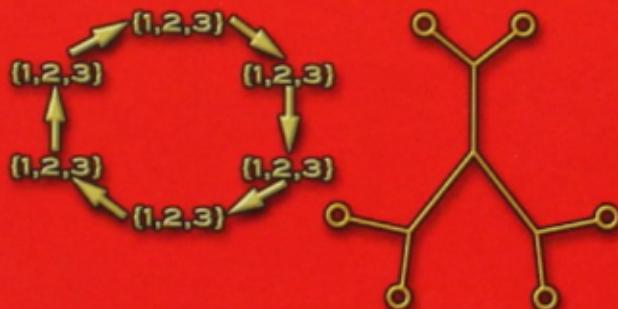
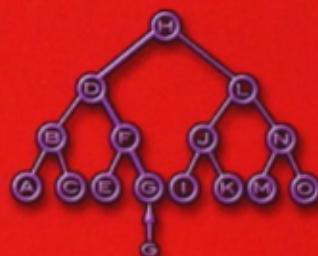
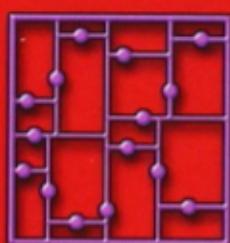
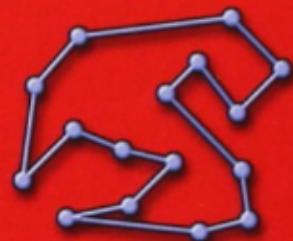


Second Edition

THE Algorithm Design MANUAL



XYZXYZ\$
YZXYZ\$
ZXYZ\$
XYZ\$
YZ\$
Z\$
\$



Steven S. Skiena

 Springer

Steven S. Skiena

The Algorithm Design Manual

Second Edition



Springer

Steven S. Skiena
Department of Computer Science
State University of New York
at Stony Brook
New York, USA
skiena@cs.sunysb.edu

ISBN: 978-1-84800-069-8 e-ISBN: 978-1-84800-070-4
DOI: 10.1007/978-1-84800-070-4

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2008931136

© Springer-Verlag London Limited 2008

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licenses issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.

The use of registered names, trademarks, etc., in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Printed on acid-free paper

Springer Science+Business Media
springer.com

The Algorithm Design Manual

Second Edition

Preface

Most professional programmers that I've encountered are not well prepared to tackle algorithm design problems. This is a pity, because the techniques of algorithm design form one of the core practical *technologies* of computer science. Designing correct, efficient, and implementable algorithms for real-world problems requires access to two distinct bodies of knowledge:

- *Techniques* – Good algorithm designers understand several fundamental algorithm design techniques, including data structures, dynamic programming, depth-first search, backtracking, and heuristics. Perhaps the single most important design technique is *modeling*, the art of abstracting a messy real-world application into a clean problem suitable for algorithmic attack.
- *Resources* – Good algorithm designers stand on the shoulders of giants. Rather than laboring from scratch to produce a new algorithm for every task, they can figure out what is known about a particular problem. Rather than re-implementing popular algorithms from scratch, they seek existing implementations to serve as a starting point. They are familiar with many classic algorithmic problems, which provide sufficient source material to model most any application.

This book is intended as a manual on algorithm design, providing access to combinatorial algorithm technology for both students and computer professionals. It is divided into two parts: *Techniques* and *Resources*. The former is a general guide to techniques for the design and analysis of computer algorithms. The Resources section is intended for browsing and reference, and comprises the catalog of algorithmic resources, implementations, and an extensive bibliography.

To the Reader

I have been gratified by the warm reception the first edition of *The Algorithm Design Manual* has received since its initial publication in 1997. It has been recognized as a unique guide to using algorithmic techniques to solve problems that often arise in practice. But much has changed in the world since the *The Algorithm Design Manual* was first published over ten years ago. Indeed, if we date the origins of modern algorithm design and analysis to about 1970, then roughly 30% of modern algorithmic history has happened since the first coming of *The Algorithm Design Manual*.

Three aspects of *The Algorithm Design Manual* have been particularly beloved: (1) the catalog of algorithmic problems, (2) the war stories, and (3) the electronic component of the book. These features have been preserved and strengthened in this edition:

- *The Catalog of Algorithmic Problems* – Since finding out what is known about an algorithmic problem can be a difficult task, I provide a catalog of the 75 most important problems arising in practice. By browsing through this catalog, the student or practitioner can quickly identify what their problem is called, what is known about it, and how they should proceed to solve it. To aid in problem identification, we include a pair of “before” and “after” pictures for each problem, illustrating the required input and output specifications. One perceptive reviewer called my book “The Hitchhiker’s Guide to Algorithms” on the strength of this catalog.

The catalog is *the* most important part of this book. To update the catalog for this edition, I have solicited feedback from the world’s leading experts on each associated problem. Particular attention has been paid to updating the discussion of available software implementations for each problem.

- *War Stories* – In practice, algorithm problems do not arise at the beginning of a large project. Rather, they typically arise as subproblems when it becomes clear that the programmer does not know how to proceed or that the current solution is inadequate.

To provide a better perspective on how algorithm problems arise in the real world, we include a collection of “war stories,” or tales from our experience with real problems. The moral of these stories is that algorithm design and analysis is not just theory, but an important tool to be pulled out and used as needed.

This edition retains all the original war stories (with updates as appropriate) plus additional new war stories covering external sorting, graph algorithms, simulated annealing, and other topics.

- *Electronic Component* – Since the practical person is usually looking for a program more than an algorithm, we provide pointers to solid implementations whenever they are available. We have collected these implementations

at one central website site (<http://www.cs.sunysb.edu/~algorith>) for easy retrieval. We have been the number one “Algorithm” site on Google pretty much since the initial publication of the book.

Further, we provide recommendations to make it easier to identify the correct code for the job. With these implementations available, the critical issue in algorithm design becomes properly modeling your application, more so than becoming intimate with the details of the actual algorithm. This focus permeates the entire book.

Equally important is what we do not do in this book. We do not stress the mathematical analysis of algorithms, leaving most of the analysis as informal arguments. You will not find a single theorem anywhere in this book. When more details are needed, the reader should study the cited programs or references. The goal of this manual is to get you going in the right direction as quickly as possible.

To the Instructor

This book covers enough material for a standard *Introduction to Algorithms* course. We assume the reader has completed the equivalent of a second programming course, typically titled *Data Structures* or *Computer Science II*.

A full set of lecture slides for teaching this course is available online at <http://www.algorist.com>. Further, I make available online audio and video lectures using these slides to teach a full-semester algorithm course. Let me help teach your course, by the magic of the Internet!

This book stresses design over analysis. It is suitable for both traditional lecture courses and the new “active learning” method, where the professor does not lecture but instead guides student groups to solve real problems. The “war stories” provide an appropriate introduction to the active learning method.

I have made several pedagogical improvements throughout the book. Textbook-oriented features include:

- *More Leisurely Discussion* – The tutorial material in the first part of the book has been *doubled* over the previous edition. The pages have been devoted to more thorough and careful exposition of fundamental material, instead of adding more specialized topics.
- *False Starts* – Algorithms textbooks generally present important algorithms as a fait accompli, obscuring the ideas involved in designing them and the subtle reasons why other approaches fail. The war stories illustrate such development on certain applied problems, but I have expanded such coverage into classical algorithm design material as well.
- *Stop and Think* – Here I illustrate my thought process as I solve a topic-specific homework problem—false starts and all. I have interspersed such

problem blocks throughout the text to increase the problem-solving activity of my readers. Answers appear immediately following each problem.

- *More and Improved Homework Problems* – This edition of *The Algorithm Design Manual* has twice as many homework exercises as the previous one. Exercises that proved confusing or ambiguous have been improved or replaced. Degree of difficulty ratings (from 1 to 10) have been assigned to all problems.
- *Self-Motivating Exam Design* – In my algorithms courses, I promise the students that *all* midterm and final exam questions will be taken directly from homework problems in this book. This provides a “student-motivated exam,” so students know exactly how to study to do well on the exam. I have carefully picked the quantity, variety, and difficulty of homework exercises to make this work; ensuring there are neither too few or too many candidate problems.
- *Take-Home Lessons* – Highlighted “take-home” lesson boxes scattered throughout the text emphasize the big-picture concepts to be gained from the chapter.
- *Links to Programming Challenge Problems* – Each chapter’s exercises will contain links to 3-5 relevant “Programming Challenge” problems from <http://www.programming-challenges.com>. These can be used to add a programming component to paper-and-pencil algorithms courses.
- *More Code, Less Pseudo-code* – More algorithms in this book appear as code (written in C) instead of pseudo-code. I believe the concreteness and reliability of actual tested implementations provides a big win over less formal presentations for simple algorithms. Full implementations are available for study at <http://www.algorist.com>.
- *Chapter Notes* – Each tutorial chapter concludes with a brief notes section, pointing readers to primary sources and additional references.

