. For example, the graph representing the states of the Rubik's Cube puzzle would have more than 10^{19} vertices. Second, although a specific location of points representing vertices of a graph has no theoretical significance, a good selection of the way the vertices are placed in the plane can provide an important insight into the puzzle in question. For example, consider the following puzzle, which is often attributed to Paolo Guarini (1512) but in fact was found in Arab chess manuscripts dating from around 840.

Guarini's Puzzle There are four knights on the 3×3 chessboard: the two white knights are at the two bottom corners, and the two black knights are at the two upper corners of the board (Figure 1.6). The goal is to switch the knights in the minimum number of moves so that the white knights are at the upper corners and the black knights are at the bottom corners.

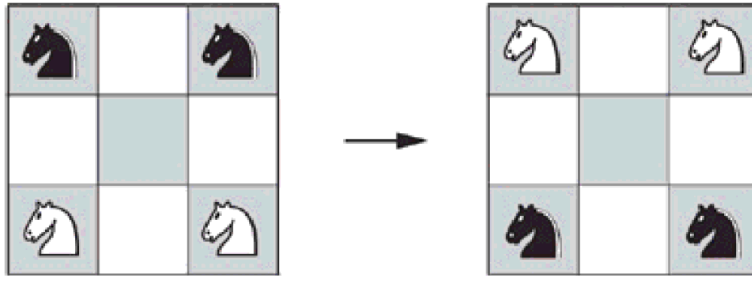


FIGURE 1.6 *Guarini's Puzzle*.

It is natural to represent the squares of the board (numbered for simplicity by consecutive integers in Figure 1.7a) by vertices of a graph in which an edge connects two vertices if a knight can make a move between the squares represented by the vertices. Placing the vertices in a way mimicking the positions of the squares on the board, we obtain the graph shown in Figure 1.7b. (Since the square number 5 at the center of the board cannot be reached by any of the knights, it is omitted there.) The graph in Figure 1.7b does not seem to help much in solving the problem. If, on the other hand, we place the vertices along a circumference in the order they can be reached from vertex 1 by knight moves—as shown in Figure 1.7c—we will obtain a much more revealing picture.[4] It is clear from Figure 1.7c that every legitimate move of a knight preserves the knights' relative ordering in the clockwise and counterclockwise directions. Therefore, there are only two ways to solve the puzzle in the minimum number of moves: move the knights along the edges in either a clockwise or counterclockwise direction until each of the knights reaches the diagonally opposite corner for the first time. Either of these symmetric options requires a total of 16 moves.

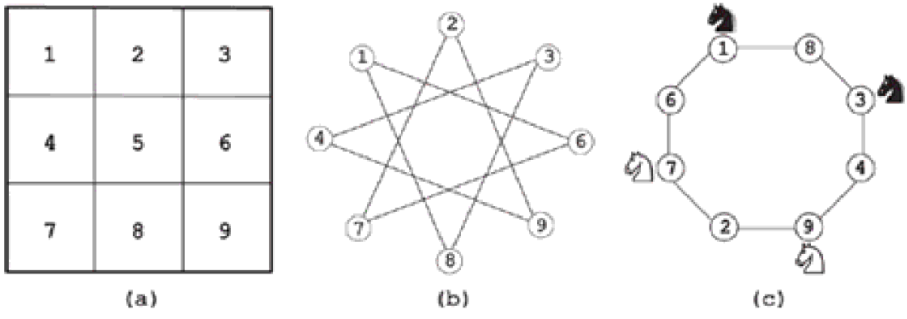


FIGURE 1.7 (a) Numbering of the board's squares for *Guarini's Puzzle*. (b) Straightforward representation of the puzzle's graph. (c) Better representation of the puzzle's graph.

We recommend solving the *Coins on a Star* puzzle (#34) as an exercise in the graph unfolding method.

There are also puzzles that can be solved by reducing them to a mathematical problem such as solving an equation or finding the maximum or minimum of a function. Here is an example of such a puzzle.

Optimal Pie Cutting What is the maximum number of pieces one can get by making n straight cuts in a rectangular pie, if each cut has to be parallel to one of the pie's sides, vertical or horizontal?

