Chapter 2

Basics of Algorithm Analysis

Analyzing algorithms involves thinking about how their resource requirements—the amount of time and space they use—will scale with increasing input size. We begin this chapter by talking about how to put this notion on a concrete footing, as making it concrete opens the door to a rich understanding of computational tractability. Having done this, we develop the mathematical machinery needed to talk about the way in which different functions scale with increasing input size, making precise what it means for one function to grow faster than another.

We then develop running-time bounds for some basic algorithms, beginning with an implementation of the Gale-Shapley algorithm from Chapter 1 and continuing to a survey of many different running times and certain characteristic types of algorithms that achieve these running times. In some cases, obtaining a good running-time bound relies on the use of more sophisticated data structures, and we conclude this chapter with a very useful example of such a data structure: priority queues and their implementation using heaps.

2.1 Computational Tractability

A major focus of this book is to find efficient algorithms for computational problems. At this level of generality, our topic seems to encompass the whole of computer science; so what is specific to our approach here?

First, we will try to identify broad themes and design principles in the development of algorithms. We will look for paradigmatic problems and approaches that illustrate, with a minimum of irrelevant detail, the basic approaches to designing efficient algorithms. At the same time, it would be pointless to pursue these design principles in a vacuum, so the problems and

approaches we consider are drawn from fundamental issues that arise throughout computer science, and a general study of algorithms turns out to serve as a nice survey of computational ideas that arise in many areas.

Another property shared by many of the problems we study is their fundamentally *discrete* nature. That is, like the Stable Matching Problem, they will involve an implicit search over a large set of combinatorial possibilities; and the goal will be to efficiently find a solution that satisfies certain clearly delineated conditions.

As we seek to understand the general notion of computational efficiency, we will focus primarily on efficiency in running time: we want algorithms that run quickly. But it is important that algorithms be efficient in their use of other resources as well. In particular, the amount of *space* (or memory) used by an algorithm is an issue that will also arise at a number of points in the book, and we will see techniques for reducing the amount of space needed to perform a computation.

Some Initial Attempts at Defining Efficiency

The first major question we need to answer is the following: How should we turn the fuzzy notion of an "efficient" algorithm into something more concrete?

A first attempt at a working definition of *efficiency* is the following.

> Proposed Definition of Efficiency (1): *An algorithm is efficient if, when implemented, it runs quickly on real input instances.*

Let's spend a little time considering this definition. At a certain level, it's hard to argue with: one of the goals at the bedrock of our study of algorithms is solving real problems quickly. And indeed, there is a significant area of research devoted to the careful implementation and profiling of different algorithms for discrete computational problems.

But there are some crucial things missing from this definition, even if our main goal is to solve real problem instances quickly on real computers. The first is the omission of *where*, and *how well*, we implement an algorithm. Even bad algorithms can run quickly when applied to small test cases on extremely fast processors; even good algorithms can run slowly when they are coded sloppily. Also, what is a "real" input instance? We don't know the full range of input instances that will be encountered in practice, and some input instances can be much harder than others. Finally, this proposed definition above does not consider how well, or badly, an algorithm may *scale* as problem sizes grow to unexpected levels. A common situation is that two very different algorithms will perform comparably on inputs of size 100; multiply the input size tenfold, and one will still run quickly while the other consumes a huge amount of time.

So what we could ask for is a concrete definition of efficiency that is platform-independent, instance-independent, and of predictive value with respect to increasing input sizes. Before focusing on any specific consequences of this claim, we can at least explore its implicit, high-level suggestion: that we need to take a more mathematical view of the situation.

We can use the Stable Matching Problem as an example to guide us. The input has a natural "size" parameter N; we could take this to be the total size of the representation of all preference lists, since this is what any algorithm for the problem will receive as input. N is closely related to the other natural parameter in this problem: n, the number of men and the number of women. Since there are $2n$ preference lists, each of length n, we can view $N = 2n^2$, suppressing more fine-grained details of how the data is represented. In considering the problem, we will seek to describe an algorithm at a high level, and then analyze its running time mathematically as a function of this input size N.

Worst-Case Running Times and Brute-Force Search

To begin with, we will focus on analyzing the *worst-case* running time: we will look for a bound on the largest possible running time the algorithm could have over all inputs of a given size N, and see how this scales with N. The focus on worst-case performance initially seems quite draconian: what if an algorithm performs well on most instances and just has a few pathological inputs on which it is very slow? This certainly is an issue in some cases, but in general the worst-case analysis of an algorithm has been found to do a reasonable job of capturing its efficiency in practice. Moreover, once we have decided to go the route of mathematical analysis, it is hard to find an effective alternative to worst-case analysis. Average-case analysis—the obvious appealing alternative, in which one studies the performance of an algorithm averaged over "random" instances—can sometimes provide considerable insight, but very often it can also become a quagmire. As we observed earlier, it's very hard to express the full range of input instances that arise in practice, and so attempts to study an algorithm's performance on "random" input instances can quickly devolve into debates over how a random input should be generated: the same algorithm can perform very well on one class of random inputs and very poorly on another. After all, real inputs to an algorithm are generally not being produced from a random distribution, and so average-case analysis risks telling us more about the means by which the random inputs were generated than about the algorithm itself.

So in general we will think about the worst-case analysis of an algorithm's running time. But what is a reasonable analytical benchmark that can tell us whether a running-time bound is impressive or weak? A first simple guide

is by comparison with brute-force search over the search space of possible solutions.

Let's return to the example of the Stable Matching Problem. Even when the size of a Stable Matching input instance is relatively small, the *search space* it defines is enormous (there are $n!$ possible perfect matchings between n men and n women), and we need to find a matching that is stable. The natural "brute-force" algorithm for this problem would plow through all perfect matchings by enumeration, checking each to see if it is stable. The surprising punchline, in a sense, to our solution of the Stable Matching Problem is that we needed to spend time proportional only to N in finding a stable matching from among this stupendously large space of possibilities. This was a conclusion we reached at an *analytical level*. We did not implement the algorithm and try it out on sample preference lists; we reasoned about it mathematically. Yet, at the same time, our analysis indicated how the algorithm could be implemented in practice and gave fairly conclusive evidence that it would be a big improvement over exhaustive enumeration.

This will be a common theme in most of the problems we study: a compact representation, implicitly specifying a giant search space. For most of these problems, there will be an obvious brute-force solution: try all possibilities and see if any one of them works. Not only is this approach almost always too slow to be useful, it is an intellectual cop-out; it provides us with absolutely no insight into the structure of the problem we are studying. And so if there is a common thread in the algorithms we emphasize in this book, it would be the following alternative definition of efficiency.

> Proposed Definition of Efficiency (2): *An algorithm is efficient if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search.*

This will turn out to be a very useful working definition for us. Algorithms that improve substantially on brute-force search nearly always contain a valuable heuristic idea that makes them work; and they tell us something about the intrinsic structure, and computational tractability, of the underlying problem itself.

But if there is a problem with our second working definition, it is vagueness. What do we mean by "qualitatively better performance?" This suggests that we consider the actual running time of algorithms more carefully, and try to quantify what a reasonable running time would be.

Polynomial Time as a Definition of Efficiency

When people first began analyzing discrete algorithms mathematically—a thread of research that began gathering momentum through the 1960s—

a consensus began to emerge on how to quantify the notion of a "reasonable" running time. Search spaces for natural combinatorial problems tend to grow exponentially in the size N of the input; if the input size increases by one, the number of possibilities increases multiplicatively. We'd like a good algorithm for such a problem to have a better scaling property: when the input size increases by a constant factor—say, a factor of 2—the algorithm should only slow down by some constant factor C.

Arithmetically, we can formulate this scaling behavior as follows. Suppose an algorithm has the following property: There are absolute constants $c > 0$ and $d > 0$ so that on every input instance of size N, its running time is bounded by cN^d primitive computational steps. (In other words, its running time is at most proportional to N^d.) For now, we will remain deliberately vague on what we mean by the notion of a "primitive computational step"—but it can be easily formalized in a model where each step corresponds to a single assembly-language instruction on a standard processor, or one line of a standard programming language such as C or Java. In any case, if this running-time bound holds, for some c and d, then we say that the algorithm has a *polynomial running time*, or that it is a *polynomial-time algorithm*. Note that any polynomial-time bound has the scaling property we're looking for. If the input size increases from N to $2N$, the bound on the running time increases from cN^d to $c(2N)^d = c \cdot 2^d N^d$, which is a slow-down by a factor of 2^d. Since d is a constant, so is 2^d; of course, as one might expect, lower-degree polynomials exhibit better scaling behavior than higher-degree polynomials.

From this notion, and the intuition expressed above, emerges our third attempt at a working definition of efficiency.

Proposed Definition of Efficiency (3): *An algorithm is efficient if it has a polynomial running time.*

Where our previous definition seemed overly vague, this one seems much too prescriptive. Wouldn't an algorithm with running time proportional to n^{100}—and hence polynomial—be hopelessly inefficient? Wouldn't we be relatively pleased with a nonpolynomial running time of $n^{1+.02(\log n)}$? The answers are, of course, "yes" and "yes." And indeed, however much one may try to abstractly motivate the definition of efficiency in terms of polynomial time, a primary justification for it is this: *It really works.* Problems for which polynomial-time algorithms exist almost invariably turn out to have algorithms with running times proportional to very moderately growing polynomials like n, $n \log n$, n^2, or n^3. Conversely, problems for which no polynomial-time algorithm is known tend to be very difficult in practice. There are certainly exceptions to this principle in both directions: there are cases, for example, in

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

which an algorithm with exponential worst-case behavior generally runs well on the kinds of instances that arise in practice; and there are also cases where the best polynomial-time algorithm for a problem is completely impractical due to large constants or a high exponent on the polynomial bound. All this serves to reinforce the point that our emphasis on worst-case, polynomial-time bounds is only an abstraction of practical situations. But overwhelmingly, the concrete mathematical definition of polynomial time has turned out to correspond surprisingly well in practice to what we observe about the efficiency of algorithms, and the tractability of problems, in real life.