Acknowledgments

Updating a book dedication after ten years focuses attention on the effects of time. Since the first edition, Renee has become my wife and then the mother of our two children, Bonnie and Abby. My father has left this world, but Mom and my brothers Len and Rob remain a vital presence in my life. I dedicate this book to my family, new and old, here and departed.

I would like to thank several people for their concrete contributions to this new edition. Andrew Gaun and Betson Thomas helped in many ways, particularly in building the infrastructure for the new <http://www.cs.sunysb.edu/~algorithmand> dealing with a variety of manuscript preparation issues. David Gries offered valuable feedback well beyond the call of duty. Himanshu Gupta and Bin Tang bravely

taught courses using a manuscript version of this edition. Thanks also to my Springer-Verlag editors, Wayne Wheeler and Allan Wyldé.

A select group of algorithmic sages reviewed sections of the Hitchhiker's guide, sharing their wisdom and alerting me to new developments. Thanks to:

Ami Amir, Herve Bronnimann, Bernard Chazelle, Chris Chu, Scott Cotton, Yefim Dinitz, Komei Fukuda, Michael Goodrich, Lenny Heath, Cihat Imamoglu, Tao Jiang, David Karger, Giuseppe Liotta, Albert Mao, Silvano Martello, Catherine McGeoch, Kurt Mehlhorn, Scott A. Mitchell, Naceur Meskini, Gene Myers, Gonzalo Navarro, Stephen North, Joe O'Rourke, Mike Paterson, Theo Pavlidis, Seth Pettie, Michel Pocchiola, Bart Preneel, Tomasz Radzik, Edward Reingold, Frank Ruskey, Peter Sanders, Joao Setubal, Jonathan Shewchuk, Robert Skeel, Jens Stoye, Torsten Suel, Bruce Watson, and Uri Zwick.

Several exercises were originated by colleagues or inspired by other texts. Reconstructing the original sources years later can be challenging, but credits for each problem (to the best of my recollection) appear on the website.

It would be rude not to thank important contributors to the original edition. Ricky Bradley and Dario Vlah built up the substantial infrastructure required for the WWW site in a logical and extensible manner. Zhong Li drew most of the catalog figures using xfig. Richard Crandall, Ron Danielson, Takis Metaxas, Dave Miller, Giri Narasimhan, and Joe Zachary all reviewed preliminary versions of the first edition; their thoughtful feedback helped to shape what you see here.

Much of what I know about algorithms I learned along with my graduate students. Several of them (Yaw-Ling Lin, Sundaram Gopalakrishnan, Ting Chen, Francine Evans, Harald Rau, Ricky Bradley, and Dimitris Margaritis) are the real heroes of the war stories related within. My Stony Brook friends and algorithm colleagues Estie Arkin, Michael Bender, Jie Gao, and Joe Mitchell have always been a pleasure to work and be with. Finally, thanks to Michael Brochstein and the rest of the city contingent for revealing a proper life well beyond Stony Brook.

Caveat

It is traditional for the author to magnanimously accept the blame for whatever deficiencies remain. I don't. Any errors, deficiencies, or problems in this book are somebody else's fault, but I would appreciate knowing about them so as to determine who is to blame.

Steven S. Skiena
Department of Computer Science
Stony Brook University
Stony Brook, NY 11794-4400
<http://www.cs.sunysb.edu/~skiena>
April 2008

Contents

I Practical Algorithm Design	1
1 Introduction to Algorithm Design	3
1.1 Robot Tour Optimization	5
1.2 Selecting the Right Jobs	9
1.3 Reasoning about Correctness	11
1.4 Modeling the Problem	19
1.5 About the War Stories	22
1.6 War Story: Psychic Modeling	23
1.7 Exercises	27
2 Algorithm Analysis	31
2.1 The RAM Model of Computation	31
2.2 The Big Oh Notation	34
2.3 Growth Rates and Dominance Relations	37
2.4 Working with the Big Oh	40
2.5 Reasoning About Efficiency	41
2.6 Logarithms and Their Applications	46
2.7 Properties of Logarithms	50
2.8 War Story: Mystery of the Pyramids	51
2.9 Advanced Analysis (*)	54
2.10 Exercises	57
3 Data Structures	65
3.1 Contiguous vs. Linked Data Structures	66

3.2	Stacks and Queues	71
3.3	Dictionaries	72
3.4	Binary Search Trees	77
3.5	Priority Queues	83
3.6	War Story: Stripping Triangulations	85
3.7	Hashing and Strings	89
3.8	Specialized Data Structures	93
3.9	War Story: String 'em Up	94
3.10	Exercises	98
4	Sorting and Searching	103
4.1	Applications of Sorting	104
4.2	Pragmatics of Sorting	107
4.3	Heapsort: Fast Sorting via Data Structures	108
4.4	War Story: Give me a Ticket on an Airplane	118
4.5	Mergesort: Sorting by Divide-and-Conquer	120
4.6	Quicksort: Sorting by Randomization	123
4.7	Distribution Sort: Sorting via Bucketing	129
4.8	War Story: Skiena for the Defense	131
4.9	Binary Search and Related Algorithms	132
4.10	Divide-and-Conquer	135
4.11	Exercises	139
5	Graph Traversal	145
5.1	Flavors of Graphs	146
5.2	Data Structures for Graphs	151
5.3	War Story: I was a Victim of Moore's Law	155
5.4	War Story: Getting the Graph	158
5.5	Traversing a Graph	161
5.6	Breadth-First Search	162
5.7	Applications of Breadth-First Search	166
5.8	Depth-First Search	169
5.9	Applications of Depth-First Search	172
5.10	Depth-First Search on Directed Graphs	178
5.11	Exercises	184
6	Weighted Graph Algorithms	191
6.1	Minimum Spanning Trees	192
6.2	War Story: Nothing but Nets	202
6.3	Shortest Paths	205
6.4	War Story: Dialing for Documents	212
6.5	Network Flows and Bipartite Matching	217
6.6	Design Graphs, Not Algorithms	222
6.7	Exercises	225

7 Combinatorial Search and Heuristic Methods	230
7.1 Backtracking	231
7.2 Search Pruning	238
7.3 Sudoku	239
7.4 War Story: Covering Chessboards	244
7.5 Heuristic Search Methods	247
7.6 War Story: Only it is Not a Radio	260
7.7 War Story: Annealing Arrays	263
7.8 Other Heuristic Search Methods	266
7.9 Parallel Algorithms	267
7.10 War Story: Going Nowhere Fast	268
7.11 Exercises	270
8 Dynamic Programming	273
8.1 Caching vs. Computation	274
8.2 Approximate String Matching	280
8.3 Longest Increasing Sequence	289
8.4 War Story: Evolution of the Lobster	291
8.5 The Partition Problem	294
8.6 Parsing Context-Free Grammars	298
8.7 Limitations of Dynamic Programming: TSP	301
8.8 War Story: What's Past is Prolog	304
8.9 War Story: Text Compression for Bar Codes	307
8.10 Exercises	310
9 Intractable Problems and Approximation Algorithms	316
9.1 Problems and Reductions	317
9.2 Reductions for Algorithms	319
9.3 Elementary Hardness Reductions	323
9.4 Satisfiability	328
9.5 Creative Reductions	330
9.6 The Art of Proving Hardness	334
9.7 War Story: Hard Against the Clock	337
9.8 War Story: And Then I Failed	339
9.9 P vs. NP	341
9.10 Dealing with NP-complete Problems	344
9.11 Exercises	350
10 How to Design Algorithms	356
II The Hitchhiker's Guide to Algorithms	361
11 A Catalog of Algorithmic Problems	363