If the pie is cut by h horizontal and v vertical cuts, the total number of pieces obtained will be $(h + 1)(v + 1)$. Since the total number of cuts $h + v$ is equal to n, the problem is reduced to maximizing

$$(h + 1)(v + 1) = hv + (h + v) + 1 = hv + n + 1 = h(n - h) + n + 1$$

among all integer values of h between 0 and n, inclusive. Since $h(n - h)$ is a quadratic function of h, the maximum is obtained for $h = n/2$ if n is even and for $h = n/2$ rounded down (denoted $\lfloor n/2 \rfloor$) or up (denoted $\lceil n/2 \rceil$) if n is odd. Hence, the puzzle has a unique solution $h = v = n/2$ if n is even and two solutions (which can be considered symmetric) $h = \lfloor n/2 \rfloor$, $v = \lceil n/2 \rceil$ and $h = \lceil n/2 \rceil$, $v = \lfloor n/2 \rfloor$ if n is odd.

Greedy Approach

The *greedy approach* solves an optimization problem by a sequence of steps, each expanding a partially constructed solution until a complete solution is reached.

At each step—and this is the central point of this strategy—the choice is to produce the largest immediate gain without violating the problem's constraints. Such a "greedy" grab of the best alternative available at each step is made in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem. This simple-minded approach works in some cases and fails in others.

One should not expect a rich bounty from a hunt for puzzles solvable by the greedy approach: good puzzles are usually too "tricky" to be solvable in such a straightforward fashion. Still, there are puzzles that can be solved by a greedy algorithm. Usually in these cases it is not difficult to design a greedy algorithm itself; rather, a more difficult task is to prove that it indeed yields an optimal solution. The following puzzle provides an example.

Non-Attacking Kings Place the greatest possible number of kings on an 8×8 chessboard so that no two kings are placed on adjacent—vertically, horizontally, or diagonally—squares.

Following the prescription of the greedy strategy, we can start by placing the maximum number of nonadjacent kings (four) in the first column of the board. Then, after skipping the next column because each of its squares is adjacent to one of the placed kings in the first column, we can place four kings in the third column, skip the fourth, and so on, until we end up with the total of 16 kings on the board (Figure 1.8a).

In order to show that it is impossible to place more than 16 nonadjacent kings on the board, we partition the board into 16 four-by-four squares, as shown in Figure 1.8b. Obviously, it is impossible to place more than one king in each of these squares, which implies that the total number of nonadjacent kings on the board cannot exceed 16.

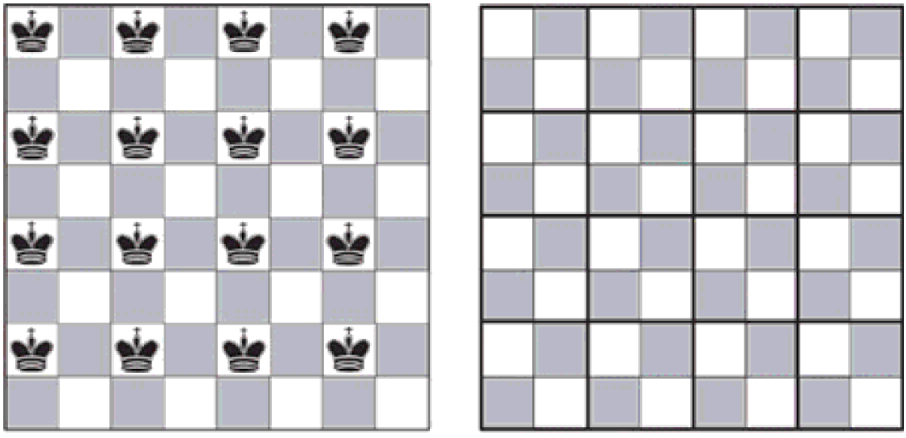


FIGURE 1.8 (a) Placement of 16 non-attacking kings. (b) Partition of the board proving impossibility of placing more than 16 non-attacking kings.

As our second example, we consider a puzzle that has become especially popular since it was reported to have been asked during job interviews at Microsoft.

Bridge Crossing at Night A group of four people, who have one flashlight, need to cross a rickety bridge at night. A maximum of two people can cross the bridge at one time, and any party that crosses (either one or two people) must have the flashlight with them. The flashlight must be walked back and forth; it cannot be thrown. Person A takes 1 minute to cross the bridge, person B takes 2 minutes, person C takes 5 minutes, and person D takes 10 minutes. A pair must walk together at the rate of the slower person's pace. Find the fastest way they can accomplish this task.

The greedy algorithm, illustrated in Figure 1.9, would start by sending to the other side the two fastest people, that is, persons A and B (it will take 2 minutes) and then return the flashlight with the fastest of the two, that is, with person A (1 more minute). Then it will send to the other side the two fastest persons available, that is, persons A and C (5 minutes) and return the flashlight with the

fastest person A (1 minute). Finally, the two persons remaining will cross the river together (10 minutes). The total amount of time this greedy-based schedule requires is (2 + 1) + (5 + 1) + 10 = 19 minutes, but this is *not* the fastest possible solution (see this puzzle again later in the book (#7)).

