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,

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

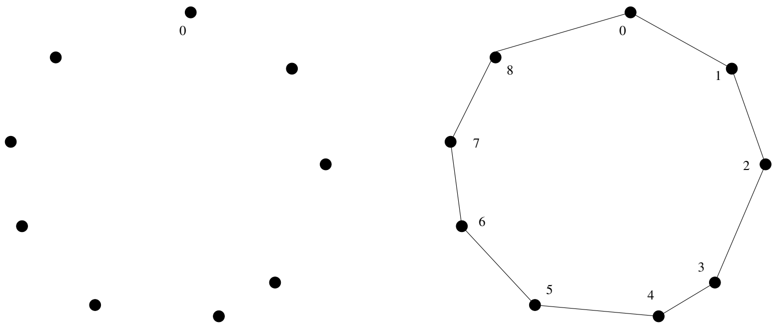

Figure 1.2: A good instance for the nearest-neighbor heuristic

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.

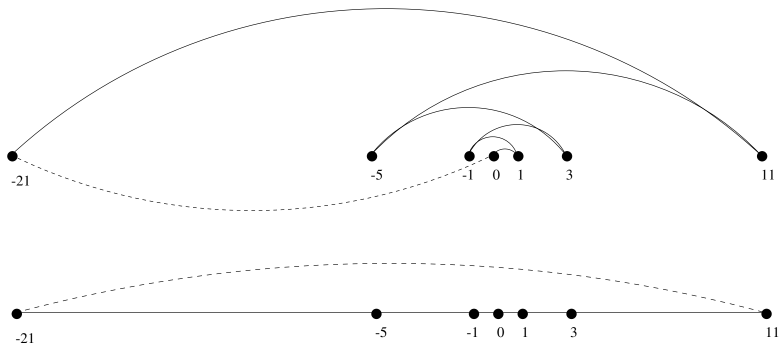

Figure 1.3: A bad instance for the nearest-neighbor heuristic, with the optimal solution

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 $dist(s, t) \leq d$ 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

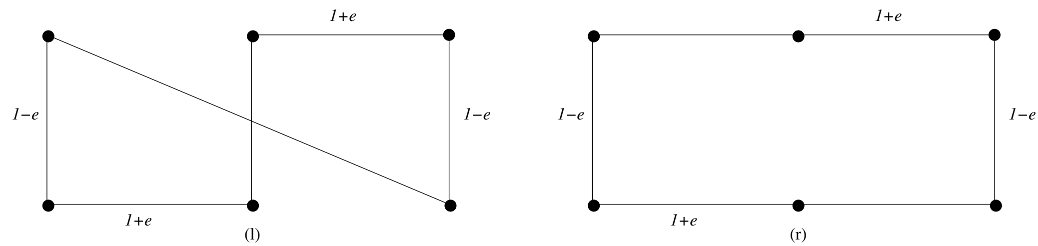

Figure 1.4: A bad instance for the closest-pair heuristic, with the optimal solution

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)
    $d = \infty$
    For each of the $n!$ permutations P_i of point set P
        If $(cost(P_i) \leq d)$ 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).

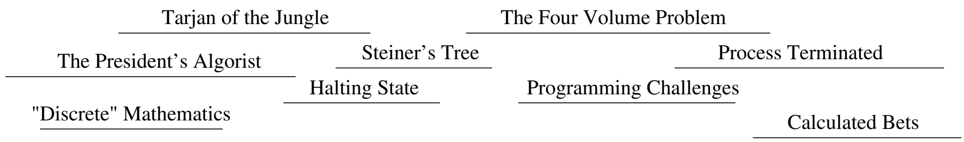

Figure 1.5: An instance of the non-overlapping movie scheduling problem

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 begining of this period.

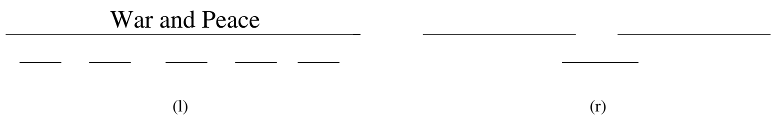

Figure 1.6: Bad instances for the (l) earliest job first and (r) shortest job first heuristics.

EarliestJobFirst(I)
    Accept the earlest 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 mutally non-overlapping) and ($size(S_i) > j$)
            then $j = size(S_i)$ 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 ($\blacksquare$) 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 cavallier 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

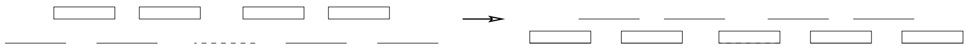

Figure 1.7: Large-scale changes in the optimal solution (boxes) after inserting a single interval (dashed) into the instance

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($2 \cdot Increment(\lfloor y/2 \rfloor)$)
        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 $Increment(\lfloor y/2 \rfloor)$. 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:

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

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

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

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 ith 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 $n \to \infty$. 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 $n \to \infty$.

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, $1 + 1/2 + 1/4 + 1/8 + \ldots \leq 2$ 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

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

$$= (n+1)! \times ((n+1)+1) - 1$$
$$= (n+2)! - 1$$

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."

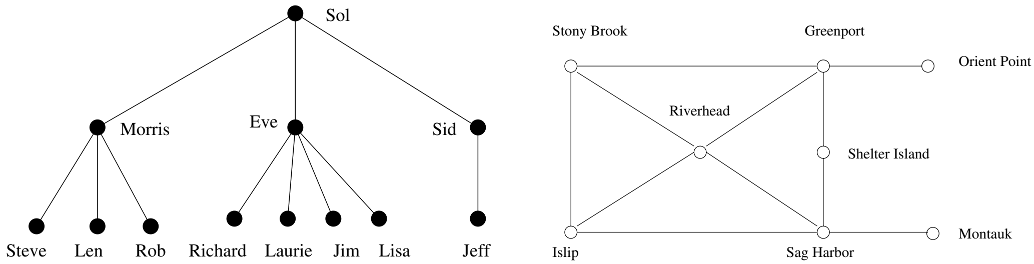

Figure 1.8: Modeling real-world structures with trees and graphs

- *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.

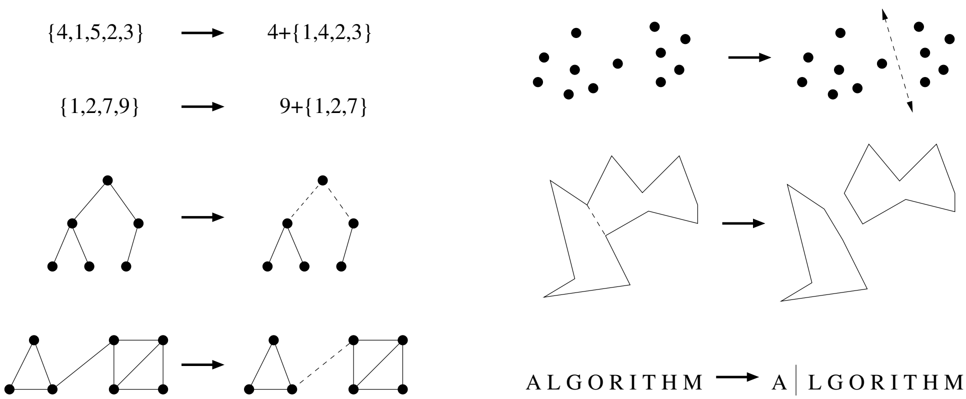

Figure 1.9: Recursive decompositions of combinatorial objects. (left column) Permutations, subsets, trees, and graphs. (right column) Point sets, polygons, and strings

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 $\{1, \ldots, n\}$ things and you get a permutation of the remaining $n - 1$ things. Permutations are recursive objects.

- *Subsets* – Every subset of the elements $\{1, \ldots, n\}$ contains a subset of $\{1, \ldots, n-1\}$ 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

[1]Yes, this is a true story.

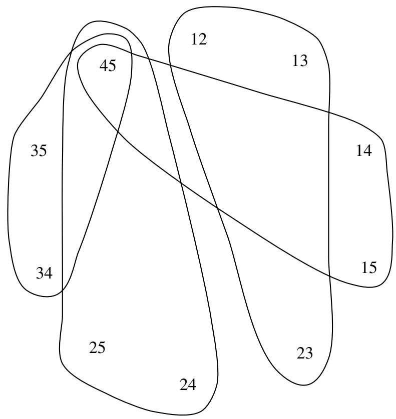

Figure 1.10: Covering all pairs of $\{1,2,3,4,5\}$ with tickets $\{1,2,3\}$, $\{1,4,5\}$, $\{2,4,5\}$, $\{3,4,5\}$

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:

- We needed the ability to generate all subsets of k numbers from the candidate set S. Algorithms for generating and ranking/unranking subsets of sets are presented in Section 14.5 (page 452).
- We needed the right formulation of what it meant to have a covering set of purchased tickets. The obvious criteria would be to pick a small set of tickets such that we have purchased at least one ticket containing each of the $\binom{n}{l}$ l-subsets of S that might pay off with the prize.
- We needed to keep track of which prize combinations we have thus far covered. We seek tickets to cover as many thus-far-uncovered prize combinations as possible. The currently covered combinations are a subset of all possible combinations. Data structures for subsets are discussed in Section 12.5 (page 385). The best candidate seemed to be a bit vector, which would answer in constant time "is this combination already covered?"
- We needed a search mechanism to decide which ticket to buy next. For small enough set sizes, we could do an exhaustive search over all possible subsets of tickets and pick the smallest one. For larger problems, a randomized search process like simulated annealing (see Section 7.5.3 (page 254)) would select tickets-to-buy to cover as many uncovered combinations as possible. By repeating this randomized procedure several times and picking the best solution, we would be likely to come up with a good set of tickets.

Excluding the details of the search mechanism, the pseudocode for the bookkeeping looked something like this:

LottoTicketSet(n, k, l)
    Initialize the $\binom{n}{l}$-element bit-vector V to all false
    While there exists a false entry in V
        Select a k-subset T of $\{1, \ldots, n\}$ as the next ticket to buy
        For each of the l-subsets T_i of T, $V[rank(T_i)] =$ true
    Report the set of tickets bought

The bright undergraduate, Fayyaz Younas, rose to the challenge. Based on this framework, he implemented a brute-force search algorithm and found optimal solutions for problems with $n \leq 5$ in a reasonable time. He implemented a random search procedure to solve larger problems, tweaking it for a while before settling on the best variant. Finally, the day arrived when we could call Lotto Systems Group and announce that we had solved the problem.

"Our program found an optimal solution for $n = 15$, $k = 6$, $j = 6$, $l = 3$ meant buying 28 tickets."

"Twenty-eight tickets!" complained the president. "You must have a bug. Look, these five tickets will suffice to cover everything *twice* over: $\{2, 4, 8, 10, 13, 14\}$, $\{4, 5, 7, 8, 12, 15\}$, $\{1, 2, 3, 6, 11, 13\}$, $\{3, 5, 6, 9, 10, 15\}$, $\{1, 7, 9, 11, 12, 14\}$."

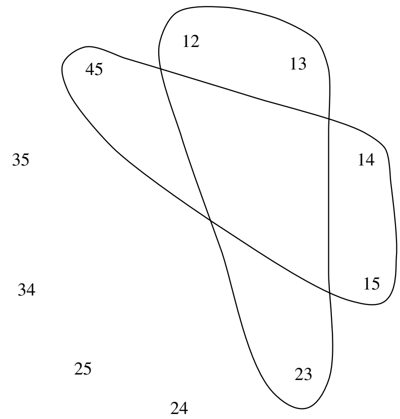

Figure 1.11: Guaranteeing a winning pair from $\{1, 2, 3, 4, 5\}$ using only tickets $\{1, 2, 3\}$ and $\{1, 4, 5\}$

We fiddled with this example for a while before admitting that he was right. *We hadn't modeled the problem correctly!* In fact, we didn't need to explicitly cover all possible winning combinations. Figure 1.11 illustrates the principle by giving a two-ticket solution to our previous four-ticket example. Such unpromising outcomes as $\{2, 3, 4\}$ and $\{3, 4, 5\}$ each agree in one matching pair with tickets from Figure 1.11. We were trying to cover too many combinations, and the penny-pinching psychics were unwilling to pay for such extravagance.

Fortunately, this story has a happy ending. The general outline of our search-based solution still holds for the real problem. All we must fix is which subsets we get credit for covering with a given set of tickets. After this modification, we obtained the kind of results they were hoping for. Lotto Systems Group gratefully accepted our program to incorporate into their product, and hopefully hit the jackpot with it.

The moral of this story is to make sure that you model the problem correctly before trying to solve it. In our case, we came up with a reasonable model, but didn't work hard enough to validate it before we started to program. Our misinterpretation would have become obvious had we worked out a small example by hand and bounced it off our sponsor before beginning work. Our success in recovering from this error is a tribute to the basic correctness of our initial formulation, and our use of well-defined abstractions for such tasks as (1) ranking/unranking k-subsets, (2) the set data structure, and (3) combinatorial search.

Chapter Notes

Every decent algorithm book reflects the design philosophy of its author. For students seeking alternative presentations and viewpoints, we particularly recommend the books of Corman, et. al [CLRS01], Kleinberg/Tardos [KT06], and Manber [Man89].

Formal proofs of algorithm correctness are important, and deserve a fuller discussion than we are able to provide in this chapter. See Gries [Gri89] for a thorough introduction to the techniques of program verification.

The movie scheduling problem represents a very special case of the general *independent set* problem, which is discussed in Section 16.2 (page 528). The restriction limits the allowable input instances to *interval* graphs, where the vertices of the graph G can be represented by intervals on the line and (i, j) is an edge of G iff the intervals overlap. Golumbic [Gol04] provides a full treatment of this interesting and important class of graphs.

Jon Bentley's *Programming Pearls* columns are probably the best known collection of algorithmic "war stories." Originally published in the *Communications of the ACM*, they have been collected in two books [Ben90, Ben99]. Brooks's *The Mythical Man Month* [Bro95] is another wonderful collection of war stories, focused more on software engineering than algorithm design, but they remain a source of considerable wisdom. Every programmer should read all these books, for pleasure as well as insight.

Our solution to the lotto ticket set covering problem is presented in more detail in [YS96].

1.7 Exercises

Finding Counterexamples

1-1. *[3]* Show that $a + b$ can be less than $\min(a, b)$.

1-2. *[3]* Show that $a \times b$ can be less than $\min(a, b)$.

1-3. *[5]* Design/draw a road network with two points a and b such that the fastest route between a and b is not the shortest route.

1-4. *[5]* Design/draw a road network with two points a and b such that the shortest route between a and b is not the route with the fewest turns.

1-5. *[4]* The *knapsack problem* is as follows: given a set of integers $S = \{s_1, s_2, \ldots, s_n\}$, and a target number T, find a subset of S which adds up exactly to T. For example, there exists a subset within $S = \{1, 2, 5, 9, 10\}$ that adds up to $T = 22$ but not $T = 23$.

Find counterexamples to each of the following algorithms for the knapsack problem. That is, giving an S and T such that the subset is selected using the algorithm does not leave the knapsack completely full, even though such a solution exists.

(a) Put the elements of S in the knapsack in left to right order if they fit, i.e. the first-fit algorithm.

(b) Put the elements of S in the knapsack from smallest to largest, i.e. the best-fit algorithm.

(c) Put the elements of S in the knapsack from largest to smallest.

1-6. *[5]* The *set cover problem* is as follows: given a set of subsets $S_1, ..., S_m$ of the universal set $U = \{1, ..., n\}$, find the smallest subset of subsets $T \subset S$ such that $\cup_{t_i \in T} t_i = U$. For example, there are the following subsets, $S_1 = \{1, 3, 5\}$, $S_2 = \{2, 4\}$, $S_3 = \{1, 4\}$, and $S_4 = \{2, 5\}$ The set cover would then be S_1 and S_2.

Find a counterexample for the following algorithm: Select the largest subset for the cover, and then delete all its elements from the universal set. Repeat by adding the subset containing the largest number of uncovered elements until all are covered.

Proofs of Correctness

1-7. *[3]* Prove the correctness of the following recursive algorithm to multiply two natural numbers, for all integer constants $c \geq 2$.

function multiply(y, z)
  comment Return the product yz.
  if $z = 0$ *then* return(0) *else*
  return(multiply$(cy, \lfloor z/c \rfloor) + y \cdot (z \bmod c)$)

1-8. *[3]* Prove the correctness of the following algorithm for evaluating a polynomial.
P(x) $= a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$

function horner(A, x)
  $p = A_n$
  for i from $n - 1$ to 0
    $p = p * x + A_i$
  return p

1-9. *[3]* Prove the correctness of the following sorting algorithm.

function bubblesort $(A : \text{list}[1 \ldots n])$
  var int i, j
  for i from n to 1
    for j from 1 to $i - 1$
      if $(A[j] > A[j + 1])$
        swap the values of $A[j]$ and $A[j + 1]$

Induction

1-10. *[3]* Prove that $\sum_{i=1}^{n} i = n(n + 1)/2$ for $n \geq 0$, by induction.

1-11. *[3]* Prove that $\sum_{i=1}^{n} i^2 = n(n + 1)(2n + 1)/6$ for $n \geq 0$, by induction.

1-12. *[3]* Prove that $\sum_{i=1}^{n} i^3 = n^2(n + 1)^2/4$ for $n \geq 0$, by induction.

1-13. *[3]* Prove that

$$\sum_{i=1}^{n} i(i + 1)(i + 2) = n(n + 1)(n + 2)(n + 3)/4$$

1-14. *[5]* Prove by induction on $n \geq 1$ that for every $a \neq 1$,

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

1-15. *[3]* Prove by induction that for $n \geq 1$,

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

1-16. *[3]* Prove by induction that $n^3 + 2n$ is divisible by 3 for all $n \geq 0$.

1-17. *[3]* Prove by induction that a tree with n vertices has exactly $n - 1$ edges.

1-18. *[3]* Prove by mathematical induction that the sum of the cubes of the first n positive integers is equal to the square of the sum of these integers, i.e.

$$\sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$$

Estimation

1-19. *[3]* Do all the books you own total at least one million pages? How many total pages are stored in your school library?

1-20. *[3]* How many words are there in this textbook?

1-21. *[3]* How many hours are one million seconds? How many days? Answer these questions by doing all arithmetic in your head.

1-22. *[3]* Estimate how many cities and towns there are in the United States.

1-23. *[3]* Estimate how many cubic miles of water flow out of the mouth of the Mississippi River each day. Do not look up any supplemental facts. Describe all assumptions you made in arriving at your answer.

1-24. *[3]* Is disk drive access time normally measured in milliseconds (thousandths of a second) or microseconds (millionths of a second)? Does your RAM memory access a word in more or less than a microsecond? How many instructions can your CPU execute in one year if the machine is left running all the time?

1-25. *[4]* A sorting algorithm takes 1 second to sort 1,000 items on your local machine. How long will it take to sort 10,000 items…

(a) if you believe that the algorithm takes time proportional to n^2, and

(b) if you believe that the algorithm takes time roughly proportional to $n \log n$?

Implementation Projects

1-26. *[5]* Implement the two TSP heuristics of Section 1.1 (page 5). Which of them gives better-quality solutions in practice? Can you devise a heuristic that works better than both of them?

1-27. *[5]* Describe how to test whether a given set of tickets establishes sufficient coverage in the Lotto problem of Section 1.6 (page 23). Write a program to find good ticket sets.

Interview Problems

1-28. *[5]* Write a function to perform integer division without using either the `/` or `*` operators. Find a fast way to do it.

1-29. *[5]* There are 25 horses. At most, 5 horses can race together at a time. You must determine the fastest, second fastest, and third fastest horses. Find the minimum number of races in which this can be done.

1-30. *[3]* How many piano tuners are there in the entire world?

1-31. *[3]* How many gas stations are there in the United States?

1-32. *[3]* How much does the ice in a hockey rink weigh?

1-33. *[3]* How many miles of road are there in the United States?

1-34. *[3]* On average, how many times would you have to flip open the Manhattan phone book at random in order to find a specific name?

Programming Challenges

These programming challenge problems with robot judging are available at *http://www.programming-challenges.com* or *http://online-judge.uva.es*.

1-1. "The $3n + 1$ Problem" – Programming Challenges 110101, UVA Judge 100.

1-2. "The Trip" – Programming Challenges 110103, UVA Judge 10137.

1-3. "Australian Voting" – Programming Challenges 110108, UVA Judge 10142.