12 Data Structures	366
12.1 Dictionaries	367
12.2 Priority Queues	373
12.3 Suffix Trees and Arrays	377
12.4 Graph Data Structures	381
12.5 Set Data Structures	385
12.6 Kd-Trees	389
13 Numerical Problems	393
13.1 Solving Linear Equations	395
13.2 Bandwidth Reduction	398
13.3 Matrix Multiplication	401
13.4 Determinants and Permanents	404
13.5 Constrained and Unconstrained Optimization	407
13.6 Linear Programming	411
13.7 Random Number Generation	415
13.8 Factoring and Primality Testing	420
13.9 Arbitrary-Precision Arithmetic	423
13.10 Knapsack Problem	427
13.11 Discrete Fourier Transform	431
14 Combinatorial Problems	434
14.1 Sorting	436
14.2 Searching	441
14.3 Median and Selection	445
14.4 Generating Permutations	448
14.5 Generating Subsets	452
14.6 Generating Partitions	456
14.7 Generating Graphs	460
14.8 Calendrical Calculations	465
14.9 Job Scheduling	468
14.10 Satisfiability	472
15 Graph Problems: Polynomial-Time	475
15.1 Connected Components	477
15.2 Topological Sorting	481
15.3 Minimum Spanning Tree	484
15.4 Shortest Path	489
15.5 Transitive Closure and Reduction	495
15.6 Matching	498
15.7 Eulerian Cycle/Chinese Postman	502
15.8 Edge and Vertex Connectivity	505
15.9 Network Flow	509
15.10 Drawing Graphs Nicely	513

15.11 Drawing Trees	517
15.12 Planarity Detection and Embedding	520
16 Graph Problems: Hard Problems	523
16.1 Clique	525
16.2 Independent Set	528
16.3 Vertex Cover	530
16.4 Traveling Salesman Problem	533
16.5 Hamiltonian Cycle	538
16.6 Graph Partition	541
16.7 Vertex Coloring	544
16.8 Edge Coloring	548
16.9 Graph Isomorphism	550
16.10 Steiner Tree	555
16.11 Feedback Edge/Vertex Set	559
17 Computational Geometry	562
17.1 Robust Geometric Primitives	564
17.2 Convex Hull	568
17.3 Triangulation	572
17.4 Voronoi Diagrams	576
17.5 Nearest Neighbor Search	580
17.6 Range Search	584
17.7 Point Location	587
17.8 Intersection Detection	591
17.9 Bin Packing	595
17.10 Medial-Axis Transform	598
17.11 Polygon Partitioning	601
17.12 Simplifying Polygons	604
17.13 Shape Similarity	607
17.14 Motion Planning	610
17.15 Maintaining Line Arrangements	614
17.16 Minkowski Sum	617
18 Set and String Problems	620
18.1 Set Cover	621
18.2 Set Packing	625
18.3 String Matching	628
18.4 Approximate String Matching	631
18.5 Text Compression	637
18.6 Cryptography	641
18.7 Finite State Machine Minimization	646
18.8 Longest Common Substring/Subsequence	650
18.9 Shortest Common Superstring	654

19 Algorithmic Resources	657
19.1 Software Systems	657
19.2 Data Sources	663
19.3 Online Bibliographic Resources	663
19.4 Professional Consulting Services	664
Bibliography	665
Index	709

Introduction to Algorithm Design

What is an algorithm? An algorithm is a procedure to accomplish a specific task. An algorithm is the idea behind any reasonable computer program.

To be interesting, an algorithm must solve a general, well-specified *problem*. An algorithmic problem is specified by describing the complete set of *instances* it must work on and of its output after running on one of these instances. This distinction, between a problem and an instance of a problem, is fundamental. For example, the algorithmic *problem* known as *sorting* is defined as follows:

Problem: Sorting

Input: A sequence of n keys a_1, \dots, a_n .

Output: The permutation (reordering) of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_{n-1} \leq a'_n$.

An *instance* of sorting might be an array of names, like $\{Mike, Bob, Sally, Jill, Jan\}$, or a list of numbers like $\{154, 245, 568, 324, 654, 324\}$. Determining that you are dealing with a general problem is your first step towards solving it.

An *algorithm* is a procedure that takes any of the possible input instances and transforms it to the desired output. There are many different algorithms for solving the problem of sorting. For example, *insertion sort* is a method for sorting that starts with a single element (thus forming a trivially sorted list) and then incrementally inserts the remaining elements so that the list stays sorted. This algorithm, implemented in C, is described below:

I N S E R T I O N S O R T
I N S E R T I O N S O R T
I N S E R T I O N S O R T
E I N S R T I O N S O R T
E I N R S T I O N S O R T
E I N R S T I O N S O R T
E I I N R S T I O N S O R T
E I I N O R S T I N S O R T
E I I N N O R S T I N S O R T
E I I N N O R S S T O R T
E I I N N O O R R S S T R T
E I I N N O O R R S S T T T

Figure 1.1: Animation of insertion sort in action (time flows down)

```
insertion_sort(item s[], int n)
{
    int i,j; /* counters */

    for (i=1; i<n; i++) {
        j=i;
        while ((j>0) && (s[j] < s[j-1])) {
            swap(&s[j],&s[j-1]);
            j = j-1;
        }
    }
}
```

An animation of the logical flow of this algorithm on a particular instance (the letters in the word “INSERTIONSORT”) is given in Figure 1.1

Note the generality of this algorithm. It works just as well on names as it does on numbers, given the appropriate comparison operation ($<$) to test which of the two keys should appear first in sorted order. It can be readily verified that this algorithm correctly orders every possible input instance according to our definition of the sorting problem.

There are three desirable properties for a good algorithm. We seek algorithms that are *correct* and *efficient*, while being *easy to implement*. These goals may not be simultaneously achievable. In industrial settings, any program that seems to give good enough answers without slowing the application down is often acceptable, regardless of whether a better algorithm exists. The issue of finding the best possible answer or achieving maximum efficiency usually arises in industry only after serious performance or legal troubles.

In this chapter, we will focus on the issues of algorithm correctness, and defer a discussion of efficiency concerns to Chapter 2. It is seldom obvious whether a given

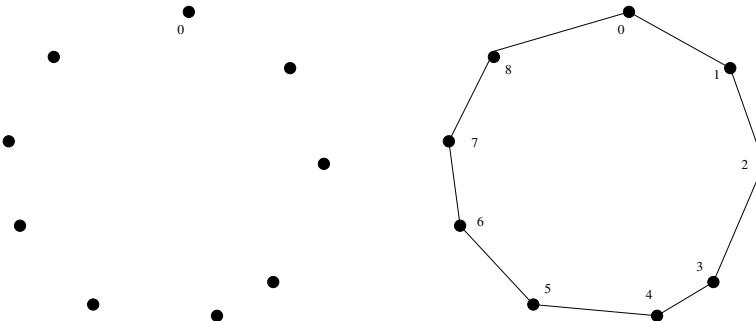


Figure 1.2: A good instance for the nearest-neighbor heuristic

algorithm correctly solves a given problem. Correct algorithms usually come with a proof of correctness, which is an explanation of *why* we know that the algorithm must take every instance of the problem to the desired result. However, before we go further we demonstrate why “*it’s obvious*” never suffices as a proof of correctness, and is usually flat-out wrong.

1.1 Robot Tour Optimization

Let’s consider a problem that arises often in manufacturing, transportation, and testing applications. Suppose we are given a robot arm equipped with a tool, say a soldering iron. In manufacturing circuit boards, all the chips and other components must be fastened onto the substrate. More specifically, each chip has a set of contact points (or wires) that must be soldered to the board. To program the robot arm for this job, we must first construct an ordering of the contact points so the robot visits (and solders) the first contact point, then the second point, third, and so forth until the job is done. The robot arm then proceeds back to the first contact point to prepare for the next board, thus turning the tool-path into a closed tour, or cycle.

Robots are expensive devices, so we want the tour that minimizes the time it takes to assemble the circuit board. A reasonable assumption is that the robot arm moves with fixed speed, so the time to travel between two points is proportional to their distance. In short, we must solve the following algorithm problem:

Problem: Robot Tour Optimization

Input: A set S of n points in the plane.

Output: What is the shortest cycle tour that visits each point in the set S ?

You are given the job of programming the robot arm. Stop right now and think up an algorithm to solve this problem. I’ll be happy to wait until you find one...