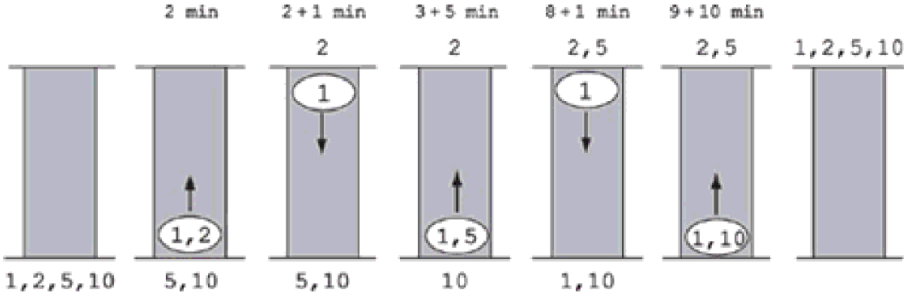


FIGURE 1.9 Greedy solution to the *Bridge Crossing at Night* puzzle.

It would be instructive for the reader to revisit the *Coins On a Star* puzzle (#34) and solve it by the greedy approach without benefits of the graph's unfolding.

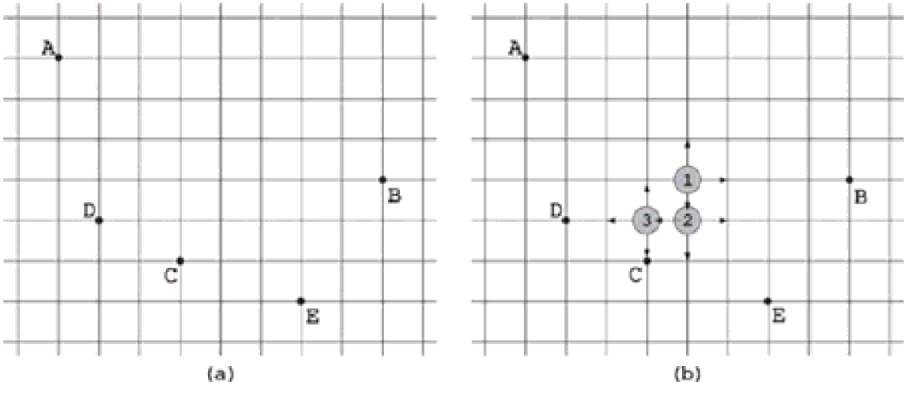
Iterative Improvement

While a greedy algorithm constructs a solution piece by piece, an iterative improvement algorithm starts with some easily obtainable approximation to a solution and improves upon it by repeated applications of some simple step. To validate such an algorithm, one needs to make sure that the algorithm in question does stop after a finite number of steps and that the final approximation obtained indeed solves the problem. Consider the following puzzle, which is a politically correct version of a problem discussed by Martin Gardner in his remarkable book *aha!Insight* [Gar78, pp. 131–132].

Lemonade Stand Placement Five friends—Alex, Brenda, Cathy, Dan, and Earl—want to set up a lemonade stand. They live at the locations denoted by letters A, B, C, D, and E on the map in Figure 1.10a. At which street intersection should they place their stand to

minimize the distance to their homes? Assume that they measure the distance by the total number of blocks—horizontally and vertically—from their homes to the stand.

Initially, the friends decided to locate their stand at intersection 1 (Figure 1.10b), which is the middle point horizontally between the leftmost and rightmost points A and B and the middle point vertically between the highest and lowest points A and E. But then somebody noticed that it is not the best location possible. So they decided on the following iterative improvement algorithm: Starting with their initial candidate, consider in turn the locations one block from it in some order, say, up (north), right (east), down (south), and left (west). As soon as a new location is closer to their homes, replace the old location with the new candidate and repeat the same verification operation; if none of the four neighboring intersections turns out to be better, consider the current location optimal and stop the algorithm. The algorithm's operation is shown in Figure 1.10b, with the computed distances given in Figure 1.10c.



