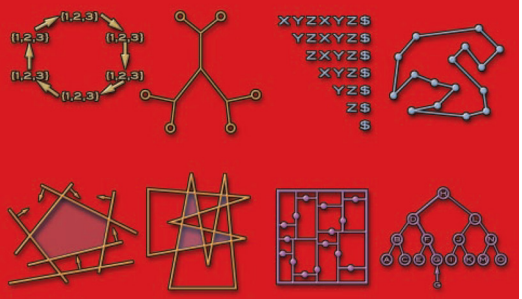
Second Edition

THE Algorithm Design MANUAL



Steven S. Skiena

Springer

The Algorithm Design Manual

Second Edition

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

Printed on acid-free paper

Springer Science+Business Media
springer.com

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/∼algorith* and 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 Wylde.

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

Part I

Practical Algorithm Design

1

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 $a_1, \ldots, a_n$.
Output: The permutation (reordering) of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_{n-1} \leq a'_n$.

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:

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_1,

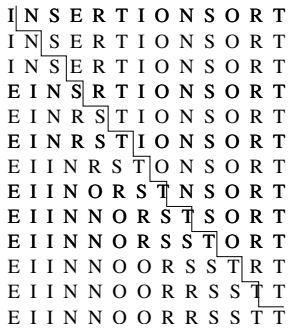


Figure 1.1: Animation of insertion sort in action (time flows down)

```
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

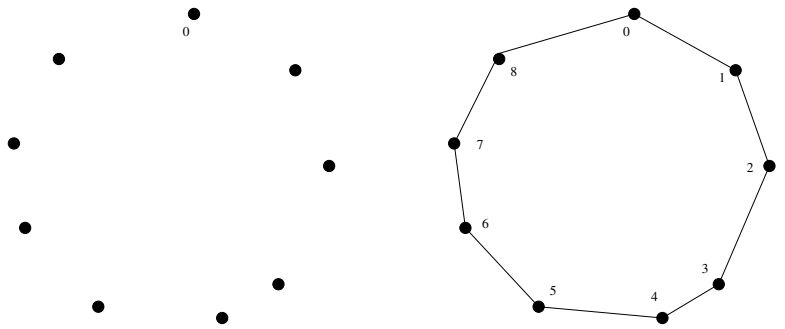


Figure 1.2: A good instance for the nearest-neighbor heuristic

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.

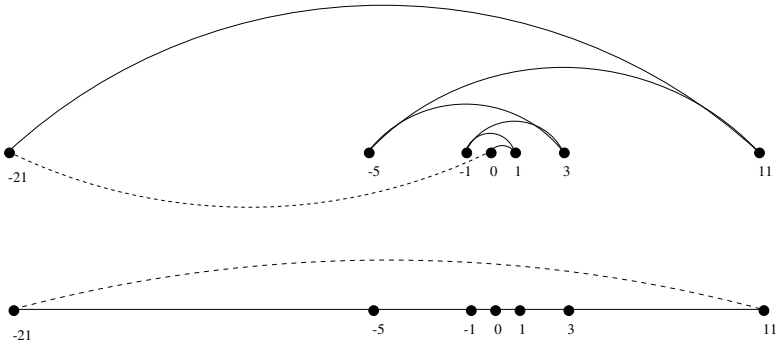


Figure 1.3: A bad instance for the nearest-neighbor heuristic, with the optimal solution

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 $dist(s, t) \leq d$ then $s_m = s$, $t_m = t$, and $d = dist(s, t)$
        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

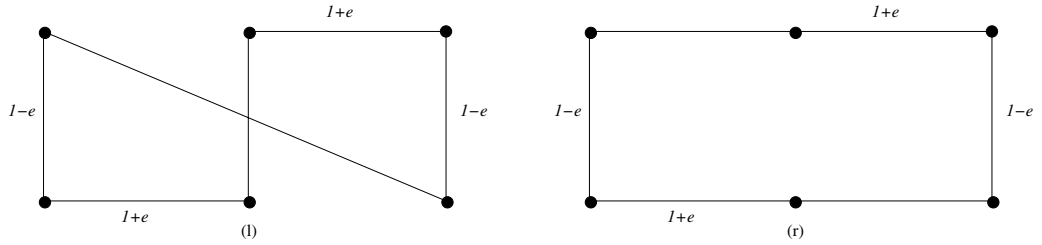


Figure 1.4: A bad instance for the closest-pair heuristic, with the optimal solution

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 = ∞
    For each of the n! permutations P_i of point set P
        If (cost(P_i) ≤ d) then d = 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).