Several algorithms might come to mind to solve this problem. Perhaps the most popular idea is the *nearest-neighbor* heuristic. Starting from some point p_0 , we walk first to its nearest neighbor p_1 . From p_1 , we walk to its nearest unvisited neighbor, thus excluding only p_0 as a candidate. We now repeat this process until we run out of unvisited points, after which we return to p_0 to close off the tour. Written in pseudo-code, the nearest-neighbor heuristic looks like this:

```

NearestNeighbor( $P$ )
    Pick and visit an initial point  $p_0$  from  $P$ 
     $p = p_0$ 
     $i = 0$ 
    While there are still unvisited points
         $i = i + 1$ 
        Select  $p_i$  to be the closest unvisited point to  $p_{i-1}$ 
        Visit  $p_i$ 
    Return to  $p_0$  from  $p_{n-1}$ 
```

This algorithm has a lot to recommend it. It is simple to understand and implement. It makes sense to visit nearby points before we visit faraway points to reduce the total travel time. The algorithm works perfectly on the example in Figure 1.2. The nearest-neighbor rule is reasonably efficient, for it looks at each pair of points (p_i, p_j) at most twice: once when adding p_i to the tour, the other when adding p_j . Against all these positives there is only one problem. This algorithm is completely wrong.

Wrong? How can it be wrong? The algorithm always finds a tour, but it doesn't necessarily find the shortest possible tour. It doesn't necessarily even come close. Consider the set of points in Figure 1.3, all of which lie spaced along a line. The numbers describe the distance that each point lies to the left or right of the point labeled '0'. When we start from the point '0' and repeatedly walk to the nearest unvisited neighbor, we might keep jumping left-right-left-right over '0' as the algorithm offers no advice on how to break ties. A much better (indeed optimal) tour for these points starts from the leftmost point and visits each point as we walk right before returning at the rightmost point.

Try now to imagine your boss's delight as she watches a demo of your robot arm hopscotching left-right-left-right during the assembly of such a simple board.

"But wait," you might be saying. "The problem was in starting at point '0'. Instead, why don't we start the nearest-neighbor rule using the leftmost point as the initial point p_0 ? By doing this, we will find the optimal solution on this instance."

That is 100% true, at least until we rotate our example 90 degrees. Now all points are equally leftmost. If the point '0' were moved just slightly to the left, it would be picked as the starting point. Now the robot arm will hopscotch up-down-up-down instead of left-right-left-right, but the travel time will be just as bad as before. No matter what you do to pick the first point, the nearest-neighbor rule is doomed to work incorrectly on certain point sets.

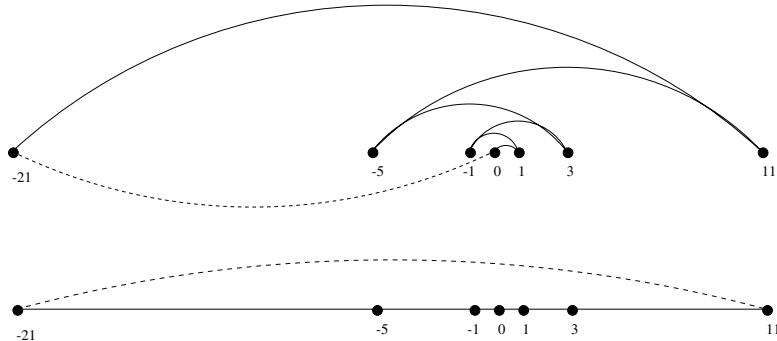


Figure 1.3: A bad instance for the nearest-neighbor heuristic, with the optimal solution

Maybe what we need is a different approach. Always walking to the closest point is too restrictive, since it seems to trap us into making moves we didn't want. A different idea might be to repeatedly connect the closest pair of endpoints whose connection will not create a problem, such as premature termination of the cycle. Each vertex begins as its own single vertex chain. After merging everything together, we will end up with a single chain containing all the points in it. Connecting the final two endpoints gives us a cycle. At any step during the execution of this *closest-pair heuristic*, we will have a set of single vertices and vertex-disjoint chains available to merge. In pseudocode:

`ClosestPair(P)`

Let n be the number of points in set P .

For $i = 1$ to $n - 1$ do

$d = \infty$

For each pair of endpoints (s, t) from distinct vertex chains

if $\text{dist}(s, t) \leq d$ then $s_m = s$, $t_m = t$, and $d = \text{dist}(s, t)$

Connect (s_m, t_m) by an edge

Connect the two endpoints by an edge

This closest-pair rule does the right thing in the example in Figure 1.3. It starts by connecting '0' to its immediate neighbors, the points 1 and -1. Subsequently, the next closest pair will alternate left-right, growing the central path by one link at a time. The closest-pair heuristic is somewhat more complicated and less efficient than the previous one, but at least it gives the right answer in this example.

But this is not true in all examples. Consider what this algorithm does on the point set in Figure 1.4(l). It consists of two rows of equally spaced points, with the rows slightly closer together (distance $1 - e$) than the neighboring points are spaced within each row (distance $1 + e$). Thus the closest pairs of points stretch across the gap, not around the boundary. After we pair off these points, the closest

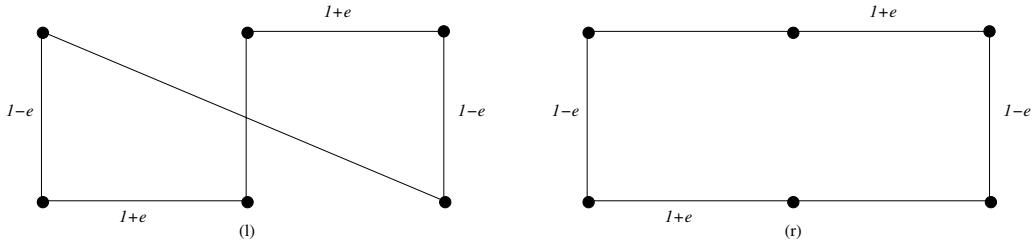


Figure 1.4: A bad instance for the closest-pair heuristic, with the optimal solution

remaining pairs will connect these pairs alternately around the boundary. The total path length of the closest-pair tour is $3(1 - e) + 2(1 + e) + \sqrt{(1 - e)^2 + (2 + 2e)^2}$. Compared to the tour shown in Figure 1.4(r), we travel over 20% farther than necessary when $e \approx 0$. Examples exist where the penalty is considerably worse than this.

Thus this second algorithm is also wrong. Which one of these algorithms performs better? You can't tell just by looking at them. Clearly, both heuristics can end up with very bad tours on very innocent-looking input.

At this point, you might wonder what a correct algorithm for our problem looks like. Well, we could try enumerating *all* possible orderings of the set of points, and then select the ordering that minimizes the total length:

OptimalTSP(P)

$d = \infty$

For each of the $n!$ permutations P_i of point set P

If $(\text{cost}(P_i) \leq d)$ then $d = \text{cost}(P_i)$ and $P_{\min} = P_i$

Return P_{\min}

Since all possible orderings are considered, we are guaranteed to end up with the shortest possible tour. This algorithm is correct, since we pick the best of all the possibilities. But it is also extremely slow. The fastest computer in the world couldn't hope to enumerate all the $20! = 2,432,902,008,176,640,000$ orderings of 20 points within a day. For real circuit boards, where $n \approx 1,000$, forget about it. All of the world's computers working full time wouldn't come close to finishing the problem before the end of the universe, at which point it presumably becomes moot.

The quest for an efficient algorithm to solve this problem, called the *traveling salesman problem* (TSP), will take us through much of this book. If you need to know how the story ends, check out the catalog entry for the traveling salesman problem in Section 16.4 (page 533).

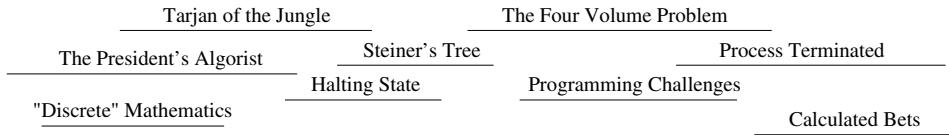


Figure 1.5: An instance of the non-overlapping movie scheduling problem

Take-Home Lesson: There is a fundamental difference between *algorithms*, which always produce a correct result, and *heuristics*, which may usually do a good job but without providing any guarantee.