	①	↑①	①→	↓① = ②
A	4 + 3	4 + 2	5 + 3	4 + 4
B	4 + 0	4 + 1	3 + 0	4 + 1
C	1 + 2	1 + 3	2 + 2	1 + 1
D	3 + 1	3 + 2	4 + 1	3 + 0
E	2 + 3	2 + 4	1 + 3	2 + 2
Total	23	26	24	**22**

	↑② = ①	②→	↓②	←② = ③
A		5 + 4	4 + 5	3 + 4
B		3 + 1	4 + 2	5 + 1
C		2 + 1	1 + 0	0 + 1
D		4 + 0	3 + 1	2 + 0
E		1 + 2	2 + 1	3 + 2
Total		23	23	**21**

	↑③	③→ = ②	↓③	←③
A	3 + 3		3 + 5	2 + 4
B	5 + 0		5 + 2	6 + 1
C	0 + 2		0 + 0	1 + 1
D	2 + 1		2 + 1	1 + 0
E	3 + 3		3 + 1	4 + 2
Total	22		22	22

(c)

FIGURE 1.10 (a) Instance of the *Lemonade Stand Placement* puzzle. (b) The algorithm's steps. (c) Distances computed by the algorithm.

While the final location marked by number 3 in Figure 1.10b certainly looks like a good choice, the algorithm does not provide a proof of the location's *global* optimality. In other words, how do we know that not only the four intersections one block away from it are inferior choices but also that it will be true for any other intersection? Well, we need not worry about our young entrepreneurs: this location is indeed the best, and the reader will have a chance to see this by solving the *Site Selection* puzzle (#74)—the general instance of this puzzle.

Here is another example of a puzzle that can be solved by iterative improvement.

Positive Changes Given an $m \times n$ table of real numbers, is there an algorithm to make all the row sums and column sums nonnegative by changing the signs of all the numbers in any row or column as the only operation allowed for the algorithm?

It would be natural to try finding an algorithm that increases the number of lines (rows and columns) with nonnegative sums on each of its iterations. However, changing the signs in a row (column) with a negative sum may make the sums in some column (row) negative! A neat way to overcome this difficulty is to pay attention to the total sum of the numbers in the table. Since it can be computed as the total of either all the row sums or all the column sums, changing the signs in a line with a negative sum definitely increases the total sum of the numbers in the table. Therefore, we can simply repeatedly search for a line with a negative sum. If we find such a line, we change the signs of all its numbers; if we do not find such a line, we have achieved our goal and can stop.

Is that all? Not quite. We also need to show that the algorithm's operation cannot continue indefinitely without stopping. This is indeed the case, because repeated applications of the algorithm's operation can create only a finite number of different tables (each of the mn elements can be in no more than two states). Therefore, the number of all element sums is also finite. Since the algorithm generates a sequence of tables with increasing sums, it must stop after a finite number of steps.

In both examples considered above, we took advantage of some quantity with the following characteristics:

- It could change its value only in a desired direction (decreasing in the first problem and increasing in the second).
- It could attain only a finite number of values, which guaranteed a stop after a finite number of steps.
- When it reached its final value, the problem was solved.

Such a quantity is called a *monovariant*. Finding an appropriate monovariant can be a tricky task. This has made puzzles involving monovariants a popular topic in mathematical competitions. For example, the second example given above was used among practice problems for the first All-Russian Mathematical Olympiad

in 1961 [Win04, p. 77]. It would be wrong, however, to dismiss iterative improvement and monovariants as just mathematical toys. Some of the most important algorithms in computer science, such as the *simplex method*, are based on this approach. The interested reader can find a few other puzzles involving monovariants in the harder puzzle section of this book.

Dynamic Programming

Dynamic programming is interpreted by computer scientists as a technique for solving problems with overlapping subproblems. Rather than solving overlapping subproblems again and again, it suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained. Dynamic programming was invented by a prominent U.S. mathematician, Richard Bellman, in the 1950s as a general method for optimizing multistage decision processes. For an optimization problem to be solved by this technique, the problem must have a so-called optimal substructure so that its optimal solution can be constructed efficiently from optimal solutions to its subproblems.

As an example, consider a problem of counting shortest paths.

Shortest Path Counting Find the number of the shortest paths from intersection A to intersection B in a city with perfectly horizontal streets and vertical avenues shown in Figure 1.11a.

Let $P[i, j]$ be the number of shortest paths from intersection A to the intersection of street i ($1 \leq i \leq 4$) and avenue j ($1 \leq j \leq 5$). Any shortest path here is composed of horizontal segments going right along the streets and vertical segments going down the avenues. Therefore, the number of shortest paths from A to the intersection of street i and avenue j can be found as the sum of the number of shortest paths from A to the intersection of street $i - 1$ and avenue j ($P[i - 1, j]$ in our notation) and the number of shortest paths from