One further reason why the mathematical formalism and the empirical evidence seem to line up well in the case of polynomial-time solvability is that the gulf between the growth rates of polynomial and exponential functions is enormous. Suppose, for example, that we have a processor that executes a million high-level instructions per second, and we have algorithms with running-time bounds of n, $n \log_2 n$, n^2, n^3, 1.5^n, 2^n, and $n!$. In Table 2.1, we show the running times of these algorithms (in seconds, minutes, days, or years) for inputs of size $n = 10, 30, 50, 100, 1{,}000, 10{,}000, 100{,}000,$ and 1,000,000.

There is a final, fundamental benefit to making our definition of efficiency so specific: it becomes negatable. It becomes possible to express the notion that *there is no efficient algorithm for a particular problem*. In a sense, being able to do this is a prerequisite for turning our study of algorithms into good science, for it allows us to ask about the existence or nonexistence of efficient algorithms as a well-defined question. In contrast, both of our

previous definitions were completely subjective, and hence limited the extent to which we could discuss certain issues in concrete terms.

In particular, the first of our definitions, which was tied to the specific implementation of an algorithm, turned efficiency into a moving target: as processor speeds increase, more and more algorithms fall under this notion of efficiency. Our definition in terms of polynomial time is much more an absolute notion; it is closely connected with the idea that each problem has an intrinsic level of computational tractability: some admit efficient solutions, and others do not.

2.2 Asymptotic Order of Growth

Our discussion of computational tractability has turned out to be intrinsically based on our ability to express the notion that an algorithm's worst-case running time on inputs of size n grows at a rate that is at most proportional to some function $f(n)$. The function $f(n)$ then becomes a bound on the running time of the algorithm. We now discuss a framework for talking about this concept.

We will mainly express algorithms in the pseudo-code style that we used for the Gale-Shapley algorithm. At times we will need to become more formal, but this style of specifying algorithms will be completely adequate for most purposes. When we provide a bound on the running time of an algorithm, we will generally be counting the number of such pseudo-code steps that are executed; in this context, one *step* will consist of assigning a value to a variable, looking up an entry in an array, following a pointer, or performing an arithmetic operation on a fixed-size integer.

When we seek to say something about the running time of an algorithm on inputs of size n, one thing we could aim for would be a very concrete statement such as, "On any input of size n, the algorithm runs for at most $1.62n^2 + 3.5n + 8$ steps." This may be an interesting statement in some contexts, but as a general goal there are several things wrong with it. First, getting such a precise bound may be an exhausting activity, and more detail than we wanted anyway. Second, because our ultimate goal is to identify broad classes of algorithms that have similar behavior, we'd actually like to classify running times at a coarser level of granularity so that similarities among different algorithms, and among different problems, show up more clearly. And finally, extremely detailed statements about the number of steps an algorithm executes are often—in a strong sense—meaningless. As just discussed, we will generally be counting steps in a pseudo-code specification of an algorithm that resembles a high-level programming language. Each one of these steps will typically unfold into some fixed number of primitive steps when the program is compiled into

an intermediate representation, and then into some further number of steps depending on the particular architecture being used to do the computing. So the most we can safely say is that as we look at different levels of computational abstraction, the notion of a "step" may grow or shrink by a constant factor—for example, if it takes 25 low-level machine instructions to perform one operation in our high-level language, then our algorithm that took at most $1.62n^2 + 3.5n + 8$ steps can also be viewed as taking $40.5n^2 + 87.5n + 200$ steps when we analyze it at a level that is closer to the actual hardware.

O, Ω, and Θ

For all these reasons, we want to express the growth rate of running times and other functions in a way that is insensitive to constant factors and low-order terms. In other words, we'd like to be able to take a running time like the one we discussed above, $1.62n^2 + 3.5n + 8$, and say that it grows like n^2, up to constant factors. We now discuss a precise way to do this.

Asymptotic Upper Bounds Let $T(n)$ be a function—say, the worst-case running time of a certain algorithm on an input of size n. (We will assume that all the functions we talk about here take nonnegative values.) Given another function $f(n)$, we say that $T(n)$ *is* $O(f(n))$ (read as "$T(n)$ is order $f(n)$") if, for sufficiently large n, the function $T(n)$ is bounded above by a constant multiple of $f(n)$. We will also sometimes write this as $T(n) = O(f(n))$. More precisely, $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ so that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$. In this case, we will say that T is *asymptotically upper-bounded by* f. It is important to note that this definition requires a constant c to exist that works for *all* n; in particular, c cannot depend on n.

As an example of how this definition lets us express upper bounds on running times, consider an algorithm whose running time (as in the earlier discussion) has the form $T(n) = pn^2 + qn + r$ for positive constants p, q, and r. We'd like to claim that any such function is $O(n^2)$. To see why, we notice that for all $n \geq 1$, we have $qn \leq qn^2$, and $r \leq rn^2$. So we can write

$$T(n) = pn^2 + qn + r \leq pn^2 + qn^2 + rn^2 = (p + q + r)n^2$$

for all $n \geq 1$. This inequality is exactly what the definition of $O(\cdot)$ requires: $T(n) \leq cn^2$, where $c = p + q + r$.

Note that $O(\cdot)$ expresses only an upper bound, not the exact growth rate of the function. For example, just as we claimed that the function $T(n) = pn^2 + qn + r$ is $O(n^2)$, it's also correct to say that it's $O(n^3)$. Indeed, we just argued that $T(n) \leq (p + q + r)n^2$, and since we also have $n^2 \leq n^3$, we can conclude that $T(n) \leq (p + q + r)n^3$ as the definition of $O(n^3)$ requires. The fact that a function can have many upper bounds is not just a trick of the notation; it shows up in the analysis of running times as well. There are cases

where an algorithm has been proved to have running time $O(n^3)$; some years pass, people analyze the same algorithm more carefully, and they show that in fact its running time is $O(n^2)$. There was nothing wrong with the first result; it was a correct upper bound. It's simply that it wasn't the "tightest" possible running time.

Asymptotic Lower Bounds There is a complementary notation for lower bounds. Often when we analyze an algorithm—say we have just proven that its worst-case running time $T(n)$ is $O(n^2)$—we want to show that this upper bound is the best one possible. To do this, we want to express the notion that for arbitrarily large input sizes n, the function $T(n)$ is *at least* a constant multiple of some specific function $f(n)$. (In this example, $f(n)$ happens to be n^2.) Thus, we say that $T(n)$ is $\Omega(f(n))$ (also written $T(n) = \Omega(f(n))$) if there exist constants $\epsilon > 0$ and $n_0 \geq 0$ so that for all $n \geq n_0$, we have $T(n) \geq \epsilon \cdot f(n)$. By analogy with $O(\cdot)$ notation, we will refer to T in this case as being *asymptotically lower-bounded by* f. Again, note that the constant ϵ must be fixed, independent of n.

This definition works just like $O(\cdot)$, except that we are bounding the function $T(n)$ from below, rather than from above. For example, returning to the function $T(n) = pn^2 + qn + r$, where p, q, and r are positive constants, let's claim that $T(n) = \Omega(n^2)$. Whereas establishing the upper bound involved "inflating" the terms in $T(n)$ until it looked like a constant times n^2, now we need to do the opposite: we need to reduce the size of $T(n)$ until it looks like a constant times n^2. It is not hard to do this; for all $n \geq 0$, we have

$$T(n) = pn^2 + qn + r \geq pn^2,$$

which meets what is required by the definition of $\Omega(\cdot)$ with $\epsilon = p > 0$.

Just as we discussed the notion of "tighter" and "weaker" upper bounds, the same issue arises for lower bounds. For example, it is correct to say that our function $T(n) = pn^2 + qn + r$ is $\Omega(n)$, since $T(n) \geq pn^2 \geq pn$.

Asymptotically Tight Bounds If we can show that a running time $T(n)$ is both $O(f(n))$ and also $\Omega(f(n))$, then in a natural sense we've found the "right" bound: $T(n)$ grows exactly like $f(n)$ to within a constant factor. This, for example, is the conclusion we can draw from the fact that $T(n) = pn^2 + qn + r$ is both $O(n^2)$ and $\Omega(n^2)$.

There is a notation to express this: if a function $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$, we say that $T(n)$ is $\Theta(f(n))$. In this case, we say that $f(n)$ is an *asymptotically tight bound* for $T(n)$. So, for example, our analysis above shows that $T(n) = pn^2 + qn + r$ is $\Theta(n^2)$.

Asymptotically tight bounds on worst-case running times are nice things to find, since they characterize the worst-case performance of an algorithm

precisely up to constant factors. And as the definition of $\Theta(\cdot)$ shows, one can obtain such bounds by closing the gap between an upper bound and a lower bound. For example, sometimes you will read a (slightly informally phrased) sentence such as "An upper bound of $O(n^3)$ has been shown on the worst-case running time of the algorithm, but there is no example known on which the algorithm runs for more than $\Omega(n^2)$ steps." This is implicitly an invitation to search for an asymptotically tight bound on the algorithm's worst-case running time.

Sometimes one can also obtain an asymptotically tight bound directly by computing a limit as n goes to infinity. Essentially, if the ratio of functions $f(n)$ and $g(n)$ converges to a positive constant as n goes to infinity, then $f(n) = \Theta(g(n))$.

(2.1) *Let f and g be two functions that*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists and is equal to some number $c > 0$. Then $f(n) = \Theta(g(n))$.

Proof. We will use the fact that the limit exists and is positive to show that $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, as required by the definition of $\Theta(\cdot)$.

Since

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0,$$

it follows from the definition of a limit that there is some n_0 beyond which the ratio is always between $\frac{1}{2}c$ and $2c$. Thus, $f(n) \leq 2cg(n)$ for all $n \geq n_0$, which implies that $f(n) = O(g(n))$; and $f(n) \geq \frac{1}{2}cg(n)$ for all $n \geq n_0$, which implies that $f(n) = \Omega(g(n))$. ■

Properties of Asymptotic Growth Rates

Having seen the definitions of O, Ω, and Θ, it is useful to explore some of their basic properties.

Transitivity A first property is *transitivity*: if a function f is asymptotically upper-bounded by a function g, and if g in turn is asymptotically upper-bounded by a function h, then f is asymptotically upper-bounded by h. A similar property holds for lower bounds. We write this more precisely as follows.

(2.2)

(a) If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.

(b) If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.

Proof. We'll prove part (a) of this claim; the proof of part (b) is very similar.

For (a), we're given that for some constants c and n_0, we have $f(n) \leq cg(n)$ for all $n \geq n_0$. Also, for some (potentially different) constants c' and n_0', we have $g(n) \leq c'h(n)$ for all $n \geq n_0'$. So consider any number n that is at least as large as both n_0 and n_0'. We have $f(n) \leq cg(n) \leq cc'h(n)$, and so $f(n) \leq cc'h(n)$ for all $n \geq \max(n_0, n_0')$. This latter inequality is exactly what is required for showing that $f = O(h)$. ■

Combining parts (a) and (b) of (2.2), we can obtain a similar result for asymptotically tight bounds. Suppose we know that $f = \Theta(g)$ and that $g = \Theta(h)$. Then since $f = O(g)$ and $g = O(h)$, we know from part (a) that $f = O(h)$; since $f = \Omega(g)$ and $g = \Omega(h)$, we know from part (b) that $f = \Omega(h)$. It follows that $f = \Theta(h)$. Thus we have shown

(2.3) *If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.*

Sums of Functions It is also useful to have results that quantify the effect of adding two functions. First, if we have an asymptotic upper bound that applies to each of two functions f and g, then it applies to their sum.

(2.4) *Suppose that f and g are two functions such that for some other function h, we have $f = O(h)$ and $g = O(h)$. Then $f + g = O(h)$.*

Proof. We're given that for some constants c and n_0, we have $f(n) \leq ch(n)$ for all $n \geq n_0$. Also, for some (potentially different) constants c' and n_0', we have $g(n) \leq c'h(n)$ for all $n \geq n_0'$. So consider any number n that is at least as large as both n_0 and n_0'. We have $f(n) + g(n) \leq ch(n) + c'h(n)$. Thus $f(n) + g(n) \leq (c + c')h(n)$ for all $n \geq \max(n_0, n_0')$, which is exactly what is required for showing that $f + g = O(h)$. ■

There is a generalization of this to sums of a fixed constant number of functions k, where k may be larger than two. The result can be stated precisely as follows; we omit the proof, since it is essentially the same as the proof of (2.4), adapted to sums consisting of k terms rather than just two.

(2.5) *Let k be a fixed constant, and let $f_1, f_2, \ldots, f_k$ and h be functions such that $f_i = O(h)$ for all i. Then $f_1 + f_2 + \cdots + f_k = O(h)$.*

There is also a consequence of (2.4) that covers the following kind of situation. It frequently happens that we're analyzing an algorithm with two high-level parts, and it is easy to show that one of the two parts is slower than the other. We'd like to be able to say that the running time of the whole algorithm is asymptotically comparable to the running time of the slow part. Since the overall running time is a sum of two functions (the running times of

the two parts), results on asymptotic bounds for sums of functions are directly relevant.

(2.6) *Suppose that f and g are two functions (taking nonnegative values) such that $g = O(f)$. Then $f + g = \Theta(f)$. In other words, f is an asymptotically tight bound for the combined function $f + g$.*

Proof. Clearly $f + g = \Omega(f)$, since for all $n \geq 0$, we have $f(n) + g(n) \geq f(n)$. So to complete the proof, we need to show that $f + g = O(f)$.

But this is a direct consequence of (2.4): we're given the fact that $g = O(f)$, and also $f = O(f)$ holds for any function, so by (2.4) we have $f + g = O(f)$. ■

This result also extends to the sum of any fixed, constant number of functions: the most rapidly growing among the functions is an asymptotically tight bound for the sum.

Asymptotic Bounds for Some Common Functions

There are a number of functions that come up repeatedly in the analysis of algorithms, and it is useful to consider the asymptotic properties of some of the most basic of these: polynomials, logarithms, and exponentials.

Polynomials Recall that a polynomial is a function that can be written in the form $f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d$ for some integer constant $d > 0$, where the final coefficient a_d is nonzero. This value d is called the *degree* of the polynomial. For example, the functions of the form $pn^2 + qn + r$ (with $p \neq 0$) that we considered earlier are polynomials of degree 2.

A basic fact about polynomials is that their asymptotic rate of growth is determined by their "high-order term"—the one that determines the degree. We state this more formally in the following claim. Since we are concerned here only with functions that take nonnegative values, we will restrict our attention to polynomials for which the high-order term has a positive coefficient $a_d > 0$.

(2.7) *Let f be a polynomial of degree d, in which the coefficient a_d is positive. Then $f = O(n^d)$.*

Proof. We write $f = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d$, where $a_d > 0$. The upper bound is a direct application of (2.5). First, notice that coefficients a_j for $j < d$ may be negative, but in any case we have $a_j n^j \leq |a_j| n^d$ for all $n \geq 1$. Thus each term in the polynomial is $O(n^d)$. Since f is a sum of a constant number of functions, each of which is $O(n^d)$, it follows from (2.5) that f is $O(n^d)$. ■

One can also show that under the conditions of (2.7), we have $f = \Omega(n^d)$, and hence it follows that in fact $f = \Theta(n^d)$.

This is a good point at which to discuss the relationship between these types of asymptotic bounds and the notion of *polynomial time*, which we arrived at in the previous section as a way to formalize the more elusive concept of efficiency. Using $O(\cdot)$ notation, it's easy to formally define polynomial time: a *polynomial-time algorithm* is one whose running time $T(n)$ is $O(n^d)$ for some constant d, where d is independent of the input size.

So algorithms with running-time bounds like $O(n^2)$ and $O(n^3)$ are polynomial-time algorithms. But it's important to realize that an algorithm can be polynomial time even if its running time is not written as n raised to some integer power. To begin with, a number of algorithms have running times of the form $O(n^x)$ for some number x that is not an integer. For example, in Chapter 5 we will see an algorithm whose running time is $O(n^{1.59})$; we will also see exponents less than 1, as in bounds like $O(\sqrt{n}) = O(n^{1/2})$.

To take another common kind of example, we will see many algorithms whose running times have the form $O(n \log n)$. Such algorithms are also polynomial time: as we will see next, $\log n \leq n$ for all $n \geq 1$, and hence $n \log n \leq n^2$ for all $n \geq 1$. In other words, if an algorithm has running time $O(n \log n)$, then it also has running time $O(n^2)$, and so it is a polynomial-time algorithm.

Logarithms Recall that $\log_b n$ is the number x such that $b^x = n$. One way to get an approximate sense of how fast $\log_b n$ grows is to note that, if we round it down to the nearest integer, it is one less than the number of digits in the base-b representation of the number n. (Thus, for example, $1 + \log_2 n$, rounded down, is the number of bits needed to represent n.)

So logarithms are very slowly growing functions. In particular, for every base b, the function $\log_b n$ is asymptotically bounded by every function of the form n^x, even for (noninteger) values of x arbitrary close to 0.

(2.8) *For every $b > 1$ and every $x > 0$, we have $\log_b n = O(n^x)$.*

One can directly translate between logarithms of different bases using the following fundamental identity:

$$\log_a n = \frac{\log_b n}{\log_b a}.$$

This equation explains why you'll often notice people writing bounds like $O(\log n)$ without indicating the base of the logarithm. This is not sloppy usage: the identity above says that $\log_a n = \frac{1}{\log_b a} \cdot \log_b n$, so the point is that $\log_a n = \Theta(\log_b n)$, and the base of the logarithm is not important when writing bounds using asymptotic notation.

Exponentials Exponential functions are functions of the form $f(n) = r^n$ for some constant base r. Here we will be concerned with the case in which $r > 1$, which results in a very fast-growing function.

In particular, where polynomials raise n to a fixed exponent, exponentials raise a fixed number to n as a power; this leads to much faster rates of growth. One way to summarize the relationship between polynomials and exponentials is as follows.

(2.9) *For every $r > 1$ and every $d > 0$, we have $n^d = O(r^n)$.*

In particular, every exponential grows faster than every polynomial. And as we saw in Table 2.1, when you plug in actual values of n, the differences in growth rates are really quite impressive.

Just as people write $O(\log n)$ without specifying the base, you'll also see people write "The running time of this algorithm is exponential," without specifying which exponential function they have in mind. Unlike the liberal use of $\log n$, which is justified by ignoring constant factors, this generic use of the term "exponential" is somewhat sloppy. In particular, for different bases $r > s > 1$, it is never the case that $r^n = \Theta(s^n)$. Indeed, this would require that for some constant $c > 0$, we would have $r^n \leq cs^n$ for all sufficiently large n. But rearranging this inequality would give $(r/s)^n \leq c$ for all sufficiently large n. Since $r > s$, the expression $(r/s)^n$ is tending to infinity with n, and so it cannot possibly remain bounded by a fixed constant c.

So asymptotically speaking, exponential functions are all different. Still, it's usually clear what people intend when they inexactly write "The running time of this algorithm is exponential"—they typically mean that the running time grows at least as fast as *some* exponential function, and all exponentials grow so fast that we can effectively dismiss this algorithm without working out further details of the exact running time. This is not entirely fair. Occasionally there's more going on with an exponential algorithm than first appears, as we'll see, for example, in Chapter 10; but as we argued in the first section of this chapter, it's a reasonable rule of thumb.

Taken together, then, logarithms, polynomials, and exponentials serve as useful landmarks in the range of possible functions that you encounter when analyzing running times. Logarithms grow more slowly than polynomials, and polynomials grow more slowly than exponentials.

2.3 Implementing the Stable Matching Algorithm Using Lists and Arrays

We've now seen a general approach for expressing bounds on the running time of an algorithm. In order to asymptotically analyze the running time of

an algorithm expressed in a high-level fashion—as we expressed the Gale-Shapley Stable Matching algorithm in Chapter 1, for example—one doesn't have to actually program, compile, and execute it, but one does have to think about how the data will be represented and manipulated in an implementation of the algorithm, so as to bound the number of computational steps it takes.

The implementation of basic algorithms using data structures is something that you probably have had some experience with. In this book, data structures will be covered in the context of implementing specific algorithms, and so we will encounter different data structures based on the needs of the algorithms we are developing. To get this process started, we consider an implementation of the Gale-Shapley Stable Matching algorithm; we showed earlier that the algorithm terminates in at most n^2 iterations, and our implementation here provides a corresponding worst-case running time of $O(n^2)$, counting actual computational steps rather than simply the total number of iterations. To get such a bound for the Stable Matching algorithm, we will only need to use two of the simplest data structures: *lists* and *arrays*. Thus, our implementation also provides a good chance to review the use of these basic data structures as well.

In the Stable Matching Problem, each man and each woman has a ranking of all members of the opposite gender. The very first question we need to discuss is how such a ranking will be represented. Further, the algorithm maintains a matching and will need to know at each step which men and women are free, and who is matched with whom. In order to implement the algorithm, we need to decide which data structures we will use for all these things.

An important issue to note here is that the choice of data structure is up to the algorithm designer; for each algorithm we will choose data structures that make it efficient and easy to implement. In some cases, this may involve *preprocessing* the input to convert it from its given input representation into a data structure that is more appropriate for the problem being solved.

Arrays and Lists

To start our discussion we will focus on a single list, such as the list of women in order of preference by a single man. Maybe the simplest way to keep a list of n elements is to use an array A of length n, and have $A[i]$ be the i^{th} element of the list. Such an array is simple to implement in essentially all standard programming languages, and it has the following properties.

- We can answer a query of the form "What is the i^{th} element on the list?" in $O(1)$ time, by a direct access to the value $A[i]$.
- If we want to determine whether a particular element e belongs to the list (i.e., whether it is equal to $A[i]$ for some i), we need to check the

elements one by one in $O(n)$ time, assuming we don't know anything about the order in which the elements appear in A.

- If the array elements are sorted in some clear way (either numerically or alphabetically), then we can determine whether an element e belongs to the list in $O(\log n)$ time using *binary search*; we will not need to use binary search for any part of our stable matching implementation, but we will have more to say about it in the next section.

An array is less good for dynamically maintaining a list of elements that changes over time, such as the list of free men in the Stable Matching algorithm; since men go from being free to engaged, and potentially back again, a list of free men needs to grow and shrink during the execution of the algorithm. It is generally cumbersome to frequently add or delete elements to a list that is maintained as an array.

An alternate, and often preferable, way to maintain such a dynamic set of elements is via a linked list. In a linked list, the elements are sequenced together by having each element point to the next in the list. Thus, for each element v on the list, we need to maintain a pointer to the next element; we set this pointer to *null* if i is the last element. We also have a pointer `First` that points to the first element. By starting at `First` and repeatedly following pointers to the next element until we reach *null*, we can thus traverse the entire contents of the list in time proportional to its length.

A generic way to implement such a linked list, when the set of possible elements may not be fixed in advance, is to allocate a record e for each element that we want to include in the list. Such a record would contain a field e.`val` that contains the value of the element, and a field e.`Next` that contains a pointer to the next element in the list. We can create a *doubly linked list*, which is traversable in both directions, by also having a field e.`Prev` that contains a pointer to the previous element in the list. (e.`Prev` = *null* if e is the first element.) We also include a pointer `Last`, analogous to `First`, that points to the last element in the list. A schematic illustration of part of such a list is shown in the first line of Figure 2.1.

A doubly linked list can be modified as follows.

- *Deletion.* To delete the element e from a doubly linked list, we can just "splice it out" by having the previous element, referenced by e.`Prev`, and the next element, referenced by e.`Next`, point directly to each other. The deletion operation is illustrated in Figure 2.1.
- *Insertion.* To insert element e between elements d and f in a list, we "splice it in" by updating d.`Next` and f.`Prev` to point to e, and the `Next` and `Prev` pointers of e to point to d and f, respectively. This operation is

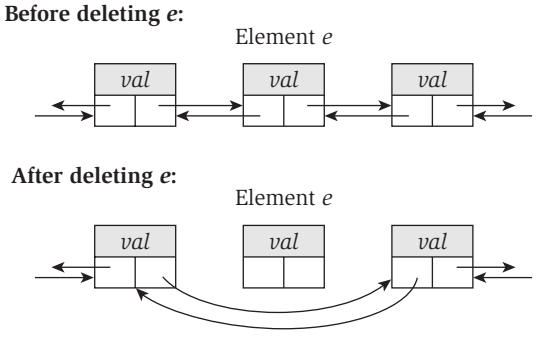

Figure 2.1 A schematic representation of a doubly linked list, showing the deletion of an element e.

essentially the reverse of deletion, and indeed one can see this operation at work by reading Figure 2.1 from bottom to top.

Inserting or deleting e at the beginning of the list involves updating the `First` pointer, rather than updating the record of the element before e.

While lists are good for maintaining a dynamically changing set, they also have disadvantages. Unlike arrays, we cannot find the i^{th} element of the list in $O(1)$ time: to find the i^{th} element, we have to follow the `Next` pointers starting from the beginning of the list, which takes a total of $O(i)$ time.

Given the relative advantages and disadvantages of arrays and lists, it may happen that we receive the input to a problem in one of the two formats and want to convert it into the other. As discussed earlier, such preprocessing is often useful; and in this case, it is easy to convert between the array and list representations in $O(n)$ time. This allows us to freely choose the data structure that suits the algorithm better and not be constrained by the way the information is given as input.

Implementing the Stable Matching Algorithm

Next we will use arrays and linked lists to implement the Stable Matching algorithm from Chapter 1. We have already shown that the algorithm terminates in at most n^2 iterations, and this provides a type of upper bound on the running time. However, if we actually want to implement the G-S algorithm so that it runs in time proportional to n^2, we need to be able to implement each iteration in constant time. We discuss how to do this now.

For simplicity, assume that the set of men and women are both $\{1, \ldots, n\}$. To ensure this, we can order the men and women (say, alphabetically), and associate number i with the i^{th} man m_i or i^{th} women w_i in this order. This

assumption (or notation) allows us to define an array indexed by all men or all women. We need to have a preference list for each man and for each woman. To do this we will have two arrays, one for women's preference lists and one for the men's preference lists; we will use $\texttt{ManPref}[m, i]$ to denote the i^{th} woman on man m's preference list, and similarly $\texttt{WomanPref}[w, i]$ to be the i^{th} man on the preference list of woman w. Note that the amount of space needed to give the preferences for all $2n$ individuals is $O(n^2)$, as each person has a list of length n.

We need to consider each step of the algorithm and understand what data structure allows us to implement it efficiently. Essentially, we need to be able to do each of four things in constant time.

1. We need to be able to identify a free man.
2. We need, for a man m, to be able to identify the highest-ranked woman to whom he has not yet proposed.
3. For a woman w, we need to decide if w is currently engaged, and if she is, we need to identify her current partner.
4. For a woman w and two men m and m', we need to be able to decide, again in constant time, which of m or m' is preferred by w.

First, consider selecting a free man. We will do this by maintaining the set of free men as a linked list. When we need to select a free man, we take the first man m on this list. We delete m from the list if he becomes engaged, and possibly insert a different man m', if some other man m' becomes free. In this case, m' can be inserted at the front of the list, again in constant time.

Next, consider a man m. We need to identify the highest-ranked woman to whom he has not yet proposed. To do this we will need to maintain an extra array $\texttt{Next}$ that indicates for each man m the position of the next woman he will propose to on his list. We initialize $\texttt{Next}[m] = 1$ for all men m. If a man m needs to propose to a woman, he'll propose to $w = \texttt{ManPref}[m,\texttt{Next}[m]]$, and once he proposes to w, we increment the value of $\texttt{Next}[m]$ by one, regardless of whether or not w accepts the proposal.

Now assume man m proposes to woman w; we need to be able to identify the man m' that w is engaged to (if there is such a man). We can do this by maintaining an array $\texttt{Current}$ of length n, where $\texttt{Current}[w]$ is the woman w's current partner m'. We set $\texttt{Current}[w]$ to a special null symbol when we need to indicate that woman w is not currently engaged; at the start of the algorithm, $\texttt{Current}[w]$ is initialized to this null symbol for all women w.

To sum up, the data structures we have set up thus far can implement the operations (1)–(3) in $O(1)$ time each.

Maybe the trickiest question is how to maintain women's preferences to keep step (4) efficient. Consider a step of the algorithm, when man m proposes to a woman w. Assume w is already engaged, and her current partner is $m' =$`Current`$[w]$. We would like to decide in $O(1)$ time if woman w prefers m or m'. Keeping the women's preferences in an array `WomanPref`, analogous to the one we used for men, does not work, as we would need to walk through w's list one by one, taking $O(n)$ time to find m and m' on the list. While $O(n)$ is still polynomial, we can do a lot better if we build an auxiliary data structure at the beginning.

At the start of the algorithm, we create an $n \times n$ array `Ranking`, where `Ranking`$[w, m]$ contains the rank of man m in the sorted order of w's preferences. By a single pass through w's preference list, we can create this array in linear time for each woman, for a total initial time investment proportional to n^2. Then, to decide which of m or m' is preferred by w, we simply compare the values `Ranking`$[w, m]$ and `Ranking`$[w, m']$.

This allows us to execute step (4) in constant time, and hence we have everything we need to obtain the desired running time.

(2.10) *The data structures described above allow us to implement the G-S algorithm in $O(n^2)$ time.*

2.4 A Survey of Common Running Times

When trying to analyze a new algorithm, it helps to have a rough sense of the "landscape" of different running times. Indeed, there are styles of analysis that recur frequently, and so when one sees running-time bounds like $O(n)$, $O(n \log n)$, and $O(n^2)$ appearing over and over, it's often for one of a very small number of distinct reasons. Learning to recognize these common styles of analysis is a long-term goal. To get things under way, we offer the following survey of common running-time bounds and some of the typical approaches that lead to them.

Earlier we discussed the notion that most problems have a natural "search space"—the set of all possible solutions—and we noted that a unifying theme in algorithm design is the search for algorithms whose performance is more efficient than a brute-force enumeration of this search space. In approaching a new problem, then, it often helps to think about two kinds of bounds: one on the running time you hope to achieve, and the other on the size of the problem's natural search space (and hence on the running time of a brute-force algorithm for the problem). The discussion of running times in this section will begin in many cases with an analysis of the brute-force algorithm, since it is a useful

way to get one's bearings with respect to a problem; the task of improving on such algorithms will be our goal in most of the book.

Linear Time

An algorithm that runs in $O(n)$, or linear, time has a very natural property: its running time is at most a constant factor times the size of the input. One basic way to get an algorithm with this running time is to process the input in a single pass, spending a constant amount of time on each item of input encountered. Other algorithms achieve a linear time bound for more subtle reasons. To illustrate some of the ideas here, we consider two simple linear-time algorithms as examples.

Computing the Maximum Computing the maximum of n numbers, for example, can be performed in the basic "one-pass" style. Suppose the numbers are provided as input in either a list or an array. We process the numbers $a_1, a_2, \ldots, a_n$ in order, keeping a running estimate of the maximum as we go. Each time we encounter a number a_i, we check whether a_i is larger than our current estimate, and if so we update the estimate to a_i.

```
max = a_1
For i = 2 to n
  If a_i > max then
    set max = a_i
  Endif
Endfor
```

In this way, we do constant work per element, for a total running time of $O(n)$.

Sometimes the constraints of an application force this kind of one-pass algorithm on you—for example, an algorithm running on a high-speed switch on the Internet may see a stream of packets flying past it, and it can try computing anything it wants to as this stream passes by, but it can only perform a constant amount of computational work on each packet, and it can't save the stream so as to make subsequent scans through it. Two different subareas of algorithms, *online algorithms* and *data stream algorithms*, have developed to study this model of computation.

Merging Two Sorted Lists Often, an algorithm has a running time of $O(n)$, but the reason is more complex. We now describe an algorithm for merging two sorted lists that stretches the one-pass style of design just a little, but still has a linear running time.

Suppose we are given two lists of n numbers each, $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, and each is already arranged in ascending order. We'd like to

merge these into a single list $c_1, c_2, \ldots, c_{2n}$ that is also arranged in ascending order. For example, merging the lists 2, 3, 11, 19 and 4, 9, 16, 25 results in the output 2, 3, 4, 9, 11, 16, 19, 25.

To do this, we could just throw the two lists together, ignore the fact that they're separately arranged in ascending order, and run a sorting algorithm. But this clearly seems wasteful; we'd like to make use of the existing order in the input. One way to think about designing a better algorithm is to imagine performing the merging of the two lists by hand: suppose you're given two piles of numbered cards, each arranged in ascending order, and you'd like to produce a single ordered pile containing all the cards. If you look at the top card on each stack, you know that the smaller of these two should go first on the output pile; so you could remove this card, place it on the output, and now iterate on what's left.

In other words, we have the following algorithm.

```
To merge sorted lists A = a_1, ..., a_n and B = b_1, ..., b_n:
  Maintain a Current pointer into each list, initialized to
    point to the front elements
  While both lists are nonempty:
    Let a_i and b_j be the elements pointed to by the Current pointer
    Append the smaller of these two to the output list
    Advance the Current pointer in the list from which the
      smaller element was selected
  EndWhile
  Once one list is empty, append the remainder of the other list
      to the output
```

See Figure 2.2 for a picture of this process.

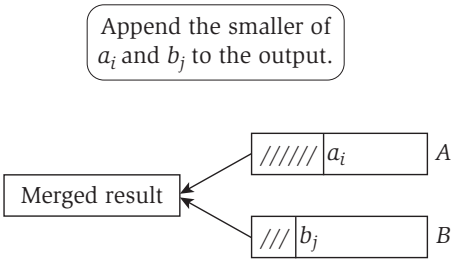

Figure 2.2 To merge sorted lists A and B, we repeatedly extract the smaller item from the front of the two lists and append it to the output.

Now, to show a linear-time bound, one is tempted to describe an argument like what worked for the maximum-finding algorithm: "We do constant work per element, for a total running time of $O(n)$." But it is actually not true that we do only constant work per element. Suppose that n is an even number, and consider the lists $A = 1, 3, 5, \ldots, 2n-1$ and $B = n, n+2, n+4, \ldots, 3n-2$. The number b_1 at the front of list B will sit at the front of the list for $n/2$ iterations while elements from A are repeatedly being selected, and hence it will be involved in $\Omega(n)$ comparisons. Now, it is true that each element can be involved in at most $O(n)$ comparisons (at worst, it is compared with each element in the other list), and if we sum this over all elements we get a running-time bound of $O(n^2)$. This is a correct bound, but we can show something much stronger.

The better way to argue is to bound the number of iterations of the `While` loop by an "accounting" scheme. Suppose we *charge* the cost of each iteration to the element that is selected and added to the output list. An element can be charged only once, since at the moment it is first charged, it is added to the output and never seen again by the algorithm. But there are only $2n$ elements total, and the cost of each iteration is accounted for by a charge to some element, so there can be at most $2n$ iterations. Each iteration involves a constant amount of work, so the total running time is $O(n)$, as desired.

While this merging algorithm iterated through its input lists in order, the "interleaved" way in which it processed the lists necessitated a slightly subtle running-time analysis. In Chapter 3 we will see linear-time algorithms for graphs that have an even more complex flow of control: they spend a constant amount of time on each node and edge in the underlying graph, but the order in which they process the nodes and edges depends on the structure of the graph.

$O(n \log n)$ Time

$O(n \log n)$ is also a very common running time, and in Chapter 5 we will see one of the main reasons for its prevalence: it is the running time of any algorithm that splits its input into two equal-sized pieces, solves each piece recursively, and then combines the two solutions in linear time.

Sorting is perhaps the most well-known example of a problem that can be solved this way. Specifically, the *Mergesort* algorithm divides the set of input numbers into two equal-sized pieces, sorts each half recursively, and then merges the two sorted halves into a single sorted output list. We have just seen that the merging can be done in linear time; and Chapter 5 will discuss how to analyze the recursion so as to get a bound of $O(n \log n)$ on the overall running time.

One also frequently encounters $O(n \log n)$ as a running time simply because there are many algorithms whose most expensive step is to sort the input. For example, suppose we are given a set of n time-stamps $x_1, x_2, \ldots, x_n$ on which copies of a file arrived at a server, and we'd like to find the largest interval of time between the first and last of these time-stamps during which no copy of the file arrived. A simple solution to this problem is to first sort the time-stamps $x_1, x_2, \ldots, x_n$ and then process them in sorted order, determining the sizes of the gaps between each number and its successor in ascending order. The largest of these gaps is the desired subinterval. Note that this algorithm requires $O(n \log n)$ time to sort the numbers, and then it spends constant work on each number in ascending order. In other words, the remainder of the algorithm after sorting follows the basic recipe for linear time that we discussed earlier.

Quadratic Time

Here's a basic problem: suppose you are given n points in the plane, each specified by (x, y) coordinates, and you'd like to find the pair of points that are closest together. The natural brute-force algorithm for this problem would enumerate all pairs of points, compute the distance between each pair, and then choose the pair for which this distance is smallest.

What is the running time of this algorithm? The number of pairs of points is $\binom{n}{2} = \frac{n(n-1)}{2}$, and since this quantity is bounded by $\frac{1}{2}n^2$, it is $O(n^2)$. More crudely, the number of pairs is $O(n^2)$ because we multiply the number of ways of choosing the first member of the pair (at most n) by the number of ways of choosing the second member of the pair (also at most n). The distance between points (x_i, y_i) and (x_j, y_j) can be computed by the formula $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ in constant time, so the overall running time is $O(n^2)$. This example illustrates a very common way in which a running time of $O(n^2)$ arises: performing a search over all pairs of input items and spending constant time per pair.

Quadratic time also arises naturally from a pair of *nested loops*: An algorithm consists of a loop with $O(n)$ iterations, and each iteration of the loop launches an internal loop that takes $O(n)$ time. Multiplying these two factors of n together gives the running time.

The brute-force algorithm for finding the closest pair of points can be written in an equivalent way with two nested loops:

```
For each input point (x_i, y_i)
  For each other input point (x_j, y_j)
    Compute distance d = sqrt((x_i − x_j)^2 + (y_i − y_j)^2)
```

```
    If d is less than the current minimum, update minimum to d
  Endfor
Endfor
```

Note how the "inner" loop, over (x_j, y_j), has $O(n)$ iterations, each taking constant time; and the "outer" loop, over (x_i, y_i), has $O(n)$ iterations, each invoking the inner loop once.

It's important to notice that the algorithm we've been discussing for the Closest-Pair Problem really is just the brute-force approach: the natural search space for this problem has size $O(n^2)$, and we're simply enumerating it. At first, one feels there is a certain inevitability about this quadratic algorithm—we have to measure all the distances, don't we?—but in fact this is an illusion. In Chapter 5 we describe a very clever algorithm that finds the closest pair of points in the plane in only $O(n \log n)$ time, and in Chapter 13 we show how randomization can be used to reduce the running time to $O(n)$.

Cubic Time

More elaborate sets of nested loops often lead to algorithms that run in $O(n^3)$ time. Consider, for example, the following problem. We are given sets $S_1, S_2, \ldots, S_n$, each of which is a subset of $\{1, 2, \ldots, n\}$, and we would like to know whether some pair of these sets is disjoint—in other words, has no elements in common.

What is the running time needed to solve this problem? Let's suppose that each set S_i is represented in such a way that the elements of S_i can be listed in constant time per element, and we can also check in constant time whether a given number p belongs to S_i. The following is a direct way to approach the problem.

```
For pair of sets Si and Sj
  Determine whether Si and Sj have an element in common
Endfor
```

This is a concrete algorithm, but to reason about its running time it helps to open it up (at least conceptually) into three nested loops.

```
For each set Si
  For each other set Sj
    For each element p of Si
      Determine whether p also belongs to Sj
    Endfor
    If no element of Si belongs to Sj then
```

```
      Report that S_i and S_j are disjoint
    Endif
  Endfor
Endfor
```

Each of the sets has maximum size $O(n)$, so the innermost loop takes time $O(n)$. Looping over the sets S_j involves $O(n)$ iterations around this innermost loop; and looping over the sets S_i involves $O(n)$ iterations around this. Multiplying these three factors of n together, we get the running time of $O(n^3)$.

For this problem, there are algorithms that improve on $O(n^3)$ running time, but they are quite complicated. Furthermore, it is not clear whether the improved algorithms for this problem are practical on inputs of reasonable size.

$O(n^k)$ Time

In the same way that we obtained a running time of $O(n^2)$ by performing brute-force search over all pairs formed from a set of n items, we obtain a running time of $O(n^k)$ for any constant k when we search over all subsets of size k.

Consider, for example, the problem of finding independent sets in a graph, which we discussed in Chapter 1. Recall that a set of nodes is independent if no two are joined by an edge. Suppose, in particular, that for some fixed constant k, we would like to know if a given n-node input graph G has an independent set of size k. The natural brute-force algorithm for this problem would enumerate all subsets of k nodes, and for each subset S it would check whether there is an edge joining any two members of S. That is,

```
For each subset S of k nodes
  Check whether S constitutes an independent set
  If S is an independent set then
    Stop and declare success
  Endif
Endfor
If no k-node independent set was found then
  Declare failure
Endif
```

To understand the running time of this algorithm, we need to consider two quantities. First, the total number of k-element subsets in an n-element set is

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!}.$$

Since we are treating k as a constant, this quantity is $O(n^k)$. Thus, the outer loop in the algorithm above will run for $O(n^k)$ iterations as it tries all k-node subsets of the n nodes of the graph.

Inside this loop, we need to test whether a given set S of k nodes constitutes an independent set. The definition of an independent set tells us that we need to check, for each pair of nodes, whether there is an edge joining them. Hence this is a search over pairs, like we saw earlier in the discussion of quadratic time; it requires looking at $\binom{k}{2}$, that is, $O(k^2)$, pairs and spending constant time on each.

Thus the total running time is $O(k^2 n^k)$. Since we are treating k as a constant here, and since constants can be dropped in $O(\cdot)$ notation, we can write this running time as $O(n^k)$.

Independent Set is a principal example of a problem believed to be computationally hard, and in particular it is believed that no algorithm to find k-node independent sets in arbitrary graphs can avoid having some dependence on k in the exponent. However, as we will discuss in Chapter 10 in the context of a related problem, even once we've conceded that brute-force search over k-element subsets is necessary, there can be different ways of going about this that lead to significant differences in the efficiency of the computation.

Beyond Polynomial Time

The previous example of the Independent Set Problem starts us rapidly down the path toward running times that grow faster than any polynomial. In particular, two kinds of bounds that come up very frequently are 2^n and $n!$, and we now discuss why this is so.

Suppose, for example, that we are given a graph and want to find an independent set of *maximum* size (rather than testing for the existence of one with a given number of nodes). Again, people don't know of algorithms that improve significantly on brute-force search, which in this case would look as follows.

```
For each subset S of nodes
  Check whether S constitutes an independent set
  If S is a larger independent set than the largest seen so far then
    Record the size of S as the current maximum
  Endif
Endfor
```

This is very much like the brute-force algorithm for k-node independent sets, except that now we are iterating over *all* subsets of the graph. The total number

of subsets of an n-element set is 2^n, and so the outer loop in this algorithm will run for 2^n iterations as it tries all these subsets. Inside the loop, we are checking all pairs from a set S that can be as large as n nodes, so each iteration of the loop takes at most $O(n^2)$ time. Multiplying these two together, we get a running time of $O(n^2 2^n)$.

Thus see that 2^n arises naturally as a running time for a search algorithm that must consider all subsets. In the case of Independent Set, something at least nearly this inefficient appears to be necessary; but it's important to keep in mind that 2^n is the size of the search space for many problems, and for many of them we will be able to find highly efficient polynomial-time algorithms. For example, a brute-force search algorithm for the Interval Scheduling Problem that we saw in Chapter 1 would look very similar to the algorithm above: try all subsets of intervals, and find the largest subset that has no overlaps. But in the case of the Interval Scheduling Problem, as opposed to the Independent Set Problem, we will see (in Chapter 4) how to find an optimal solution in $O(n \log n)$ time. This is a recurring kind of dichotomy in the study of algorithms: two algorithms can have very similar-looking search spaces, but in one case you're able to bypass the brute-force search algorithm, and in the other you aren't.

The function $n!$ grows even more rapidly than 2^n, so it's even more menacing as a bound on the performance of an algorithm. Search spaces of size $n!$ tend to arise for one of two reasons. First, $n!$ is the number of ways to match up n items with n other items—for example, it is the number of possible perfect matchings of n men with n women in an instance of the Stable Matching Problem. To see this, note that there are n choices for how we can match up the first man; having eliminated this option, there are $n-1$ choices for how we can match up the second man; having eliminated these two options, there are $n-2$ choices for how we can match up the third man; and so forth. Multiplying all these choices out, we get $n(n-1)(n-2)\cdots(2)(1) = n!$

Despite this enormous set of possible solutions, we were able to solve the Stable Matching Problem in $O(n^2)$ iterations of the proposal algorithm. In Chapter 7, we will see a similar phenomenon for the Bipartite Matching Problem we discussed earlier; if there are n nodes on each side of the given bipartite graph, there can be up to $n!$ ways of pairing them up. However, by a fairly subtle search algorithm, we will be able to find the largest bipartite matching in $O(n^3)$ time.

The function $n!$ also arises in problems where the search space consists of all ways to arrange n items in order. A basic problem in this genre is the Traveling Salesman Problem: given a set of n cities, with distances between all pairs, what is the shortest tour that visits all cities? We assume that the salesman starts and ends at the first city, so the crux of the problem is the

implicit search over all orders of the remaining $n - 1$ cities, leading to a search space of size $(n - 1)!$. In Chapter 8, we will see that Traveling Salesman is another problem that, like Independent Set, belongs to the class of NP-complete problems and is believed to have no efficient solution.

Sublinear Time

Finally, there are cases where one encounters running times that are asymptotically smaller than linear. Since it takes linear time just to read the input, these situations tend to arise in a model of computation where the input can be "queried" indirectly rather than read completely, and the goal is to minimize the amount of querying that must be done.

Perhaps the best-known example of this is the binary search algorithm. Given a sorted array A of n numbers, we'd like to determine whether a given number p belongs to the array. We could do this by reading the entire array, but we'd like to do it much more efficiently, taking advantage of the fact that the array is sorted, by carefully *probing* particular entries. In particular, we probe the middle entry of A and get its value—say it is q—and we compare q to p. If $q = p$, we're done. If $q > p$, then in order for p to belong to the array A, it must lie in the lower half of A; so we ignore the upper half of A from now on and recursively apply this search in the lower half. Finally, if $q < p$, then we apply the analogous reasoning and recursively search in the upper half of A.

The point is that in each step, there's a region of A where p might possibly be; and we're shrinking the size of this region by a factor of two with every probe. So how large is the "active" region of A after k probes? It starts at size n, so after k probes it has size at most $(\frac{1}{2})^k n$.

Given this, how long will it take for the size of the active region to be reduced to a constant? We need k to be large enough so that $(\frac{1}{2})^k = O(1/n)$, and to do this we can choose $k = \log_2 n$. Thus, when $k = \log_2 n$, the size of the active region has been reduced to a constant, at which point the recursion bottoms out and we can search the remainder of the array directly in constant time.

So the running time of binary search is $O(\log n)$, because of this successive shrinking of the search region. In general, $O(\log n)$ arises as a time bound whenever we're dealing with an algorithm that does a constant amount of work in order to throw away a constant *fraction* of the input. The crucial fact is that $O(\log n)$ such iterations suffice to shrink the input down to constant size, at which point the problem can generally be solved directly.

2.5 A More Complex Data Structure: Priority Queues

Our primary goal in this book was expressed at the outset of the chapter: we seek algorithms that improve qualitatively on brute-force search, and in general we use polynomial-time solvability as the concrete formulation of this. Typically, achieving a polynomial-time solution to a nontrivial problem is not something that depends on fine-grained implementation details; rather, the difference between exponential and polynomial is based on overcoming higher-level obstacles. Once one has an efficient algorithm to solve a problem, however, it is often possible to achieve further improvements in running time by being careful with the implementation details, and sometimes by using more complex data structures.

Some complex data structures are essentially tailored for use in a single kind of algorithm, while others are more generally applicable. In this section, we describe one of the most broadly useful sophisticated data structures, the *priority queue.* Priority queues will be useful when we describe how to implement some of the graph algorithms developed later in the book. For our purposes here, it is a useful illustration of the analysis of a data structure that, unlike lists and arrays, must perform some nontrivial processing each time it is invoked.

The Problem

In the implementation of the Stable Matching algorithm in Section 2.3, we discussed the need to maintain a dynamically changing set S (such as the set of all free men in that case). In such situations, we want to be able to add elements to and delete elements from the set S, and we want to be able to select an element from S when the algorithm calls for it. A priority queue is designed for applications in which elements have a *priority value,* or *key,* and each time we need to select an element from S, we want to take the one with highest priority.

A priority queue is a data structure that maintains a set of elements S, where each element $v \in S$ has an associated value $\texttt{key}(v)$ that denotes the priority of element v; smaller keys represent higher priorities. Priority queues support the addition and deletion of elements from the set, and also the selection of the element with smallest key. Our implementation of priority queues will also support some additional operations that we summarize at the end of the section.

A motivating application for priority queues, and one that is useful to keep in mind when considering their general function, is the problem of managing

real-time events such as the scheduling of processes on a computer. Each process has a priority, or urgency, but processes do not arrive in order of their priorities. Rather, we have a current set of active processes, and we want to be able to extract the one with the currently highest priority and run it. We can maintain the set of processes in a priority queue, with the key of a process representing its priority value. Scheduling the highest-priority process corresponds to selecting the element with minimum key from the priority queue; concurrent with this, we will also be inserting new processes as they arrive, according to their priority values.

How efficiently do we hope to be able to execute the operations in a priority queue? We will show how to implement a priority queue containing at most n elements at any time so that elements can be added and deleted, and the element with minimum key selected, in $O(\log n)$ time per operation.

Before discussing the implementation, let us point out a very basic application of priority queues that highlights why $O(\log n)$ time per operation is essentially the "right" bound to aim for.

(2.11) *A sequence of $O(n)$ priority queue operations can be used to sort a set of n numbers.*

Proof. Set up a priority queue H, and insert each number into H with its value as a key. Then extract the smallest number one by one until all numbers have been extracted; this way, the numbers will come out of the priority queue in sorted order. ■

Thus, with a priority queue that can perform insertion and the extraction of minima in $O(\log n)$ per operation, we can sort n numbers in $O(n \log n)$ time. It is known that, in a comparison-based model of computation (when each operation accesses the input only by comparing a pair of numbers), the time needed to sort must be at least proportional to $n \log n$, so (2.11) highlights a sense in which $O(\log n)$ time per operation is the best we can hope for. We should note that the situation is a bit more complicated than this: implementations of priority queues more sophisticated than the one we present here can improve the running time needed for certain operations, and add extra functionality. But (2.11) shows that any sequence of priority queue operations that results in the sorting of n numbers must take time at least proportional to $n \log n$ in total.

A Data Structure for Implementing a Priority Queue

We will use a data structure called a *heap* to implement a priority queue. Before we discuss the structure of heaps, we should consider what happens with some simpler, more natural approaches to implementing the functions

of a priority queue. We could just have the elements in a list, and separately have a pointer labeled `Min` to the one with minimum key. This makes adding new elements easy, but extraction of the minimum hard. Specifically, finding the minimum is quick—we just consult the `Min` pointer—but after removing this minimum element, we need to update the `Min` pointer to be ready for the next operation, and this would require a scan of all elements in $O(n)$ time to find the new minimum.

This complication suggests that we should perhaps maintain the elements in the sorted order of the keys. This makes it easy to extract the element with smallest key, but now how do we add a new element to our set? Should we have the elements in an array, or a linked list? Suppose we want to add s with key value $\texttt{key}(s)$. If the set S is maintained as a sorted array, we can use binary search to find the array position where s should be inserted in $O(\log n)$ time, but to insert s in the array, we would have to move all later elements one position to the right. This would take $O(n)$ time. On the other hand, if we maintain the set as a sorted doubly linked list, we could insert it in $O(1)$ time into any position, but the doubly linked list would not support binary search, and hence we may need up to $O(n)$ time to find the position where s should be inserted.

The Definition of a Heap So in all these simple approaches, at least one of the operations can take up to $O(n)$ time—much more than the $O(\log n)$ per operation that we're hoping for. This is where heaps come in. The *heap* data structure combines the benefits of a sorted array and list for purposes of this application. Conceptually, we think of a heap as a balanced binary tree as shown on the left of Figure 2.3. The tree will have a root, and each node can have up to two children, a left and a right child. The keys in such a binary tree are said to be in *heap order* if the key of any element is at least as large as the key of the element at its parent node in the tree. In other words,

> *Heap order: For every element v, at a node i, the element w at i's parent satisfies* $\texttt{key}(w) \leq \texttt{key}(v)$.

In Figure 2.3 the numbers in the nodes are the keys of the corresponding elements.

Before we discuss how to work with a heap, we need to consider what data structure should be used to represent it. We can use pointers: each node at the heap could keep the element it stores, its key, and three pointers pointing to the two children and the parent of the heap node. We can avoid using pointers, however, if a bound N is known in advance on the total number of elements that will ever be in the heap at any one time. Such heaps can be maintained in an array H indexed by $i = 1, \ldots, N$. We will think of the heap nodes as corresponding to the positions in this array. $H[1]$ is the root, and for any node

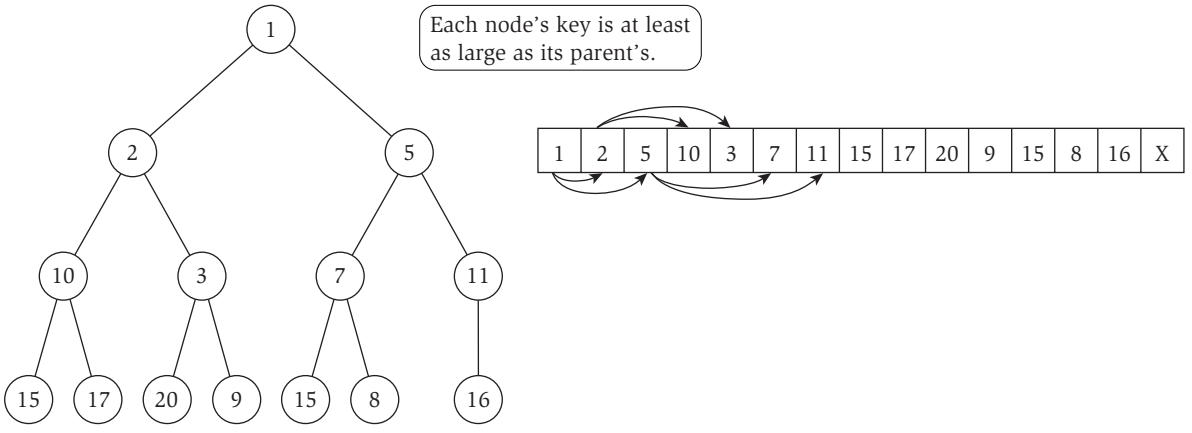

Figure 2.3 Values in a heap shown as a binary tree on the left, and represented as an array on the right. The arrows show the children for the top three nodes in the tree.

at position i, the children are the nodes at positions $\texttt{leftChild}(i) = 2i$ and $\texttt{rightChild}(i) = 2i + 1$. So the two children of the root are at positions 2 and 3, and the parent of a node at position i is at position $\texttt{parent}(i) = \lfloor i/2 \rfloor$. If the heap has $n < N$ elements at some time, we will use the first n positions of the array to store the n heap elements, and use $\texttt{length}(H)$ to denote the number of elements in H. This representation keeps the heap balanced at all times. See the right-hand side of Figure 2.3 for the array representation of the heap on the left-hand side.

Implementing the Heap Operations

The heap element with smallest key is at the root, so it takes $O(1)$ time to identify the minimal element. How do we add or delete heap elements? First consider adding a new heap element v, and assume that our heap H has $n < N$ elements so far. Now it will have $n + 1$ elements. To start with, we can add the new element v to the final position $i = n + 1$, by setting $\texttt{H}[i] = v$. Unfortunately, this does not maintain the heap property, as the key of element v may be smaller than the key of its parent. So we now have something that is almost a heap, except for a small "damaged" part where v was pasted on at the end.

We will use the procedure `Heapify-up` to fix our heap. Let $j = \texttt{parent}(i) = \lfloor i/2 \rfloor$ be the parent of the node i, and assume $\texttt{H}[j] = w$. If $\texttt{key}[v] < \texttt{key}[w]$, then we will simply swap the positions of v and w. This will fix the heap property at position i, but the resulting structure will possibly fail to satisfy the heap property at position j—in other words, the site of the "damage" has moved upward from i to j. We thus call the process recursively from position

Now, the function f_3 isn't so hard to deal with. It starts out smaller than 10^n, but once $n \geq 10$, then clearly $10^n \leq n^n$. This is exactly what we need for the definition of $O(\cdot)$ notation: for all $n \geq 10$, we have $10^n \leq cn^n$, where in this case $c = 1$, and so $10^n = O(n^n)$.

Finally, we come to function f_5, which is admittedly kind of strange-looking. A useful rule of thumb in such situations is to try taking logarithms to see whether this makes things clearer. In this case, $\log_2 f_5(n) = \sqrt{\log_2 n} = (\log_2 n)^{1/2}$. What do the logarithms of the other functions look like? $\log f_4(n) = \log_2 \log_2 n$, while $\log f_2(n) = \frac{1}{3} \log_2 n$. All of these can be viewed as functions of $\log_2 n$, and so using the notation $z = \log_2 n$, we can write

$$\log f_2(n) = \frac{1}{3}z$$
$$\log f_4(n) = \log_2 z$$
$$\log f_5(n) = z^{1/2}$$

Now it's easier to see what's going on. First, for $z \geq 16$, we have $\log_2 z \leq z^{1/2}$. But the condition $z \geq 16$ is the same as $n \geq 2^{16} = 65,536$; thus once $n \geq 2^{16}$ we have $\log f_4(n) \leq \log f_5(n)$, and so $f_4(n) \leq f_5(n)$. Thus we can write $f_4(n) = O(f_5(n))$. Similarly we have $z^{1/2} \leq \frac{1}{3}z$ once $z \geq 9$—in other words, once $n \geq 2^9 = 512$. For n above this bound we have $\log f_5(n) \leq \log f_2(n)$ and hence $f_5(n) \leq f_2(n)$, and so we can write $f_5(n) = O(f_2(n))$. Essentially, we have discovered that $2^{\sqrt{\log_2 n}}$ is a function whose growth rate lies somewhere between that of logarithms and polynomials.

Since we have sandwiched f_5 between f_4 and f_2, this finishes the task of putting the functions in order.

Solved Exercise 2

Let f and g be two functions that take nonnegative values, and suppose that $f = O(g)$. Show that $g = \Omega(f)$.

Solution This exercise is a way to formalize the intuition that $O(\cdot)$ and $\Omega(\cdot)$ are in a sense opposites. It is, in fact, not difficult to prove; it is just a matter of unwinding the definitions.

We're given that, for some constants c and n_0, we have $f(n) \leq cg(n)$ for all $n \geq n_0$. Dividing both sides by c, we can conclude that $g(n) \geq \frac{1}{c}f(n)$ for all $n \geq n_0$. But this is exactly what is required to show that $g = \Omega(f)$: we have established that $g(n)$ is at least a constant multiple of $f(n)$ (where the constant is $\frac{1}{c}$), for all sufficiently large n (at least n_0).

Exercises

1. Suppose you have algorithms with the five running times listed below. (Assume these are the exact running times.) How much slower do each of these algorithms get when you (a) double the input size, or (b) increase the input size by one?
   (a) n^2
   (b) n^3
   (c) $100n^2$
   (d) $n \log n$
   (e) 2^n

2. Suppose you have algorithms with the six running times listed below. (Assume these are the exact number of operations performed as a function of the input size n.) Suppose you have a computer that can perform 10^{10} operations per second, and you need to compute a result in at most an hour of computation. For each of the algorithms, what is the largest input size n for which you would be able to get the result within an hour?
   (a) n^2
   (b) n^3
   (c) $100n^2$
   (d) $n \log n$
   (e) 2^n
   (f) 2^{2^n}

3. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

$$f_1(n) = n^{2.5}$$
$$f_2(n) = \sqrt{2n}$$
$$f_3(n) = n + 10$$
$$f_4(n) = 10^n$$
$$f_5(n) = 100^n$$
$$f_6(n) = n^2 \log n$$

4. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

$$g_1(n) = 2^{\sqrt{\log n}}$$
$$g_2(n) = 2^n$$
$$g_4(n) = n^{4/3}$$
$$g_3(n) = n(\log n)^3$$
$$g_5(n) = n^{\log n}$$
$$g_6(n) = 2^{2^n}$$
$$g_7(n) = 2^{n^2}$$

5. Assume you have functions f and g such that $f(n)$ is $O(g(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

(a) $\log_2 f(n)$ is $O(\log_2 g(n))$.

(b) $2^{f(n)}$ is $O(2^{g(n)})$.

(c) $f(n)^2$ is $O(g(n)^2)$.

6. Consider the following basic problem. You're given an array A consisting of n integers $A[1], A[2], \ldots, A[n]$. You'd like to output a two-dimensional n-by-n array B in which $B[i, j]$ (for $i < j$) contains the sum of array entries $A[i]$ through $A[j]$—that is, the sum $A[i] + A[i+1] + \cdots + A[j]$. (The value of array entry $B[i, j]$ is left unspecified whenever $i \geq j$, so it doesn't matter what is output for these values.)

Here's a simple algorithm to solve this problem.

```
For i = 1, 2, ..., n
  For j = i + 1, i + 2, ..., n
    Add up array entries A[i] through A[j]
    Store the result in B[i, j]
  Endfor
Endfor
```

(a) For some function f that you should choose, give a bound of the form $O(f(n))$ on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm).

(b) For this same function f, show that the running time of the algorithm on an input of size n is also $\Omega(f(n))$. (This shows an asymptotically tight bound of $\Theta(f(n))$ on the running time.)

(c) Although the algorithm you analyzed in parts (a) and (b) is the most natural way to solve the problem—after all, it just iterates through

the relevant entries of the array B, filling in a value for each—it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time $O(g(n))$, where $\lim_{n\to\infty} g(n)/f(n) = 0$.

7. There's a class of folk songs and holiday songs in which each verse consists of the previous verse, with one extra line added on. "The Twelve Days of Christmas" has this property; for example, when you get to the fifth verse, you sing about the five golden rings and then, reprising the lines from the fourth verse, also cover the four calling birds, the three French hens, the two turtle doves, and of course the partridge in the pear tree. The Aramaic song "Had gadya" from the Passover Haggadah works like this as well, as do many other songs.

These songs tend to last a long time, despite having relatively short scripts. In particular, you can convey the words plus instructions for one of these songs by specifying just the new line that is added in each verse, without having to write out all the previous lines each time. (So the phrase "five golden rings" only has to be written once, even though it will appear in verses five and onward.)

There's something asymptotic that can be analyzed here. Suppose, for concreteness, that each line has a length that is bounded by a constant c, and suppose that the song, when sung out loud, runs for n words total. Show how to encode such a song using a script that has length $f(n)$, for a function $f(n)$ that grows as slowly as possible.

8. You're doing some stress-testing on various models of glass jars to determine the height from which they can be dropped and still not break. The setup for this experiment, on a particular type of jar, is as follows. You have a ladder with n rungs, and you want to find the highest rung from which you can drop a copy of the jar and not have it break. We call this the *highest safe rung*.

It might be natural to try binary search: drop a jar from the middle rung, see if it breaks, and then recursively try from rung $n/4$ or $3n/4$ depending on the outcome. But this has the drawback that you could break a lot of jars in finding the answer.

If your primary goal were to conserve jars, on the other hand, you could try the following strategy. Start by dropping a jar from the first rung, then the second rung, and so forth, climbing one higher each time until the jar breaks. In this way, you only need a single jar—at the moment