1.2 Selecting the Right Jobs

Now consider the following scheduling problem. Imagine you are a highly-in-demand actor, who has been presented with offers to star in n different movie projects under development. Each offer comes specified with the first and last day of filming. To take the job, you must commit to being available throughout this entire period. Thus you cannot simultaneously accept two jobs whose intervals overlap.

For an artist such as yourself, the criteria for job acceptance is clear: you want to make as much money as possible. Because each of these films pays the same fee per film, this implies you seek the largest possible set of jobs (intervals) such that no two of them conflict with each other.

For example, consider the available projects in Figure 1.5. We can star in at most four films, namely “*Discrete*” *Mathematics*, *Programming Challenges*, *Calculated Bets*, and one of either *Halting State* or *Steiner’s Tree*.

You (or your agent) must solve the following algorithmic scheduling problem:

Problem: Movie Scheduling Problem

Input: A set I of n intervals on the line.

Output: What is the largest subset of mutually non-overlapping intervals which can be selected from I ?

You are given the job of developing a scheduling algorithm for this task. Stop right now and try to find one. Again, I’ll be happy to wait.

There are several ideas that may come to mind. One is based on the notion that it is best to work whenever work is available. This implies that you should start with the job with the earliest start date – after all, there is no other job you can work on, then at least during the beginning of this period.

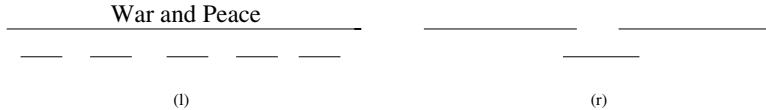


Figure 1.6: Bad instances for the (l) earliest job first and (r) shortest job first heuristics.

EarliestJobFirst(I)

Accept the earliest starting job j from I which does not overlap any previously accepted job, and repeat until no more such jobs remain.

This idea makes sense, at least until we realize that accepting the earliest job might block us from taking many other jobs if that first job is long. Check out Figure 1.6(l), where the epic “War and Peace” is both the first job available and long enough to kill off all other prospects.

This bad example naturally suggests another idea. The problem with “War and Peace” is that it is too long. Perhaps we should start by taking the shortest job, and keep seeking the shortest available job at every turn. Maximizing the number of jobs we do in a given period is clearly connected to banging them out as quickly as possible. This yields the heuristic:

ShortestJobFirst(I)

While ($I \neq \emptyset$) do

 Accept the shortest possible job j from I .

 Delete j , and any interval which intersects j from I .

Again this idea makes sense, at least until we realize that accepting the shortest job might block us from taking two other jobs, as shown in Figure 1.6(r). While the potential loss here seems smaller than with the previous heuristic, it can readily limit us to half the optimal payoff.

At this point, an algorithm where we try all possibilities may start to look good, because we can be certain it is correct. If we ignore the details of testing whether a set of intervals are in fact disjoint, it looks something like this:

ExhaustiveScheduling(I)

$j = 0$

$S_{max} = \emptyset$

 For each of the 2^n subsets S_i of intervals I

 If (S_i is mutually non-overlapping) and ($\text{size}(S_i) > j$)
 then $j = \text{size}(S_i)$ and $S_{max} = S_i$.

 Return S_{max}

But how slow is it? The key limitation is enumerating the 2^n subsets of n things. The good news is that this is *much* better than enumerating all $n!$ orders

of n things, as proposed for the robot tour optimization problem. There are only about one million subsets when $n = 20$, which could be exhaustively counted within seconds on a decent computer. However, when fed $n = 100$ movies, 2^{100} is much much greater than the $20!$ which made our robot cry “uncle” in the previous problem.

The difference between our scheduling and robotics problems are that there *is* an algorithm which solves movie scheduling both correctly and efficiently. Think about the first job to terminate—i.e. the interval x which contains the rightmost point which is leftmost among all intervals. This role is played by “*Discrete Mathematics*” in Figure 1.5. Other jobs may well have started before x , but all of these must at least partially overlap each other, so we can select at most one from the group. The first of these jobs to terminate is x , so any of the overlapping jobs potentially block out other opportunities to the right of it. Clearly we can never lose by picking x . This suggests the following correct, efficient algorithm:

```
OptimalScheduling(I)
  While ( $I \neq \emptyset$ ) do
    Accept the job  $j$  from  $I$  with the earliest completion date.
    Delete  $j$ , and any interval which intersects  $j$  from  $I$ .
```

Ensuring the optimal answer over all possible inputs is a difficult but often achievable goal. Seeking counterexamples that break pretender algorithms is an important part of the algorithm design process. Efficient algorithms are often lurking out there; this book seeks to develop your skills to help you find them.

Take-Home Lesson: Reasonable-looking algorithms can easily be incorrect. Algorithm correctness is a property that must be carefully demonstrated.

1.3 Reasoning about Correctness

Hopefully, the previous examples have opened your eyes to the subtleties of algorithm correctness. We need tools to distinguish correct algorithms from incorrect ones, the primary one of which is called a *proof*.

A proper mathematical proof consists of several parts. First, there is a clear, precise statement of what you are trying to prove. Second, there is a set of assumptions of things which are taken to be true and hence used as part of the proof. Third, there is a chain of reasoning which takes you from these assumptions to the statement you are trying to prove. Finally, there is a little square (■) or *QED* at the bottom to denote that you have finished, representing the Latin phrase for “thus it is demonstrated.”

This book is not going to emphasize formal proofs of correctness, because they are very difficult to do right and quite misleading when you do them wrong. A proof is indeed a *demonstration*. Proofs are useful only when they are honest; crisp arguments explaining why an algorithm satisfies a nontrivial correctness property.

Correct algorithms require careful exposition, and efforts to show both correctness and *not incorrectness*. We develop tools for doing so in the subsections below.

1.3.1 Expressing Algorithms

Reasoning about an algorithm is impossible without a careful description of the sequence of steps to be performed. The three most common forms of algorithmic notation are (1) English, (2) pseudocode, or (3) a real programming language. We will use all three in this book. Pseudocode is perhaps the most mysterious of the bunch, but it is best defined as a programming language that never complains about syntax errors. All three methods are useful because there is a natural tradeoff between greater ease of expression and precision. English is the most natural but least precise programming language, while Java and C/C++ are precise but difficult to write and understand. Pseudocode is generally useful because it represents a happy medium.

The choice of which notation is best depends upon which method you are most comfortable with. I usually prefer to describe the *ideas* of an algorithm in English, moving to a more formal, programming-language-like pseudocode or even real code to clarify sufficiently tricky details.

A common mistake my students make is to use pseudocode to dress up an ill-defined idea so that it looks more formal. Clarity should be the goal. For example, the `ExhaustiveScheduling` algorithm on page 10 could have been better written in English as:

`ExhaustiveScheduling(I)`

Test all 2^n subsets of intervals from I , and return the largest subset consisting of mutually non-overlapping intervals.

Take-Home Lesson: The heart of any algorithm is an *idea*. If your idea is not clearly revealed when you express an algorithm, then you are using too low-level a notation to describe it.

1.3.2 Problems and Properties

We need more than just an algorithm description in order to demonstrate correctness. We also need a careful description of the problem that it is intended to solve.

Problem specifications have two parts: (1) the set of allowed input instances, and (2) the required properties of the algorithm's output. It is impossible to prove the correctness of an algorithm for a fuzzily-stated problem. Put another way, ask the wrong problem and you will get the wrong answer.

Some problem specifications allow too broad a class of input instances. Suppose we had allowed film projects in our movie scheduling problem to have gaps in

production (i.e., filming in September and November but a hiatus in October). Then the schedule associated with any particular film would consist of a given *set* of intervals. Our star would be free to take on two interleaving but not overlapping projects (such as the film above nested with one filming in August and October). The earliest completion algorithm would not work for such a generalized scheduling problem. Indeed, *no* efficient algorithm exists for this generalized problem.

Take-Home Lesson: An important and honorable technique in algorithm design is to narrow the set of allowable instances until there *is* a correct and efficient algorithm. For example, we can restrict a graph problem from general graphs down to trees, or a geometric problem from two dimensions down to one.

There are two common traps in specifying the output requirements of a problem. One is asking an ill-defined question. Asking for the *best* route between two places on a map is a silly question unless you define what *best* means. Do you mean the shortest route in total distance, or the fastest route, or the one minimizing the number of turns?

The second trap is creating compound goals. The three path-planning criteria mentioned above are all well-defined goals that lead to correct, efficient optimization algorithms. However, you must pick a single criteria. A goal like *Find the shortest path from a to b that doesn't use more than twice as many turns as necessary* is perfectly well defined, but complicated to reason and solve.

I encourage you to check out the problem statements for each of the 75 catalog problems in the second part of this book. Finding the right formulation for your problem is an important part of solving it. And studying the definition of all these classic algorithm problems will help you recognize when someone else has thought about similar problems before you.

1.3.3 Demonstrating Incorrectness

The best way to prove that an algorithm is *incorrect* is to produce an instance in which it yields an incorrect answer. Such instances are called *counter-examples*. No rational person will ever leap to the defense of an algorithm after a counter-example has been identified. Very simple instances can instantly kill reasonable-looking heuristics with a quick *touché*. Good counter-examples have two important properties:

- *Verifiability* – To demonstrate that a particular instance is a counter-example to a particular algorithm, you must be able to (1) calculate what answer your algorithm will give in this instance, and (2) display a better answer so as to prove the algorithm didn’t find it.

Since you must hold the given instance in your head to reason about it, an important part of verifiability is...

- *Simplicity* – Good counter-examples have all unnecessary details boiled away. They make clear exactly *why* the proposed algorithm fails. Once a counter-example has been found, it is worth simplifying it down to its essence. For example, the counter-example of Figure 1.6(l) could be made simpler and better by reducing the number of overlapped segments from four to two.

Hunting for counter-examples is a skill worth developing. It bears some similarity to the task of developing test sets for computer programs, but relies more on inspiration than exhaustion. Here are some techniques to aid your quest:

- *Think small* – Note that the robot tour counter-examples I presented boiled down to six points or less, and the scheduling counter-examples to only three intervals. This is indicative of the fact that when algorithms fail, there is usually a very simple example on which they fail. Amateur algorists tend to draw a big messy instance and then stare at it helplessly. The pros look carefully at several small examples, because they are easier to verify and reason about.
- *Think exhaustively* – There are only a small number of possibilities for the smallest nontrivial value of n . For example, there are only three interesting ways two intervals on the line can occur: (1) as disjoint intervals, (2) as overlapping intervals, and (3) as properly nesting intervals, one within the other. All cases of three intervals (including counter-examples to both movie heuristics) can be systematically constructed by adding a third segment in each possible way to these three instances.
- *Hunt for the weakness* – If a proposed algorithm is of the form “always take the biggest” (better known as the *greedy algorithm*), think about why that might prove to be the wrong thing to do. In particular, . . .
- *Go for a tie* – A devious way to break a greedy heuristic is to provide instances where everything is the same size. Suddenly the heuristic has nothing to base its decision on, and perhaps has the freedom to return something suboptimal as the answer.
- *Seek extremes* – Many counter-examples are mixtures of huge and tiny, left and right, few and many, near and far. It is usually easier to verify or reason about extreme examples than more muddled ones. Consider two tightly bunched clouds of points separated by a much larger distance d . The optimal TSP tour will be essentially $2d$ regardless of the number of points, because what happens within each cloud doesn’t really matter.

Take-Home Lesson: Searching for counterexamples is the best way to disprove the correctness of a heuristic.

1.3.4 Induction and Recursion

Failure to find a counterexample to a given algorithm does not mean “it is obvious” that the algorithm is correct. A proof or demonstration of correctness is needed. Often mathematical induction is the method of choice.

When I first learned about mathematical induction it seemed like complete magic. You proved a formula like $\sum_{i=1}^n i = n(n + 1)/2$ for some basis case like 1 or 2, then *assumed* it was true all the way to $n - 1$ before proving it was true for general n using the assumption. That was a proof? Ridiculous!

When I first learned the programming technique of recursion it also seemed like complete magic. The program tested whether the input argument was some basis case like 1 or 2. If not, you solved the bigger case by breaking it into pieces and *calling the subprogram itself* to solve these pieces. That was a program? Ridiculous!

The reason both seemed like magic is because recursion *is* mathematical induction. In both, we have general and boundary conditions, with the general condition breaking the problem into smaller and smaller pieces. The *initial* or boundary condition terminates the recursion. Once you understand either recursion or induction, you should be able to see why the other one also works.

I’ve heard it said that a computer scientist is a mathematician who only knows how to prove things by induction. This is partially true because computer scientists are lousy at proving things, but primarily because so many of the algorithms we study are either recursive or incremental.

Consider the correctness of *insertion sort*, which we introduced at the beginning of this chapter. The *reason* it is correct can be shown inductively:

- The basis case consists of a single element, and by definition a one-element array is completely sorted.
- In general, we can assume that the first $n - 1$ elements of array A are completely sorted after $n - 1$ iterations of insertion sort.
- To insert one last element x to A , we find where it goes, namely the unique spot between the biggest element less than or equal to x and the smallest element greater than x . This is done by moving all the greater elements back by one position, creating room for x in the desired location. ■

One must be suspicious of inductive proofs, however, because very subtle reasoning errors can creep in. The first are *boundary errors*. For example, our insertion sort correctness proof above boldly stated that there was a unique place to insert x between two elements, when our basis case was a single-element array. Greater care is needed to properly deal with the special cases of inserting the minimum or maximum elements.

The second and more common class of inductive proof errors concerns cavalier extension claims. Adding one extra item to a given problem instance might cause the entire optimal solution to change. This was the case in our scheduling problem (see Figure 1.7). The optimal schedule after inserting a new segment may contain



Figure 1.7: Large-scale changes in the optimal solution (boxes) after inserting a single interval (dashed) into the instance

none of the segments of any particular optimal solution prior to insertion. Boldly ignoring such difficulties can lead to very convincing inductive proofs of incorrect algorithms.

Take-Home Lesson: Mathematical induction is usually the right way to verify the correctness of a recursive or incremental insertion algorithm.

Stop and Think: Incremental Correctness

Problem: Prove the correctness of the following recursive algorithm for incrementing natural numbers, i.e. $y \rightarrow y + 1$:

```
Increment(y)
  if  $y = 0$  then return(1) else
    if  $(y \bmod 2) = 1$  then
      return( $2 \cdot \text{Increment}(\lfloor y/2 \rfloor)$ )
    else return( $y + 1$ )
```

Solution: The correctness of this algorithm is certainly *not* obvious to me. But as it is recursive and I am a computer scientist, my natural instinct is to try to prove it by induction.

The basis case of $y = 0$ is obviously correctly handled. Clearly the value 1 is returned, and $0 + 1 = 1$.

Now assume the function works correctly for the general case of $y = n - 1$. Given this, we must demonstrate the truth for the case of $y = n$. Half of the cases are easy, namely the even numbers (For which $(y \bmod 2) = 0$), since $y + 1$ is explicitly returned.

For the odd numbers, the answer depends upon what is returned by $\text{Increment}(\lfloor y/2 \rfloor)$. Here we want to use our inductive assumption, but it isn't quite right. We have assumed that `increment` worked correctly for $y = n - 1$, but not for a value which is about half of it. We can fix this problem by strengthening our assumption to declare that the general case holds for all $y \leq n - 1$. This costs us nothing in principle, but is necessary to establish the correctness of the algorithm.

Now, the case of odd y (i.e. $y = 2m + 1$ for some integer m) can be dealt with as:

$$\begin{aligned} 2 \cdot \text{Increment}(\lfloor (2m+1)/2 \rfloor) &= 2 \cdot \text{Increment}(\lfloor m+1/2 \rfloor) \\ &= 2 \cdot \text{Increment}(m) \\ &= 2(m+1) \\ &= 2m+2 = y+1 \end{aligned}$$

and the general case is resolved. ■

1.3.5 Summations

Mathematical summation formulae arise often in algorithm analysis, which we will study in Chapter 2. Further, proving the correctness of summation formulae is a classic application of induction. Several exercises on inductive proofs of summations appear as exercises at the end this chapter. To make these more accessible, I review the basics of summations here.

Summation formulae are concise expressions describing the addition of an arbitrarily large set of numbers, in particular the formula

$$\sum_{i=1}^n f(i) = f(1) + f(2) + \dots + f(n)$$

There are simple closed forms for summations of many algebraic functions. For example, since n ones is n ,

$$\sum_{i=1}^n 1 = n$$

The sum of the first n integers can be seen by pairing up the i th and $(n-i+1)$ th integers:

$$\sum_{i=1}^n i = \sum_{i=1}^{n/2} (i + (n - i + 1)) = n(n+1)/2$$

Recognizing two basic classes of summation formulae will get you a long way in algorithm analysis:

- *Arithmetic progressions* – We already encountered arithmetic progressions when we saw $S(n) = \sum_i^n i = n(n+1)/2$ in the analysis of selection sort. From the big picture perspective, the important thing is that the sum is quadratic, not that the constant is $1/2$. In general,

$$S(n, p) = \sum_i^n i^p = \Theta(n^{p+1})$$

for $p \geq 1$. Thus the sum of squares is cubic, and the sum of cubes is quartic (if you use such a word). The “big Theta” notation ($\Theta(x)$) will be properly explained in Section 2.2.

For $p < -1$, this sum always converges to a constant, even as $n \rightarrow \infty$. The interesting case is between results in ...

- *Geometric series* – In geometric progressions, the index of the loop effects the exponent, i.e.

$$G(n, a) = \sum_{i=0}^n a^i = a(a^{n+1} - 1)/(a - 1)$$

How we interpret this sum depends upon the *base* of the progression, i.e. a . When $a < 1$, this converges to a constant even as $n \rightarrow \infty$.

This series convergence proves to be the great “free lunch” of algorithm analysis. It means that the sum of a linear number of things can be constant, not linear. For example, $1 + 1/2 + 1/4 + 1/8 + \dots \leq 2$ no matter how many terms we add up.

When $a > 1$, the sum grows rapidly with each new term, as in $1 + 2 + 4 + 8 + 16 + 32 = 63$. Indeed, $G(n, a) = \Theta(a^{n+1})$ for $a > 1$.

Stop and Think: Factorial Formulae

Problem: Prove that $\sum_{i=1}^n i \times i! = (n+1)! - 1$ by induction.

Solution: The inductive paradigm is straightforward. First verify the basis case (here we do $n = 1$, although $n = 0$ would be even more general):

$$\sum_{i=1}^1 i \times i! = 1 = (1+1)! - 1 = 2 - 1 = 1$$

Now assume the statement is true up to n . To prove the general case of $n+1$, observe that rolling out the largest term

$$\sum_{i=1}^{n+1} i \times i! = (n+1) \times (n+1)! + \sum_{i=1}^n i \times i!$$

reveals the left side of our inductive assumption. Substituting the right side gives us

$$\sum_{i=1}^{n+1} i \times i! = (n+1) \times (n+1)! + (n+1)! - 1$$

$$\begin{aligned}
 &= (n+1)! \times ((n+1)+1) - 1 \\
 &= (n+2)! - 1
 \end{aligned}$$

This general trick of separating out the largest term from the summation to reveal an instance of the inductive assumption lies at the heart of all such proofs.

1.4 Modeling the Problem

Modeling is the art of formulating your application in terms of precisely described, well-understood problems. Proper modeling is the key to applying algorithmic design techniques to real-world problems. Indeed, proper modeling can eliminate the need to design or even implement algorithms, by relating your application to what has been done before. Proper modeling is the key to effectively using the “Hitchhiker’s Guide” in Part II of this book.

Real-world applications involve real-world objects. You might be working on a system to route traffic in a network, to find the best way to schedule classrooms in a university, or to search for patterns in a corporate database. Most algorithms, however, are designed to work on rigorously defined *abstract* structures such as permutations, graphs, and sets. To exploit the algorithms literature, you must learn to describe your problem abstractly, in terms of procedures on fundamental structures.

1.4.1 Combinatorial Objects

Odds are very good that others have stumbled upon your algorithmic problem before you, perhaps in substantially different contexts. But to find out what is known about your particular “widget optimization problem,” you can’t hope to look in a book under *widget*. You must formulate widget optimization in terms of computing properties of common structures such as:

- *Permutations* – which are arrangements, or orderings, of items. For example, $\{1, 4, 3, 2\}$ and $\{4, 3, 2, 1\}$ are two distinct permutations of the same set of four integers. We have already seen permutations in the robot optimization problem, and in sorting. Permutations are likely the object in question whenever your problem seeks an “arrangement,” “tour,” “ordering,” or “sequence.”
- *Subsets* – which represent selections from a set of items. For example, $\{1, 3, 4\}$ and $\{2\}$ are two distinct subsets of the first four integers. Order does not matter in subsets the way it does with permutations, so the subsets $\{1, 3, 4\}$ and $\{4, 3, 1\}$ would be considered identical. We saw subsets arise in the movie scheduling problem. Subsets are likely the object in question whenever your problem seeks a “cluster,” “collection,” “committee,” “group,” “packaging,” or “selection.”

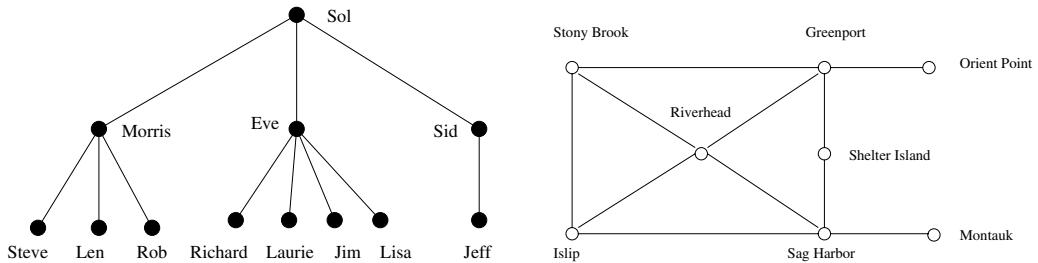


Figure 1.8: Modeling real-world structures with trees and graphs

- *Trees* – which represent hierarchical relationships between items. Figure 1.8(a) shows part of the family tree of the Skiena clan. Trees are likely the object in question whenever your problem seeks a “hierarchy,” “dominance relationship,” “ancestor/descendant relationship,” or “taxonomy.”
- *Graphs* – which represent relationships between arbitrary pairs of objects. Figure 1.8(b) models a network of roads as a graph, where the vertices are cities and the edges are roads connecting pairs of cities. Graphs are likely the object in question whenever you seek a “network,” “circuit,” “web,” or “relationship.”
- *Points* – which represent locations in some geometric space. For example, the locations of McDonald’s restaurants can be described by points on a map/plane. Points are likely the object in question whenever your problems work on “sites,” “positions,” “data records,” or “locations.”
- *Polygons* – which represent regions in some geometric spaces. For example, the borders of a country can be described by a polygon on a map/plane. Polygons and polyhedra are likely the object in question whenever you are working on “shapes,” “regions,” “configurations,” or “boundaries.”
- *Strings* – which represent sequences of characters or patterns. For example, the names of students in a class can be represented by strings. Strings are likely the object in question whenever you are dealing with “text,” “characters,” “patterns,” or “labels.”

These fundamental structures all have associated algorithm problems, which are presented in the catalog of Part II. Familiarity with these problems is important, because they provide the language we use to model applications. To become fluent in this vocabulary, browse through the catalog and study the *input* and *output* pictures for each problem. Understanding these problems, even at a cartoon/definition level, will enable you to know where to look later when the problem arises in your application.

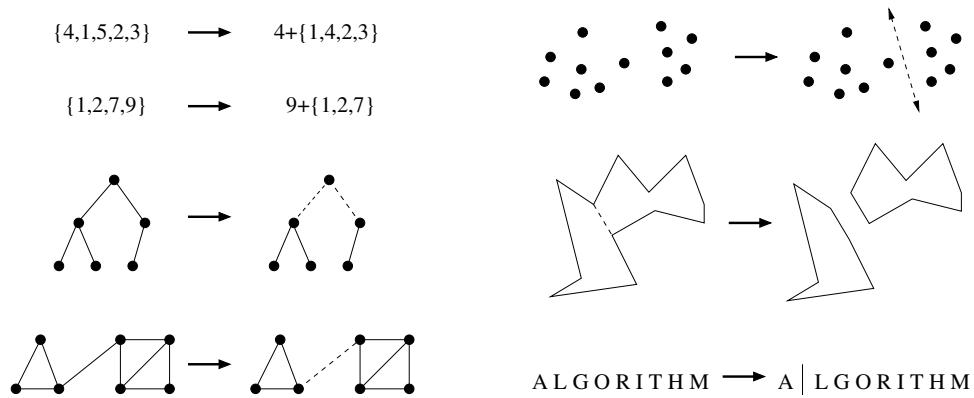


Figure 1.9: Recursive decompositions of combinatorial objects. (left column) Permutations, subsets, trees, and graphs. (right column) Point sets, polygons, and strings

Examples of successful application modeling will be presented in the war stories spaced throughout this book. However, some words of caution are in order. The act of modeling reduces your application to one of a small number of existing problems and structures. Such a process is inherently constraining, and certain details might not fit easily into the given target problem. Also, certain problems can be modeled in several different ways, some much better than others.

Modeling is only the first step in designing an algorithm for a problem. Be alert for how the details of your applications differ from a candidate model, but don't be too quick to say that your problem is unique and special. Temporarily ignoring details that don't fit can free the mind to ask whether they really were fundamental in the first place.

Take-Home Lesson: Modeling your application in terms of well-defined structures and algorithms is the most important single step towards a solution.

1.4.2 Recursive Objects

Learning to think recursively is learning to look for big things that are made from smaller things of *exactly the same type as the big thing*. If you think of houses as sets of rooms, then adding or deleting a room still leaves a house behind.

Recursive structures occur everywhere in the algorithmic world. Indeed, each of the abstract structures described above can be thought about recursively. You just have to see how you can break them down, as shown in Figure 1.9:

- *Permutations* – Delete the first element of a permutation of $\{1, \dots, n\}$ things and you get a permutation of the remaining $n - 1$ things. Permutations are recursive objects.

- *Subsets* – Every subset of the elements $\{1, \dots, n\}$ contains a subset of $\{1, \dots, n-1\}$ made visible by deleting element n if it is present. Subsets are recursive objects.
- *Trees* – Delete the root of a tree and what do you get? A collection of smaller trees. Delete any leaf of a tree and what do you get? A slightly smaller tree. Trees are recursive objects.
- *Graphs* – Delete any vertex from a graph, and you get a smaller graph. Now divide the vertices of a graph into two groups, left and right. Cut through all edges which span from left to right, and what do you get? Two smaller graphs, and a bunch of broken edges. Graphs are recursive objects.
- *Points* – Take a cloud of points, and separate them into two groups by drawing a line. Now you have two smaller clouds of points. Point sets are recursive objects.
- *Polygons* – Inserting any internal chord between two nonadjacent vertices of a simple polygon on n vertices cuts it into two smaller polygons. Polygons are recursive objects.
- *Strings* – Delete the first character from a string, and what do you get? A shorter string. Strings are recursive objects.

Recursive descriptions of objects require both decomposition rules and *basis cases*, namely the specification of the smallest and simplest objects where the decomposition stops. These basis cases are usually easily defined. Permutations and subsets of zero things presumably look like $\{\}$. The smallest interesting tree or graph consists of a single vertex, while the smallest interesting point cloud consists of a single point. Polygons are a little trickier; the smallest genuine simple polygon is a triangle. Finally, the empty string has zero characters in it. The decision of whether the basis case contains zero or one element is more a question of taste and convenience than any fundamental principle.

Such recursive decompositions will come to define many of the algorithms we will see in this book. Keep your eyes open for them.

1.5 About the War Stories

The best way to learn how careful algorithm design can have a huge impact on performance is to look at real-world case studies. By carefully studying other people's experiences, we learn how they might apply to our work.

Scattered throughout this text are several of my own algorithmic war stories, presenting our successful (and occasionally unsuccessful) algorithm design efforts on real applications. I hope that you will be able to internalize these experiences so that they will serve as models for your own attacks on problems.

Every one of the war stories is true. Of course, the stories improve somewhat in the retelling, and the dialogue has been punched up to make them more interesting to read. However, I have tried to honestly trace the process of going from a raw problem to a solution, so you can watch how this process unfolded.

The *Oxford English Dictionary* defines an *algorist* as “one skillful in reckonings or figuring.” In these stories, I have tried to capture some of the mindset of the algorist in action as they attack a problem.

The various war stories usually involve at least one, and often several, problems from the problem catalog in Part II. I reference the appropriate section of the catalog when such a problem occurs. This emphasizes the benefits of modeling your application in terms of standard algorithm problems. By using the catalog, you will be able to pull out what is known about any given problem whenever it is needed.

1.6 War Story: Psychic Modeling

The call came for me out of the blue as I sat in my office.

“Professor Skiena, I hope you can help me. I’m the President of Lotto Systems Group Inc., and we need an algorithm for a problem arising in our latest product.”

“Sure,” I replied. After all, the dean of my engineering school is always encouraging our faculty to interact more with industry.

“At Lotto Systems Group, we market a program designed to improve our customers’ psychic ability to predict winning lottery numbers.¹ In a standard lottery, each ticket consists of six numbers selected from, say, 1 to 44. Thus, any given ticket has only a very small chance of winning. However, after proper training, our clients can visualize, say, 15 numbers out of the 44 and be certain that at least four of them will be on the winning ticket. Are you with me so far?”

“Probably not,” I replied. But then I recalled how my dean encourages us to interact with industry.

“Our problem is this. After the psychic has narrowed the choices down to 15 numbers and is certain that at least 4 of them will be on the winning ticket, we must find the most efficient way to exploit this information. Suppose a cash prize is awarded whenever you pick at least three of the correct numbers on your ticket. We need an algorithm to construct the smallest set of tickets that we must buy in order to guarantee that we win at least one prize.”

“Assuming the psychic is correct?”

“Yes, assuming the psychic is correct. We need a program that prints out a list of all the tickets that the psychic should buy in order to minimize their investment. Can you help us?”

Maybe they did have psychic ability, for they had come to the right place. Identifying the best subset of tickets to buy was very much a combinatorial algorithm

¹Yes, this is a true story.

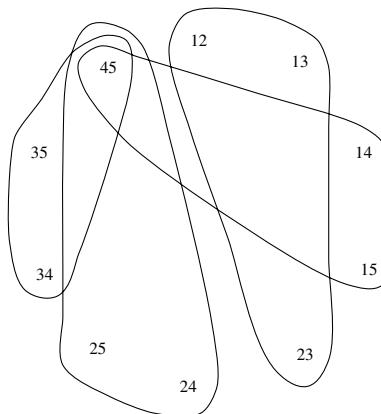


Figure 1.10: Covering all pairs of $\{1, 2, 3, 4, 5\}$ with tickets $\{1, 2, 3\}$, $\{1, 4, 5\}$, $\{2, 4, 5\}$, $\{3, 4, 5\}$

problem. It was going to be some type of covering problem, where each ticket we buy was going to “cover” some of the possible 4-element subsets of the psychic’s set. Finding the absolute smallest set of tickets to cover everything was a special instance of the NP-complete problem *set cover* (discussed in Section 18.1 (page 621)), and presumably computationally intractable.

It was indeed a special instance of set cover, completely specified by only four numbers: the size n of the candidate set S (typically $n \approx 15$), the number of slots k for numbers on each ticket (typically $k \approx 6$), the number of psychically-promised correct numbers j from S (say $j = 4$), and finally, the number of matching numbers l necessary to win a prize (say $l = 3$). Figure 1.10 illustrates a covering of a smaller instance, where $n = 5$, $j = k = 3$, and $l = 2$.

“Although it will be hard to find the *exact* minimum set of tickets to buy, with heuristics I should be able to get you pretty close to the cheapest covering ticket set,” I told him. “Will that be good enough?”

“So long as it generates better ticket sets than my competitor’s program, that will be fine. His system doesn’t always guarantee a win. I really appreciate your help on this, Professor Skiena.”

“One last thing. If your program can train people to pick lottery winners, why don’t you use it to win the lottery yourself?”

“I look forward to talking to you again real soon, Professor Skiena. Thanks for the help.”

I hung up the phone and got back to thinking. It seemed like the perfect project to give to a bright undergraduate. After modeling it in terms of sets and subsets, the basic components of a solution seemed fairly straightforward: