

Chapter 21

Complexity of Algorithms

The goal of this chapter is to develop the language, ideas and notations that computer scientists use to analyze the speeds algorithms, and to compare and contrast the speeds of different algorithms that perform the same task. This kind of analysis is called the **time-complexity** of an algorithm, or, more often, just the **complexity** of an algorithm.

In Chapter 8 we noted that the number of steps needed for an algorithm to perform a task depends on what the input is. For example, an algorithm that puts a list into numeric order is likely to expend fewer steps on an input list that's already sorted than one that's not. Also, as a general rule, the bigger the input, the more steps the algorithm needs to process it. We introduced the idea of measuring the worst-case performance of an algorithm with a function $f(n)$, meaning that for any input of size n , the algorithm takes $f(n)$ or fewer steps to process it. Perhaps for most inputs of size n the algorithm takes fewer than $f(n)$ steps, but for some "bad" inputs the algorithm may have to take as many as $f(n)$ steps. (We will be a bit vague about what is meant by the "size" of the input, and in general this depends on context. Size could be in bytes, number of list entries, or number of vertices in an input graph, etc.)

In this chapter we will continue to measure performance of algorithms in terms of functions $f(n)$, but we will sharpen our understanding of how to compare such functions: Given two of them, we will describe rigorously when one is better than the other, or when one is just as good as the other.

To start the discussion, suppose we have two algorithms, Algorithm 1 and Algorithm 2, that do exactly the same thing. Let's say Algorithm 1 takes $f(n) = 10 + x^2$ steps (in the worst case) to process an input of size n , whereas Algorithm 2 takes $g(n) = 5 + \frac{1}{100}x^3$ steps.

Which algorithm is better?

To answer this question, let's plot graphs of $f(n)$ and $g(n)$, as in Figure 21.1. The top graph plots them for values of n from 0 to 12. In this window it appears that $f(n)$ is bigger than $g(n)$, and is growing much more quickly than $g(n)$. We might take this as evidence that Algorithm 2 is better, because it involves a smaller number $g(n)$ of steps.

However, the bottom part of the figure takes a wider view, and plots the same two functions for $0 \leq n \leq 120$. We see that $g(n)$ overtakes $f(n)$ somewhere around $n = 100$, and thereafter $f(n) < g(n)$. Thus, contrary to the pervious paragraph's hasty conclusion, it is Algorithm 1 that is better, because it only requires $f(n)$ steps, and this is less than $g(n)$ for all of the infinitely many values of n except the first 100 or so.

The next section develops a method for comparing two such functions f and g , a system that filters out any superficial idiosyncrasies and captures their essential long-range behavior.

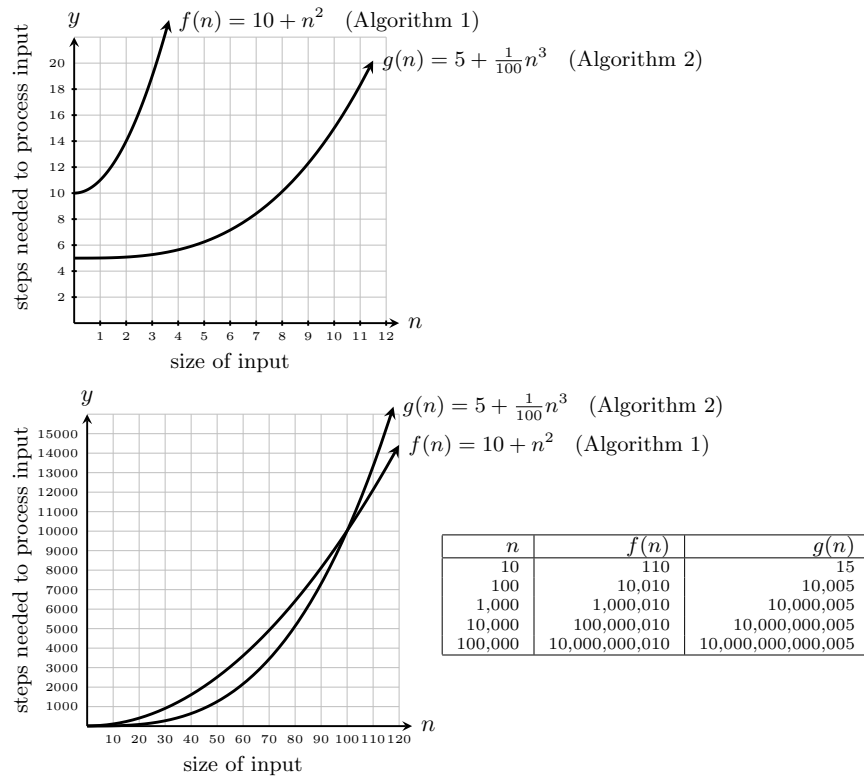


Fig. 21.1 Functions $f(n)$ and $g(n)$ for $0 \leq n \leq 12$ (top) and $0 \leq n \leq 120$ (bottom). At first, $g(n) < f(n)$, but beyond about $n = 100$ the order is reversed, and $g(n) > f(n)$. The table shows why: each time n increases by a factor of 10, $f(n)$ increases about 100-fold, while $g(n)$ increases about 1000-fold. In other words, though it starts off smaller, $g(n)$ grows about 10 times faster than $f(n)$, and eventually overtakes it.

21.1 Big-O Notation

In Figure 21.1, we regarded $f(n)$ as better than $g(n)$ because there was a number $N = 100$ for which $f(n) < g(n)$ whenever $n > 100$. This leads to the first of two guiding principles that will yield a meaningful formulation of algorithm complexity.

Guiding Principle A For worst-case performance, $f(n)$ is as good as or better than $g(n)$ if there exists an integer N for which $f(n) \leq g(n)$ whenever $n > N$.

To motivate our second guiding principle, suppose Algorithm 1 has worst-case performance $f(n)$, and Algorithm 2 has worst-case performance $g(n)$, and there is a number A for which $f(n) \leq A \cdot g(n)$. This means Algorithm 1 may need to perform up to A times as many steps as Algorithm 2. So if Algorithm 1 is too slow compared to Algorithm 2, this can be remedied by running Algorithm 1 on a computer that is A times faster than the one Algorithm 2 runs on.

But if *there is no number* A for which $f(n) \leq A \cdot g(n)$ for all n , then $f(n) > A \cdot g(n)$ for some n , no matter how big A is. This means that for some inputs, Algorithm 1 is slower than Algorithm 2, *no matter how fast the computer it is run is*.

Guiding Principle B For worst-case performance, $f(n)$ is as good as or better than $g(n)$ if there exists a number A for which $f(n) \leq A \cdot g(n)$.

These principles suggest that if $f(n)$ is “as good as or better” than $g(n)$, then it is not necessarily true that $f(n) \leq g(n)$ for all n . Instead, $f(n)$ is “as good as or better” than $g(n)$ if there are positive numbers N and A for which $f(n) \leq A \cdot g(n)$ for all $n > N$. This leads to the chapter’s main definition, a means of comparing functions in the context of our two guiding principles.

Definition 21.1. If f and g are functions of n , then **f is of order at most g** , written “ $f(n)$ is $O(g(n))$,” if there exist positive numbers N and A for which

$$|f(n)| \leq A \cdot |g(n)|$$

for all $n > N$. (In this case we sometimes say “ $f(n)$ is big-O of $g(n)$.”)

Two comments: First, we usually think of n as an integer (input size), but we often interpret it as a real number in the definition. This is done to make the graphs of f and g the continuous smooth curves that we are familiar with from calculus. (And it is a harmless assumption, as integers are real numbers.) Second, we tend to think of $f(n)$ and $g(n)$ as being *positive* (measuring run-time). But to make the definition useful and robust, they appear in absolute value.

Think of Definition 21.1 as giving a way of saying that one function $f(n)$ is less-than-or-equal to another function $g(n)$; a way that glosses over superfluous details and captures the big picture. If $f(n)$ is $O(g(n))$, then, for all intents and purposes, $f \leq g$ in the sense that $f(n) \leq A \cdot g(n)$ when n is large. In other words, compared to $g(n)$, the function $f(n)$ does not grow beyond a finite, constant multiple A of $g(n)$.

In this sense, $f(n)$ being $O(g(n))$ means that the long-term growth of $f(n)$ compares favorably with that of $g(n)$. The definition gives a concise way of saying that $f(n)$ never gets “too far” beyond $g(n)$.

Notice that proving that $f(n)$ is $O(g(n))$ amounts to proving the statement

$$\exists A > 0, \exists N > 0, \forall n > N, |f(n)| \leq A \cdot |g(n)|.$$

To prove it, we must find values for A and N for which $|f(n)| \leq A \cdot |g(n)|$ is true for all $n > N$. Usually A and N will suggest themselves from f and g .

Example 21.1. Show that the polynomial $f(n) = 3n - 2 + 4n^2$ is $O(n^2)$.

Solution As long as $n > 1$ we have $n \leq n^2$ and $n^2 \leq n^3$, so

$$\begin{aligned} |f(n)| &= |3n - 2 + 4n^2| \leq |3n| + |2| + |4n^2| \quad (\text{triangle inequality (20.7), (20.8)}) \\ &= 3n + 2 + 4n^2 \\ &< 3n^2 + 2n^2 + 4n^2 \\ &= 9n^2 = 9 \cdot |n^2|. \end{aligned}$$

So if $A = 9$ and $N = 1$, then $|f(n)| \leq A|n^2|$ when $n > N$. Thus $f(n)$ is $O(n^2)$. \square

Next we will compare power, exponential and logarithm functions to one another. Our first result explains the relations among the power functions.

Proposition 21.1. *If $1 \leq d \leq \ell$, then the power function n^d is $O(n^\ell)$. However, if $d < \ell$ then n^ℓ is **not** $O(n^d)$.*

Proof. Say $\ell \geq d$. Let $A = N = 1$. We immediately get $n^d \leq An^\ell$ for $n > N$. As all terms are positive, $|n^d| \leq A|n^\ell|$ for $n > N$. Thus n^d is $O(n^\ell)$ by Definition 21.1.

Now suppose $d < \ell$. We need to show n^ℓ is **not** $O(n^d)$. Suppose for the sake of contradiction that n^ℓ is $O(n^d)$. Definition 21.1 guarantees positive numbers A and N for which $|n^\ell| \leq A|n^d|$ when $n > N$. Dropping absolute values (all terms are positive) and dividing both sides by n^d , we get $\frac{n^\ell}{n^d} \leq A$, so $n^{\ell-d} \leq A$ for all $n > N$. As $\ell - d$ is positive, the power function $n^{\ell-d}$ grows arbitrarily large as n increases. Thus $n^{\ell-d} > A$ for large enough n , contradicting the previous sentence. \square

If you know calculus, you have some useful tools for comparing functions. Consider the next proposition. (If you don't know calculus, you can ignore it.)

Proposition 21.2. *Given functions f and g , if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L$, and L is finite, then $f(n)$ is $O(g(n))$. If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \pm\infty$, then $f(n)$ is **not** $O(g(n))$.*

For a proof, do Exercises 11 and 12. In applying Proposition 21.2 it is likely that L'Hôpital's rule comes into play. To see how, let's do Example 21.1 this way.

To show $3n - 2 + 4n^2$ is $O(n^2)$, just evaluate $\lim_{n \rightarrow \infty} \frac{3n - 2 + 4n^2}{n^2} = \lim_{n \rightarrow \infty} \frac{3+8n}{2n} = \lim_{n \rightarrow \infty} \frac{8}{2} = 4$. As 4 is finite, Proposition 21.2 says the function $3n - 2 + 4n^2$ is $O(n^2)$. Again, if you don't know calculus, you can ignore this. (But take calculus!)

Let's use Proposition 21.1 to compare specific power functions. Figure 21.2 shows the graphs of $y = n^d$ for $d = 1, 2, \dots, 5$. In order show the big-picture, the scale on the y -axis is compressed logarithmically, so that in each unit the y value doubles. This view changes the appearance of the power functions, as they "flatten out" as n increases. (Compare to this to the same functions in Figure 20.2.)

You can see that by $n = 30$, the difference between (say) n^4 and n^5 is vast, and getting vaster. This is in agreement with Proposition 21.1, which says n^5 is not $O(n^4)$; it grows beyond any finite multiple of n^4 .

It is interesting to look at the exponential functions 2^n , 3^n and 4^n in Figure 21.2, which appear as straight lines in this compressed grid. (Compare them to Figure 20.2, which plots the same functions.) What is striking is how astronomically huge they grow, especially compared to the power functions, which "flatten out."

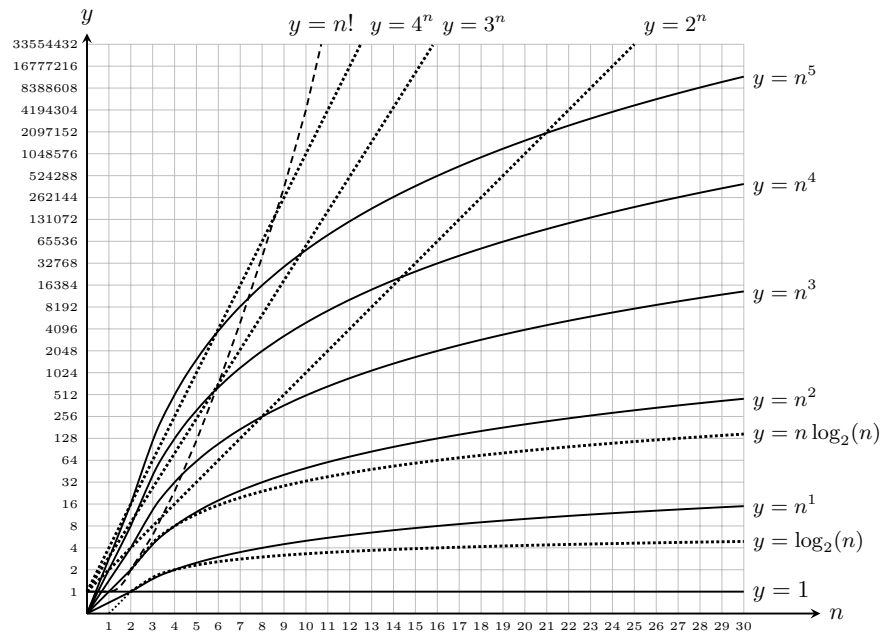


Fig. 21.2 Some functions plotted on a grid where each tick on the y -axis is twice its value on the previous tick. (So the y -axis is a \log_2 scale.) Note that any power function n^a grows vastly slower than any exponential function b^n . But $y = n!$ eventually overtakes any exponential function.

With this picture as a guide, our next task is to compare power and exponential functions in terms of Definition 21.1.

But before we do so, let's pause to lay out a road map of what we are about to do. Let's use the notation $f(x) \preceq g(x)$ to mean $f(x)$ is $O(g(x))$. By $f(x) \prec g(x)$, we mean $f(x)$ is $O(g(x))$ but $g(x)$ is **not** $O(f(x))$. Then Proposition 21.1 implies

$$1 \prec n \prec n^2 \prec n^3 \prec n^4 \prec \dots$$

In the remainder of this section we are going to show

$$1 \prec \log_2(n) \prec n \prec n^2 \prec n^3 \prec n^4 \prec \dots \prec 2^n \prec 3^n \prec 4^n \prec 5^n \prec \dots \quad (21.1)$$

This is the content of the section's remaining propositions.

But first, a quick word about the constant function 1 that appears on the left of the chain (21.1), above. This constant function $y = f(n) = 1$ is graphed in Figure 21.2. In general, a constant function has the form $f(n) = c$, where c is some constant number. Whatever n is plugged into the function, the output is c . It is immediate that any constant function c is $O(1)$, because $c \leq A \cdot 1$ for $A = c$. Also c is $O(n)$, because clearly $c \leq A \cdot n$ for all $n > N = \frac{c}{A}$.

Next we are going to compare power functions with exponential functions. First we ponder exponential functions, those at the right end of chain (21.1). Our next proposition implies $2^n \prec 3^n \prec 4^n \prec 5^n \prec 6^n$, etc.

Proposition 21.3. *If $1 < a < b$, then the exponential function a^n is $O(b^n)$. But the exponential function b^n is **not** $O(a^n)$.*

Proof. Suppose $1 < a < b$. It is immediate that a^n is $O(b^n)$, because then $a^n \leq 1 \cdot b^n$ for all $n > 0$, and Definition 21.1 applies.

Next we prove b^n not $O(a^n)$. For the sake of contradiction, say b^n is $O(a^n)$. Then Definition 21.1 says there are positive numbers A and N for which $b^n \leq Aa^n$ for all $n > N$. (We have omitted the absolute values because everything is sight is already positive.) From this, $\frac{b^n}{a^n} \leq A$, which means $(\frac{b}{a})^n < A$ for all $n > N$. This is a contradiction because $a < b$ forces $\frac{b}{a} > 1$, so the exponential function $(\frac{b}{a})^n$ is bigger than A for all sufficiently large n . \square

Our next result (Proposition 21.4) will compare power functions n^d to exponential functions b^n . It asserts that any power function n^d is $O(b^n)$ if $b > 1$. For example, consider the power function n^{1000} , compared to the exponential function 2^n . For small values of n we have $n^{1000} > 2^n$. For example, $10^{1000} = 10,000 > 1024 = 2^{10}$. But you may have a sense that, because 2^n doubles each time n increases by 1, then for large enough n , we have $n^{1000} < 2^n$. If you are comfortable with your intuition and knowledge of how an exponential function eventually surpasses a power function, then you may want to skip the proof of Proposition 21.4. (But do read the statement of the proposition.) The proof given here is somewhat involved because it avoids calculus. You are invited to write your own, simpler, proof using Proposition 21.2 (Exercise 13).

Proposition 21.4. *If $d \geq 1$ and $b > 1$, then the power function n^d is $O(b^n)$, but the exponential function b^n is not $O(n^d)$.*

Proof. First we will show n^d is $O(b^n)$. To begin, note that if $n > 1$, then $\frac{n}{n-1} > 1$, because the numerator is greater than the denominator. But as n grows bigger, $\frac{n}{n-1}$ gets closer and closer to 1. The reason is that $\frac{n}{n-1} = 1 + \frac{1}{n-1}$ (check this), and the fraction $\frac{1}{n-1}$ approaches 0 as n gets bigger. (If you have had some calculus then you know to express this phenomenon as $\lim_{n \rightarrow \infty} \frac{n}{n-1} = 1$. However, we will use no calculus here.) Because $b > 1$, the number $\sqrt[d]{b}$ is greater than 1, so there is some $N > 0$ for which $n > N$ implies $\frac{n}{n-1} < \sqrt[d]{b}$. This means

$$\left(\frac{n}{n-1}\right)^d < b \quad \text{for } n > N. \quad (21.2)$$

Now let $A = N^d$. We claim that $n^d \leq Ab^n$ when $n > N$. Indeed, if $n > N$, then

$$\begin{aligned} n^d &< (N+n)^d && \text{(because } n < N+n\text{)} \\ &= \left(\frac{N}{N}\right)^d \left(\frac{N+1}{N+1}\right)^d \left(\frac{N+2}{N+2}\right)^d \cdots \left(\frac{N+n-1}{N+n-1}\right)^d (N+n)^d && \text{(multiply by 1)} \\ &= N^d \left(\frac{N+1}{N}\right)^d \left(\frac{N+2}{N+1}\right)^d \left(\frac{N+3}{N+2}\right)^d \cdots \left(\frac{N+n}{N+n-1}\right)^d && \text{(move denominators} \\ &&& \text{one place to right)} \\ &< N^d \underbrace{b \cdot b \cdot b \cdot b \cdots b}_{n \text{ times}} && \text{(by Equation (21.2))} \\ &= Ab^n. && \text{(because } A = N^d\text{)} \end{aligned}$$

We've established $n^d \leq Ab^n$ for $n > N$, which proves n^d is $O(b^n)$.

Next we show that b^n is not $O(n^d)$. Suppose for the sake of contradiction that b^n is $O(n^d)$. Then there are positive numbers A and N for which

$$b^n \leq An^d \quad \text{when } n > N. \quad (21.3)$$

Because $1 < b$, there is a number a for which $1 < a < b$. By the first part of the proof, we know n^d is $O(a^n)$, so there exist positive A' and N' for which

$$n^d \leq A'a^n \quad \text{when } n > N'. \quad (21.4)$$

Combining inequalities (21.3) and (21.4) yields $b^n \leq An^d \leq AA'a^n$. Dividing this by a^n gives $\frac{b^n}{a^n} \leq AA'$, or $\left(\frac{b}{a}\right)^n \leq AA'$, if n is bigger than both N and N' . This is a contradiction, as the fact $1 < a < b$ ensures $\frac{b}{a} > 1$, so the exponential function $\left(\frac{b}{a}\right)^n$ actually exceeds AA' for all sufficiently large n . \square

What about logarithm functions? For fixed bases a and b , the change of base formula (Fact 20.6 on page 460) says

$$\frac{\log_a(n)}{\log_b(n)} = \frac{\frac{\log_{10}(n)}{\log_{10}(a)}}{\frac{\log_{10}(n)}{\log_{10}(b)}} = \frac{\log_{10}(b)}{\log_{10}(a)}.$$

Thus $\log_a(n) = \frac{\log_{10}(b)}{\log_{10}(a)} \log_b(n) = A \log_b(n)$ for a fixed constant $A = \frac{\log_{10}(b)}{\log_{10}(a)}$. (Let's assume $a, b > 1$, so that A is positive.) This implies $\log_a(n)$ is $O(\log_b(n))$ regardless of the bases a and b . In other words, $\log_a(n) \preceq \log_b(n)$ and $\log_b(n) \preceq \log_a(n)$, for a and b . Let's adopt the notation $f(n) \simeq g(n)$ to mean that both $f(n) \preceq g(n)$ and $g(n) \preceq f(n)$ hold, in which case we say that $f(n)$ and $g(n)$ have the same **order**.

Then we have, for instance,

$$\log_2(n) \simeq \log_3(n) \simeq \log_4(n) \simeq \log_5(n) \simeq \dots$$

Since a logarithm's base has no bearing on its order, we will finish our investigation using \log_2 .

Proposition 21.5. *The function $\log_2(n)$ is $O(n)$, but n is not $O(\log_2(n))$. Also, the constant function 1 is $O(\log_2(n))$, but $\log_2(n)$ is not $O(1)$.*

Proof. Observe that $n < 2^n$ holds for all positive integers n . (This should be obvious, or you can prove it with induction.) Therefore, for all $n > 2$ we have

$$\begin{aligned} 2 &< n < 2^n \\ \log_2(2) &< \log_2(n) < \log_2(2^n) \\ 1 &< \log_2(n) < n. \end{aligned}$$

(In taking logs in the second step here, we used the fact that $\log_2(n)$ is an *increasing* function, that is, $x < y$ implies $\log_2(x) < \log_2(y)$. Thus taking \log_2 did not reverse any $<$.) Note that $1 < \log_2(n)$ for $n > 2$ means 1 is $O(\log_2(n))$, and $\log_2(n) < n$ means $\log_2(n)$ is $O(n)$.

But n is **not** $O(\log_2(n))$ because if it were, there would be a positive A for which $n \leq A \log_2(n)$ for all $n > N$, for some N . From this we would get $2^n \leq 2^{A \log_2(n)}$ for all $n > N$. This becomes $2^n \leq (2^{\log_2(n)})^A = n^A$ for all $n > N$, meaning 2^n is $O(n^A)$, which contradicts Proposition 21.4.

To see that $\log_2(n)$ is not $O(1)$, suppose it were. Then $\log_2(n) \leq A \cdot 1$ for all sufficiently large n . But this is a contradiction, for as long as $n > 4^A$, we have $\log_2(n) > \log_2(4^A) = A \log_2(4) = 2A > A$. \square

The previous four propositions confirm the chain (21.1) on page 470, repeated here for emphasis:

$$1 < \log_2(n) < n < n^2 < n^3 < n^4 < \dots < 2^n < 3^n < 4^n < 5^n < \dots < n! < n^n.$$

(Actually, two new entries $n!$ and n^n have been slipped in on the right. Regarding them, see the exercises 6 and 7 below.)

Regarding the functions $f(n)$ that appear on this list, if the worst-case performance of an algorithm is $O(f(n))$, then we would want $f(n)$ to be as far to the left as possible, for efficiency improves the further left we can go. An algorithm whose worst-case performance was $f(n) = n!$ or $f(n) = n^n$ would be a very bad algorithm, usable only for small values of n .

Exercises for Section 21.1

1. Show that $f(n) = 3 + n + 2^n$ is $O(2^n)$.
2. Show that $f(n) = 2n^4 + n^2 - n - 3$ is $O(n^4)$.
3. Show that $f(n) = 25 + 8n + \log_2(n)$ is $O(n)$.
4. Show that $f(n) = \log_2(n) \cdot n^3$ is $O(n^4)$.
5. Show that $n \log_2(n)$ is $O(n^2)$, but n^2 is not $O(n \log_2(n))$. (See Figure 21.2.)
6. Show that the function $f(n) = n!$ is $O(n^n)$, but n^n is not $O(n!)$.
7. Show that the function $f(n) = 2^n$ is $O(n!)$, but $n!$ is not $O(2^n)$. (See Figure 21.2.)
8. Show that the function $f(n) = \binom{n}{\lfloor n/2 \rfloor}$ is $O(2^n)$.
9. Let F_n be the n th Fibonacci number. Show that F_n is $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$, and that $\left(\frac{1+\sqrt{5}}{2}\right)^n$ is $O(F_n)$.
10. Two different algorithms, Algorithm 1 and Algorithm 2, accomplish the same task. Algorithm 1 has worst-case performance $f(n)$ and Algorithm 2 has worst-case performance $g(n)$ (where n is the input size). Say these algorithms run on two different computers: Computer 1 and Computer 2, respectively.
 - (a) Suppose $f(n)$ is $O(g(n))$. Show that there exists a number B such that if Computer 1 is B times faster than Computer 2, then Algorithm 1 will always finish before Algorithm 2 when each is run on the same input.
 - (b) Suppose $f(n)$ is **not** $O(g(n))$. Show that no matter how fast Computer 1 is, there are some inputs for which Algorithm 1 is slower than Algorithm 2.
11. Prove the first part of Prop. 21.2: If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L$, then $f(n)$ is $O(g(n))$.
12. Prove the last part of Prop. 21.2: If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \pm\infty$, then $f(n)$ is *not* $O(g(n))$.
13. Use Proposition 21.2 to prove Proposition 21.4.
14. Use Proposition 21.2 to prove Proposition 21.5.
15. Show that the relation \prec (defined on page 470) is a transitive relation on the set of all real-valued functions on $(0, \infty)$. That is, show that $f(n) \prec g(n)$ and $g(n) \prec h(n)$ implies $f(n) \prec h(n)$.
16. On page 472 we defined $f(n) \simeq g(n)$ if both $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$. Prove that \simeq is an equivalence relation on the real-valued functions on $(0, \infty)$.

21.2 Big- Ω and Big- Θ

Big-O notation is very useful for measuring and comparing the efficiencies of algorithms and programs. For example, suppose you have written a program, and have determined that its worst-case behavior is modeled by the function $f(n) = 18 + 12n^2 + 3n^3$ (meaning that the program takes $f(n)$ or fewer steps when processing an input of size n .) Because $f(n)$ is $O(n^3)$, you can gloss over the details of $f(n)$ and just say that you have an $O(n^3)$ program.

Now imagine that someone else writes another program that solves the same problem as yours. Because of minor differences in coding, maybe their worst-case behavior is given by $h(n) = 16 + 5n^2 + 5n^3$. Both programs are $O(n^3)$, and hence are considered equally efficient.

But if someone down the hall has an $O(n^2)$ program, then theirs is better.

Saying that $f(n)$ is $O(g(n))$ means that f compares favorably to g in the sense that $|f(n)| \leq A \cdot |g(n)|$ for large n . But the person down the hall wants to convince you that your program compares unfavorably to theirs. A slightly different notation called big- Ω (pronounced “big-omega”) is used for this type of discussion. Below we repeat the definition of big- O to highlight the parallels between it and big- Ω .

Definition 21.2. Suppose f and g are functions of n .

- $f(n)$ is $O(g(n))$ if \exists numbers A and N such that $|f(n)| \leq A \cdot |g(n)|$ when $n > N$.
- $f(n)$ is $\Omega(g(n))$ if \exists numbers A and N such that $|f(n)| \geq A \cdot |g(n)|$ when $n > N$.
- $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Notice that $|f(n)| \geq A \cdot |g(n)|$ implies $|g(n)| \leq \frac{1}{A} \cdot |f(n)|$, so saying $f(n)$ is $\Omega(g(n))$ means exactly the same thing as saying $g(n)$ is $O(f(n))$. The definition is phrased the way it is because in practice $f(n)$ is usually a complicated function while $g(n)$ is a simple reference function, like $g(n) = n^3$ or $g(n) = n \log_2(n)$, etc.

Say $f(n)$ gives the worst-case performance for an algorithm. If $f(n)$ is $O(g(n))$, we say the algorithm is $O(g(n))$. If $f(n)$ is $\Omega(g(n))$, we say the algorithm is $\Omega(g(n))$.

Just as the statement “ $f(n)$ is $O(g(n))$ ” means “ $f(n)$ is as good as or better than $g(n)$,” the statement “ $f(n)$ is $\Omega(g(n))$ ” means “ $f(n)$ is as bad as or worse than $g(n)$ ” in the sense that $f(n)$ grows beyond some constant multiple of $g(n)$ (or is at least equal a constant multiple of $g(n)$ for sufficiently large n). If we happen to know that a particular algorithm is, say, $\Omega(n^5)$, then it is reasonable to try to devise a different algorithm (that does the same thing) that is $O(n^4)$, or better.

Definition 21.2 also defines the notion of $\Theta(g(n))$. $f(n)$ is $\Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$. For example, on page 472 we saw that for any $a, b > 1$, $\log_a(n)$ is $O(\log_b(n))$ and $\log_b(n)$ is $O(\log_a(n))$, and hence $\log_a(n)$ is $\Theta(\log_b(n))$. If $f(n)$ is $\Theta(g(n))$, we say that $f(n)$ is of **order** $g(n)$. So, for example, logarithms to different bases are of the same order.

For the remainder of this text we will phrase our discussions in terms of big- O , but you may encounter big- Ω and big- Θ in future reading.

21.3 Polynomial Algorithms

As noted in the previous section, Big-O notation is useful for measuring the efficiency of algorithms. If an algorithm's worst-case performance is $f(n)$ steps to process an input of size n , then, $f(n)$ can be fairly complex. But usually a simple $g(n)$ (such as a power or exponential function) can be found such that $f(n)$ is $O(g(n))$. Then the simple function $g(n)$ is meaningful generic measure of the algorithm's complexity. In such a case we say that the **algorithm is $O(g(n))$** .

The following propositions (and corollaries) are useful for transforming a complicated $f(n)$ to a simpler $g(n)$ for which $f(n)$ is $O(g(n))$.

Proposition 21.6. *If $f(n) = f_1(n) \pm f_2(n) \pm \cdots \pm f_k(n)$ and each $f_i(n)$ is $O(g(n))$, then $f(n)$ is $O(g(n))$.*

Proof. (Direct) Say each $f_i(n)$ is $O(g(n))$. This means that there exist positive numbers A_1, A_2, \dots, A_k and N_1, N_2, \dots, N_k , such that, for each index i , the inequality $|f_i(n)| \leq A_i |g(n)|$ holds for all $n \geq N_i$. Put $A = A_1 + A_2 + \cdots + A_k$. Let $N = \max \{N_1, N_2, \dots, N_k\}$, that is, N is the largest of the N_i . If $n > N$, then

$$\begin{aligned} |f(n)| &= |f_1(n) \pm f_2(n) \pm \cdots \pm f_k(n)| \\ &\leq |f_1(n)| + |f_2(n)| + \cdots + |f_k(n)| && \text{(triangle inequality)} \\ &\leq A_1 |g(n)| + A_2 |g(n)| + \cdots + A_k |g(n)| \\ &\leq (A_1 + A_2 + \cdots + A_k) \cdot |g(n)| \\ &\leq A \cdot |g(n)|. \end{aligned}$$

By Definition 21.1, $f(n)$ is $O(g(n))$. □

Proposition 21.7. *Suppose $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$. Then the product $f_1(n)f_2(n)$ is $O(g_1(n)g_2(n))$.*

Proof. (Direct) Suppose $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$. Definition 21.1 says there exist positive numbers A_1 and N_1 $|f_1(n)| \leq A_1 \cdot |g_1(n)|$ for all $n \geq N_1$, and positive numbers A_2 and N_2 , such that $|f_2(n)| \leq A_2 \cdot |g_2(n)|$ for all $n \geq N_2$. Let $A = A_1 \cdot A_2$ and let $N = \max \{N_1, N_2\}$. Then if $n > N$, the following holds:


$$\begin{aligned} |f_1(n)f_2(n)| &= |f_1(n)| \cdot |f_2(n)| \\ &\leq A_1 \cdot |g_1(n)| \cdot A_2 \cdot |g_2(n)| \\ &\leq A_1 \cdot A_2 \cdot |g_1(n)g_2(n)| \\ &\leq A \cdot |g_1(n)g_2(n)|. \end{aligned}$$

By Definition 21.1, $f_1(n)f_2(n)$ is $O(g_1(n)g_2(n))$. □

Example 21.2. Show that the function $f(n) = 5n^3 - n^2 \log_2(n) + 8$ is $O(n^3)$.

Solution: Our strategy is to show that each functions $5n^3$, $n^2 \log_2(n)$ and 8 is $O(n^3)$, for then Proposition 21.6 implies $5n^3 - n^2 \log_2(n) + 8$ is $O(n^3)$.

First, the constant function 5 is $O(1)$, and n^3 is $O(n^3)$, so Proposition 21.7 says $5n^3$ is $O(n^3)$. Second, $\log_2(n)$ is $O(n)$ by Proposition 21.5, so by Proposition 21.7, $n^2 \log_2(n)$ is $O(n^2n) = O(n^3)$. Finally, the constant function 8 is clearly $O(n^3)$.

Our strategy is successful, so $f(n) = 5n^3 - n^2 \log_2(n) + 8$ is $O(n^3)$. 

The methods used in this example also work to establish two simple corollaries.

Corollary 21.1. If $f(n)$ is $O(g(n))$, and c is a constant, then $cf(n)$ is $O(g(n))$.

Proof. Because c is $O(1)$ and $f(n)$ is $O(g(n))$, Proposition 21.7 implies that $cf(n)$ is $O(1 \cdot g(n)) = O(g(n))$. \square

Corollary 21.2. Any polynomial $f(n) = a_0 + a_1n + a_2n^2 + \cdots + a_dn^d$ of degree d is $O(n^d)$.

Proof. By Proposition 21.1, the power function n^i is $O(n^d)$ when $i \leq d$. Thus each term a_in^i of $f(n)$ is $O(n^d)$. Therefore $f(n)$ is $O(n^d)$, by Proposition 21.6. \square

Now we arrive at two significant ideas. An algorithm is called a **polynomial-time algorithm** if its worst-case performance for input size n is $f(n)$, where $f(n)$ is $O(g(n))$, for some polynomial $g(n)$. (Equivalently, an algorithm is a polynomial-time algorithm if $f(n)$ is $O(n^d)$, for some power function n^d .) Often a polynomial-time algorithm is simply called a **polynomial algorithm**.

An algorithm is called an **exponential-time algorithm** (or just an **exponential algorithm**) if it is not a polynomial algorithm, and its worst-case performance for input size n is $f(n)$, where $f(n)$ is $O(b^n)$ (for $b > 1$).

Figure 21.2 suggests that polynomial algorithms are much quicker than exponential algorithms. In fact, computer scientists regard exponential-time algorithms as little better than useless. For example, if an algorithm's worst-case performance is $f(n) = 2^n$, then even with a modest input size of $n = 60$, the number 2^{60} of steps needed is so great that even on the fastest computer it could take over 3 centuries to finish. And even if we got a computer that was twice as fast, and it only needed 1.5 centuries to finish, consider that all we'd have to do is give it an input of size 61, and we are back to 3 centuries!

By contrast, a polynomial algorithm finishes in a reasonable amount of time, even if the input is large. For this reason, it is important to analyze the time-complexity of the algorithms we use or write. Always aim for polynomial-time.

The next two sections examine two case studies.

21.4 Case Study: Sequential Search versus Binary Search

Let's use our new knowledge to compare the time-complexity of sequential search (Algorithm 9 on page 224) with that of binary search (Algorithm 10 on page 226). These two different algorithms accomplish the same thing: Determine whether a certain number z appears as an entry of a length- n list of numbers that are in numeric order.

Recall that sequential search merely traverses the list from left to right, stopping only when it encounters an entry equal to z , or reaches the end of the list without finding z . In the worst case, it might have visit all n entries; indeed, on page 224 we computed its worst-case performance as $f(n) = 3 + 4n$ steps. As this is a polynomial of degree 1, Corollary 21.2 says $f(n)$ is $O(n)$. In our new parlance, the sequential search algorithm is an $O(n)$ polynomial algorithm.

The binary search algorithm (page 226) is more complex, but faster than sequential search. Recall that it jumps to the middle of the list, compares z to the middle entry, and then ignores either the left- or right-half of the list, depending on whether the middle entry is less than or greater than z . Then it repeats this procedure on the new half-sized list, etc., until it encounters z or finds that it is not in the list. At the end of Section 8.7 we found that its worst-case performance is $5 + 5\lceil \log_2(n) \rceil$ steps. Although there is nothing problematic about rounding the logarithm up here, it is not quite a perfect fit for Proposition 21.5. To remedy this, note that $\log_2(n) \leq \lceil \log_2(n) \rceil \leq \log_2(n) + 1$, so we are safe in saying that the algorithm takes no more than $f(n) = 4 + 5(\log_2(n) + 1) = 9 + 5\log_2(n)$ steps.

Then $f(n) = 5 \cdot 1 + 2 \cdot \log_2(n)$. Here the functions 1 and $\log_2(n)$ are both $O(\log_2(n))$, by Proposition 21.5, so Proposition 21.6 with Corollary 21.1 imply that $f(n)$ is $O(\log_2(n))$. Consequently, the binary search algorithm is $O(\log_2(n))$. This is better than the $O(n)$ sequential search algorithm, as $\log_2(n) \prec n$.

Here is a significant point. Even though binary search is $O(\log_2(n))$, and $\log_2(n)$ is not a polynomial, binary search is nonetheless a polynomial algorithm. The reason for this is that $O(\log_2(n))$ is $O(n)$, by Proposition 21.5 (and n is a polynomial). So technically, when we say that an algorithm is a polynomial algorithm, we really mean that it is no worse than polynomial. Binary search is better than polynomial, but it gets grouped with polynomial algorithms because it's not worse than polynomial.

But although we may classify binary search as a polynomial algorithm, we still say that its time-complexity is $O(\log_2(n))$, because this is an order of magnitude better than $O(n)$. Think of it this way: Recall that Proposition 21.5 also states that n is **not** $O(\log_2(n))$. This means that there is no number A for which $n < A \log_2(n)$ for all n . No matter how big A is, $n > A \log_2(n)$ if n is large enough. Consequently, even if sequential search (which is $O(n)$) runs on an arbitrarily fast computer, and binary search (which is $O(\log_2(n))$) runs on a very slow computer, then sequential search will still be slower than binary search for all but finitely many inputs.

21.5 Case Study: Bubble Sort versus Merge Sort

In Section 8.3, we devised the algorithm Bubble Sort (Algorithm 6, page 212) that sorts a list of numbers. (So, for input $X = (5, 3, 5, 7, 4)$, the output is $X = (3, 4, 5, 5, 7)$, etc.) For convenience, the code is repeated here. (You may want to quickly review the discussion and explanation preceding page 212.)

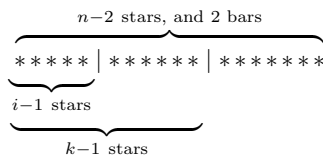
Algorithm 6: Bubble Sort

Input: A list $X = (x_1, x_2, \dots, x_n)$ of numbers
Output: The list sorted into numeric order

```

begin
  for  $k := n-1$  downto 1 do
    for  $i := 1$  to  $k$  do
      if  $x_i > x_{i+1}$  then
         $temp := x_i$  ..... temporarily holds value of  $x_i$ 
         $x_i := x_{i+1}$ 
         $x_{i+1} := temp$  ..... now  $x_i$  and  $x_{i+1}$  are swapped
      end
    end
  end
  output  $X$  ..... now  $X$  is sorted
end
    
```

Let's analyze Bubble Sort's performance. The if-statement inside the nested for-loops executes once for each pair (i, k) with $1 \leq i \leq k \leq n - 1$. In other words, as many times as there are pairs (i, k) satisfying $0 \leq i - 1 \leq k - 1 \leq n - 2$. How many such pairs are there? We have seen this kind of problem before. (See Example ??.) It can be modeled with a length- n list of 2 bars and $n-2$ stars, where there are $i-1$ stars to the left of the first bar, and $k-1$ stars to the left of the second bar.



So if $n = 8$, then $***|*|**$ means $(i, k) = (4, 5)$, and $***||***$ means $(i, k) = (4, 4)$. Also $|*****|$ means $(i, k) = (1, 7)$, and $||*****$ is $(i, k) = (1, 1)$. The number of such lists equals the number of ways to choose 2 out of n spots for the bars, which is $\binom{n}{2} = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$. So the if-statement executes $\frac{1}{2}n^2 - \frac{1}{2}n$ times, and each time it runs at most 4 steps. So in the worst-case, the loop executes $4(\frac{1}{2}n^2 - \frac{1}{2}n) = 2n^2 + 2n$ steps. Including its final **output** command, Bubble Sort does no more than $f(n) = 2n^2 - 2n + 1$ steps. Thus $f(n)$ is order $O(n^2)$.

Conclusion: Bubble Sort is an $O(n^2)$ polynomial algorithm.

Now, $O(n^2)$ is not bad. But there is another sort algorithm that is better. It is called **merge sort**. It is also polynomial, but its worst-case performance is $O(n \log_2(n))$. (Note $O(n \log_2(n))$ is better than $O(n^2)$ by Exercise 21.5)

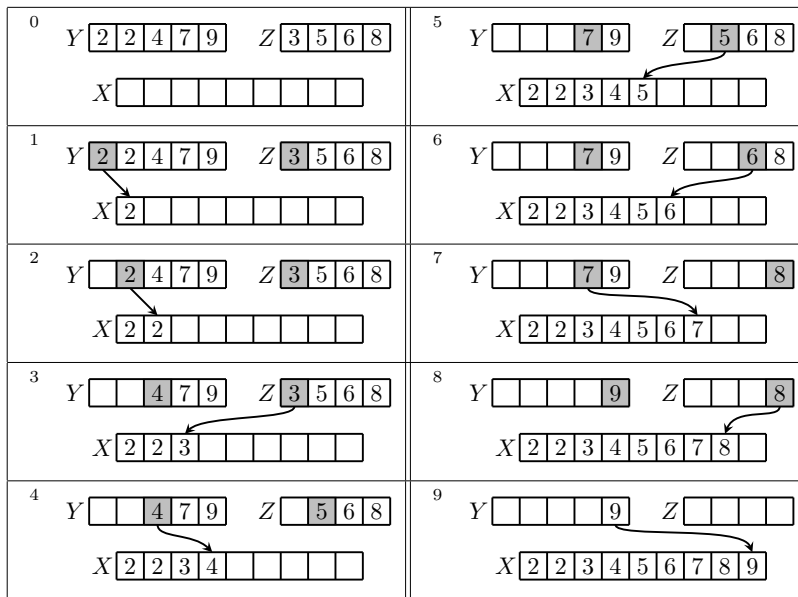
A big piece of the merge sort algorithm involves “merging” two sorted lists Y and Z into a single sorted list X . The idea is to start with $X = ()$, then continually compare the left-most entries of Y and Z , always removing the smaller one and appending it to the end of X , until Y and Z are used up.

To illustrate, say $Y = (2, 2, 4, 7, 9)$ and $Z = (3, 5, 6, 8)$. Step 1 compares the first entries of both Y and Z . The first entry 2 of Y is smaller than the first entry 3 of Z . So remove 2 from Y and make it the first entry of X , so $X = (2)$. Now Y is one entry shorter than it was previously.

Step 2 compares the first entries of Y and Z . Again, the first entry 2 of Y is smaller than the first entry 3 of Z . So remove 2 from Y and make it the next entry of X . Now $X = (2, 2)$ and Y has been shortened once again.

For step 3, compare the first entries of both Y and Z . This time the first entry 3 of Z is smaller than the first entry 4 of Y . So remove 3 from Z and make it the next entry of X . Now Z has been shortened.

Repeat until all entries of Y and Z have been removed and put onto X , as shown below. In the end, X is a sorted merging of Y and Z .



Notice that after step 8 all entries of Z have been removed. At this point we attach whatever entries are left on Y to the end of X . If at some point all entries of Y had been removed, then we’d append the remaining entries of Z to X .

Here is pseudocode for merging sorted lists Y and Z into a sorted list X . It is a procedure called **Merge** whose input is the two lists Y and Z , and whose output is the merged list X . Rather than actually removing entries from Y and Z , it maintains two indices i and j that indicate the current “first” entries of Y and Z , respectively. Initially $i = j = 1$. Then every time an entry of Y is “removed,” i increases by 1. Every time an entry of Z is “removed,” j increases by 1.

```

Procedure Merge( $Y, Z$ ).            $Y$  and  $Z$  are sorted lists to be merged


---

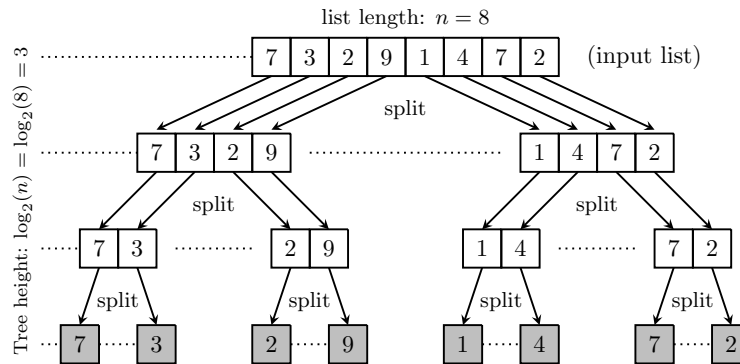

begin
   $i := 1$  .....index for list  $Y = (y_1, \dots, y_\ell)$  (initially  $y_i = y_1$ )
   $j := 1$  .....index for list  $Z = (z_1, \dots, z_m)$  (initially  $z_j = z_1$ )
   $k := 0$  .....index for merged list  $X$ 
   $X := (0, 0, \dots, 0)$  ..... initialize list  $X = (x_1, \dots, x_{\ell+m})$  (to be filled in)
  while  $(i \leq \ell) \vee (j \leq m)$  do
     $k := k + 1$  ..... advance to fresh entry of  $X$ 
    if  $(i \leq \ell) \wedge (j \leq m)$  then
      if  $y_i \leq z_j$  then
         $x_k := y_i$  .....  $x_k$  gets  $y_i$  because  $y_i \leq z_j$ 
         $i := i + 1$  ..... move to next entry of  $Y$ 
      else
         $x_k := z_j$  .....  $x_k$  gets  $z_j$  because  $z_j < y_i$ 
         $j := j + 1$  ..... move to next entry of  $Z$ 
      end
    else
      if  $i > \ell$  then
         $x_k := z_j$  } ..... if reached,  $Y$  is used up;
         $j := j + 1$  } ..... use entries of  $Z$ 
      else
         $x_k := y_i$  } ..... if reached,  $Z$  is used up;
         $i := i + 1$  } ..... use entries of  $Y$ 
      end
    end
  end
  return  $X$ 
end

```

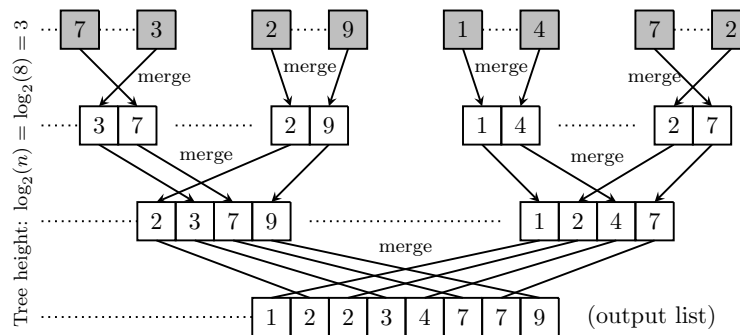
Let’s count steps. The procedure begins with four assignments, and ends with **return** X . So far that’s five steps outside the while-loop. The while-loop makes $\ell + m$ iterations. (Recall ℓ is Y ’s length and m is Z ’s length.) Each iteration makes 6 steps (3 boolean evaluations and 3 assignments), so in all the while-loop executes $6(\ell + m)$ steps.

In summary, if Y has ℓ entries and Z has m entries, then **Merge** merges them in $5 + 6(\ell + m)$ steps, so it is $O(\ell + m)$.

Now that we can merge two sorted lists into one sorted list with **Merge**, we can explain **MergeSort**, an algorithm that sorts a length- n list in $O(n \log_2(n))$ time. For simplicity, first consider a list whose length is a power of 2, like $n = 2^3 = 8$. Imagine that its entries are written on movable cards. Begin by splitting the list in half, into two smaller lists. Then split these half-lists in half, and continue until you can't split any further.

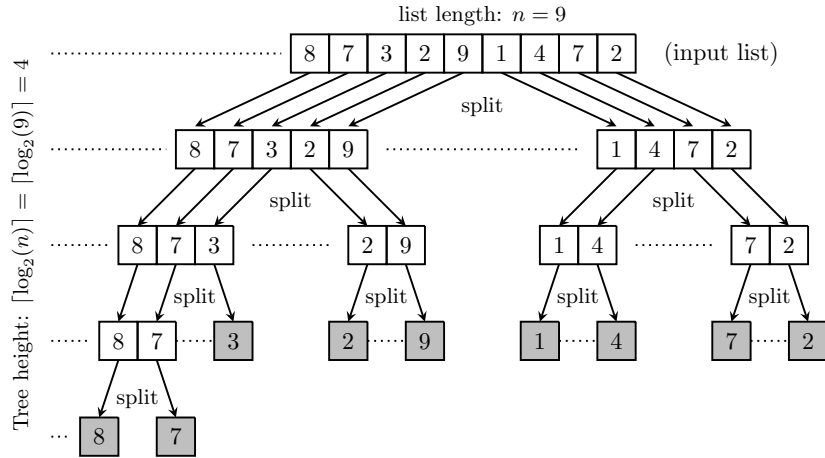


Now we have 8 lists of length 1, and each one is already sorted by default! Next merge these with **Merge** in the reverse order in which they were split.

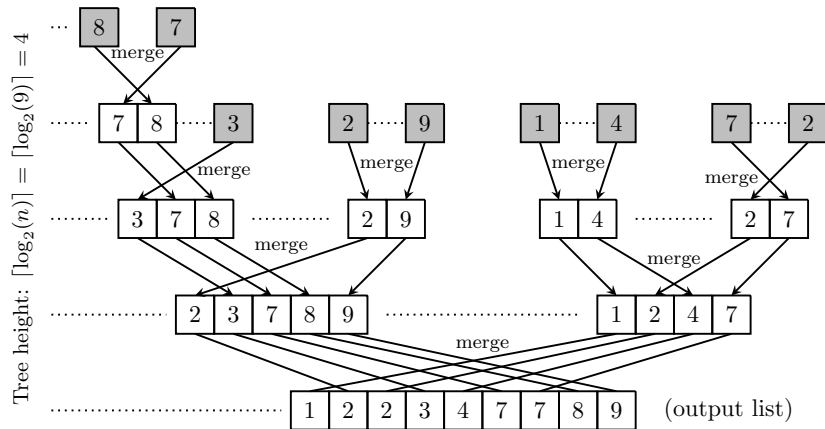


This is our sorted list. We will shortly write the **MergeSort** procedure to follow these steps. But first, let's count the number of steps (card movements) made in sorting the above example list of length $n = 2^k$. The first list-splitting phase made an inverted tree of height $\log_2(n) = \log_2(2^k) = k$ (in our example, $k = 3$). At each level of the tree we had to move all n cards, so the total number of moves in the "splitting" phase is $n \log_2(n)$. In the second "merging" phase we had to make another $n \log_2(n)$ moves. Summary: You can sort a list of length $n = 2^k$ cards with $2n \log_2(n)$ card movements.

The previous page's example was simplified by the assumption that the list's length was a power of 2. If this is not the case, then not every splitting operation will result in two equal-sized half-lists. You may have to split into two lists of lengths $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$, respectively. This is illustrated below with a list of length 9.



We had to go an extra level down to fully split our list into length-1 lists. The tree height increased to $\lceil \log_2(n) \rceil = \lceil \log_2(9) \rceil = 4$. The total number of card movements in this splitting phase is never more than n card moves per level, that is, $n \lceil \log_2(n) \rceil$. Reversing this process—but merging instead of splitting—gives our final sorted list in just $2n \lceil \log_2(n) \rceil$ card movements.



Summary: You can sort n cards with $2n \lceil \log_2(n) \rceil$ or fewer card moves.

Now we implement this idea and actually write `MergeSort`, a procedure that takes an input list X of numbers and returns X sorted into numeric order. That is, `MergeSort`(X) is a rearrangement of X into numeric order.

`MergeSort` uses the procedure `Merge`, described on page 480. But it is also recursive, calling itself. If the input list X happens to have length 0 or 1, then X is automatically already sorted, so `MergeSort`(X) just returns X . Otherwise `MergeSort` splits $X = (x_1, x_2, \dots, x_n)$ into two lists $Y = (x_1, x_2, \dots, x_{\lceil \frac{n}{2} \rceil})$ and $Z = (x_{\lceil \frac{n}{2} \rceil + 1}, \dots, x_{n-1}, x_n)$ that are each no longer than half the length of X . Then it returns `Merge`(`MergeSort`(Y), `MergeSort`(Z)).

```

Procedure MergeSort( $X$ ).            $X = (x_1, x_2, \dots, x_n)$  is list to be sorted
1 begin
2   if  $n \leq 1$  then
3     return  $X$  .....  $X$  has length 1 or 0, so it is already sorted
4   else
5      $Y := (x_1, x_2, \dots, x_{\lceil \frac{n}{2} \rceil})$  .....  $Y$  is half of  $X$ 
6      $Z := (x_{\lceil \frac{n}{2} \rceil + 1}, \dots, x_{n-1}, x_n)$  .....  $Z$  is other half
7     return Merge( MergeSort( $Y$ ), MergeSort( $Z$ ) )
8   end
9 end

```

Proposition 21.8. *For any list X of numbers, `MergeSort`(X) really does return X sorted into numeric order.*

Proof. We use strong induction on n to prove that `MergeSort` does indeed sort its input correctly. For the basis case, if $n = 0$ or $n = 1$, then its pseudocode reveals that `MergeSort`(X) returns X , unchanged, in line 3. This is the correct result, because as a list of length 0 or 1, X is already sorted.

For the inductive step we need to show that if $k > 1$ and `MergeSort` correctly sorts any list of length shorter than k , then `MergeSort` correctly sorts any list of length k .

We use direct proof. Suppose $k > 1$ and `MergeSort` correctly sorts any list of length shorter than k . Let X be a list of length k . Note that in this case `MergeSort`(X) splits X into two shorter lists Y and Z (lines 5 and 6), and then returns `Merge`(`MergeSort`(Y), `MergeSort`(Z)) in line 7. Now, Y and Z each has length shorter than k , so by the induction hypothesis, `MergeSort`(Y) and `MergeSort`(Z) are correct sortings of the two halves Y and Z of X . Thus the returned list `Merge`(`MergeSort`(Y), `MergeSort`(Z)) is a correct sorting of X . □

Next we prove that `MergeSort` is $O(n \log_2(n))$. (You may already believe this, based on the diagrams from several pages back.) The proof is a good illustration of strong induction and logarithm properties.

Proposition 21.9. MergeSort(X) is $O(n \log_2(n))$, where X has length n .

Proof. Our strategy is to show that if X has length n , then MergeSort(X) sorts X in no more than $f(n) = 2 + 12n \log_2(n)$ steps. This will imply that MergeSort(X) is $O(n \log_2(n))$, because if $n > 1$, then $1 \leq n \log_2(n)$, so

$$f(n) = 2 + 12n \log_2(n) \leq 2n \log_2(n) + 12n \log_2(n) = 14n \log_2(n),$$

and therefore $|f(n)| \leq A \cdot |n \log_2(n)|$ for $A = 14$ and $n \geq N = 1$.

So to complete the proof, we now prove that MergeSort(X) sorts X in no more than $f(n) = 2 + 12n \log_2(n)$ steps. The proof is by strong induction on n .

For the basis step, if X has length $n = 1$, then the pseudocode for MergeSort(X) shows that it returns X (already sorted) in 2 steps. As $f(1) = 2 + 12 \log_2(1) = 2 + 12 \cdot 0 = 2$, we see that indeed MergeSort(X) sorts X in no more than $f(1)$ steps.

Now for the inductive step. Let $n > 2$. Suppose that if X has length $k < n$, then MergeSort(X) takes no more than $f(k)$ steps.

Now assume X has length n . We must show that MergeSort(X) takes no more than $f(n)$ steps. Let's count steps for MergeSort(X). The first step is the boolean evaluation of $(n \leq 1)$ in line 2. Then the procedure goes straight to lines 5 and 6, and creates lists Y and Z . Note that Y has length $\lceil \frac{n}{2} \rceil$ and Z has length $\lfloor \frac{n}{2} \rfloor$, so it takes $\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor = n$ steps to fill in these new lists (one assignment per entry). So by line 6, the procedure has done $1 + n$ steps.

Next comes line 7, which returns Merge(MergeSort(Y), MergeSort(Z)). A lot happens here, and we need to track it all. First MergeSort(Y) and MergeSort(Z) are run, then Merge is run on their output, and the result is returned. By the inductive hypothesis, MergeSort(Y) takes no more than $f(\lceil \frac{n}{2} \rceil)$ steps, while MergeSort(Z) takes no more than $f(\lfloor \frac{n}{2} \rfloor)$ steps. By the remark at the bottom of page 480), it then takes at most $5 + 6(\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor) = 5 + 6n$ steps for merge to merge the two sorted lists. Lastly, the result is returned. So line 7 executes $f(\lceil \frac{n}{2} \rceil) + f(\lfloor \frac{n}{2} \rfloor) + 5 + 6n + 1$ steps

Adding this to the steps undertaken before line 7, we see that MergeSort(X) makes no more than $7 + 7n + f(\lceil \frac{n}{2} \rceil) + f(\lfloor \frac{n}{2} \rfloor)$ steps. We are aiming to show that this is no more than $f(n)$ steps. There are two cases.

Case 1. Suppose n is even. Then $\lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor = \frac{n}{2}$, so

$$\begin{aligned} 7 + 7n + f\left(\left\lceil \frac{n}{2} \right\rceil\right) + f\left(\left\lfloor \frac{n}{2} \right\rfloor\right) &= 7 + 7n + 2f\left(\frac{n}{2}\right) \\ &= 7 + 7n + 2\left(2 + 12 \frac{n}{2} \log_2\left(\frac{n}{2}\right)\right) \quad (\text{definition of } f) \\ &= 11 + 7n + 12n(\log_2(n) - \log_2(2)) \quad (\text{log property}) \\ &= (2 + 12n \log_2(n)) + (9 - 5n) \quad (\log_2(2) = 1) \\ &= f(n) + (9 - 5n) \quad (f(n) = 2 + 12n \log_2(n)) \\ &\leq f(n). \quad (\text{because } 9 - 5n \leq 0) \end{aligned}$$

Thus if X has even length n , then $\text{MergeSort}(X)$ sorts it in $f(n)$ or fewer steps.

Case 2. Suppose n is odd. Then $\lceil \frac{n}{2} \rceil = \frac{n+1}{2}$ and $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$, so $\text{MergeSort}(X)$ makes no more than

$$7 + 7n + f\left(\left\lceil \frac{n}{2} \right\rceil\right) + f\left(\left\lfloor \frac{n}{2} \right\rfloor\right) = 7 + 7n + f\left(\frac{n+1}{2}\right) + f\left(\frac{n-1}{2}\right)$$

steps. We need to show

$$7 + 7n + f\left(\frac{n+1}{2}\right) + f\left(\frac{n-1}{2}\right) \leq f(n).$$

This is somewhat more complicated than the previous case, and the simplifications are not as nice. An alternate approach is to show that the function

$$g(n) = f(n) - \left(7 + 7n + f\left(\frac{n+1}{2}\right) + f\left(\frac{n-1}{2}\right)\right)$$

is positive for $n \geq 3$. To do this, confirm $g(3) > 0$ and then use calculus to show that $g(n)$ increases for $n \geq 3$. We leave the details to the reader. \square

Solutions for Chapter 21

1. Show that $f(n) = 3 + n + 2^n$ is $O(2^n)$.

Solution: As long as $n > 2$ we have $3 \leq 2^n$ and $n \leq 2^n$, so $|f(n)| = |3 + n + 2^n| \leq |2^n + 2^n + 2^n| = |3 \cdot 2^n| = 3 \cdot |2^n|$. Therefore, for $n > N = 2$ and $A = 3$ we have $|f(n)| \leq A \cdot |2^n|$, so by definition $f(n)$ is $O(2^n)$.

3. Show that $f(n) = 25 + 8n + \log_2(n)$ is $O(n)$.

Solution: If $n > 4$, then $25 \leq 8n$ and $\log_2(n) \leq 8n$, so $|f(n)| = |25 + 8n + \log_2(n)| \leq |8n + 8n + 8n| = |24n| = 24 \cdot |n|$. Therefore, for $n > N = 4$ and $A = 24$ we have $|f(n)| \leq A \cdot |n|$, so by definition $f(n)$ is $O(n)$.

5. Show that $f(n) = n \log_2(n)$ is $O(n^2)$, but n^2 is not $O(n \log_2(n))$.

Solution: If $n > 1$, then $n \leq 2^n$, so $\log_2(n) \leq \log_2(2^n) = n$. Hence $n \log_2(n) \leq n \cdot n = n^2$. Thus for $n > N = 1$ and $A = 1$, we have $|n \log_2(n)| \leq A \cdot |n^2|$, so $n \log_2(n)$ is $O(n^2)$.

Next, suppose for the sake of contradiction that n^2 is $O(n \log_2(n))$. Then there are positive numbers N and A for which $n^2 \leq A n \log_2(n)$ whenever $n > N$. So if $n > N$, then $n \leq A \log_2(n)$. Thus n is $O(\log_2(n))$, contradicting Proposition 21.5.

7. Show that the function $f(n) = 2^n$ is $O(n!)$, but $n!$ is not $O(2^n)$.

Solution: If $n > N = 2$, then $|2^n| = \underbrace{2 \cdot 2 \cdot \dots \cdot 2}_n \leq \underbrace{n(n-1)(n-2) \dots \cdot 3 \cdot 2 \cdot 1}_n =$

$1 \cdot |n!|$. This means 2^n is $O(n!)$. Next, suppose for the sake of contradiction that $n!$ is $O(2^n)$. Then there are positive numbers N and A for which $n! \leq A \cdot 2^n$ whenever $n > N$. So if $n > N$, then $n! \leq A \cdot 2^n$, and thus $A \leq \frac{2 \cdot 2 \cdot \dots \cdot 2}{n(n-1)(n-2) \dots \cdot 3 \cdot 2 \cdot 1} < \frac{2 \cdot 2 \cdot \dots \cdot 2}{n \cdot n \cdot \dots \cdot n} = \left(\frac{2}{n}\right)^n$. Now, if $n > \frac{2}{A}$, then $\frac{2}{n} < A$. Further, if $n > 2$, then $\frac{2}{n} < 1$, and so $\left(\frac{2}{n}\right)^n < \frac{2}{n}$. Consequently, if $n \geq \max\{N, \frac{2}{A}, 2\}$, then $\left(\frac{2}{n}\right)^n < A$. This contradicts the fact (established above) that $A < \left(\frac{2}{n}\right)^n$ for all $n > N$.

9. Show that F_n is $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$, and that $\left(\frac{1+\sqrt{5}}{2}\right)^n$ is $O(F_n)$.

Outline: First use induction to prove that $\left(\frac{1+\sqrt{5}}{2}\right)^a + \left(\frac{1+\sqrt{5}}{2}\right)^{a+1} = \left(\frac{1+\sqrt{5}}{2}\right)^{a+2}$ for any positive integer a . With this fact, use induction again to show that $F_n \leq \left(\frac{1+\sqrt{5}}{2}\right)^{n-1} \leq F_{n+2}$ for any positive integer n . Now let $A = \frac{1+\sqrt{5}}{2}$. Then $F_n \leq \frac{1}{A} \left(\frac{1+\sqrt{5}}{2}\right)^n$ for all n , so F_n is $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$. Also, $\left(\frac{1+\sqrt{5}}{2}\right)^n \leq A^3 F_n$ for $n \geq 3$, which means $\left(\frac{1+\sqrt{5}}{2}\right)^n$ is $O(F_n)$.

11. Show that if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L$, then $f(n)$ is $O(g(n))$.

Suppose $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L$. Then $\lim_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| = |L|$, so $\lim_{n \rightarrow \infty} \frac{|f(n)|}{|g(n)|} = |L|$. Choose a number $\epsilon > 0$ for which $|L| + \epsilon \neq 0$. By definition of the limit, there is a number $N > 0$ for which $\left| \frac{|f(n)|}{|g(n)|} \right| \leq |L| + \epsilon$ for all $n > N$. Therefore $|f(n)| \leq (|L| + \epsilon) \cdot |g(n)|$ for all $n > N$, which means $f(n)$ is $O(g(n))$.

13. Use Proposition 21.2 to prove Proposition 21.4.

Using L'Hôpital's rule repeatedly, $\lim_{n \rightarrow \infty} \frac{n^d}{b^n} = \lim_{n \rightarrow \infty} \frac{dn^{d-1}}{\ln(b)b^n} = \lim_{n \rightarrow \infty} \frac{d(d-1)n^{d-2}}{\ln(b)^2 b^n} = \dots = \lim_{n \rightarrow \infty} \frac{d!}{\ln(b)^d b^n} = 0$, so n^d is $O(b^n)$. On the other

$$\begin{aligned} \text{hand, } \lim_{n \rightarrow \infty} \frac{b^n}{n^d} &= \lim_{n \rightarrow \infty} \frac{\ln(b)b^n}{dn^{d-1}} = \lim_{n \rightarrow \infty} \frac{\ln(b)^2 b^n}{d(d-1)n^{d-2}} = \lim_{n \rightarrow \infty} \frac{\ln(b)^3 b^n}{d(d-1)(d-2)n^{d-3}} = \dots \\ &= \lim_{n \rightarrow \infty} \frac{\ln(b)^d b^n}{d!} = \infty, \text{ so } b^n \text{ is not } O(n^d). \end{aligned}$$

- 15.** Show the relation \prec is a transitive relation on the set of all real-valued functions on $(0, \infty)$. That is, show that $f(n) \prec g(n)$ and $g(n) \prec h(n)$ implies $f(n) \prec h(n)$.
- Suppose $f(n) \prec g(n)$ and $g(n) \prec h(n)$. This means $f(n)$ is $O(g(n))$ but $g(n)$ is not $O(f(n))$, and $g(n)$ is $O(h(n))$ but $h(n)$ is not $O(g(n))$. Because $f(n)$ is $O(g(n))$, there are positive numbers N_1 and A_1 for which $|f(n)| \leq A_1 \cdot |g(n)|$ for all $n > N_1$. Because $g(n)$ is $O(h(n))$, there are positive numbers N_2 and A_2 for which $|g(n)| \leq A_2 \cdot |h(n)|$ for all $n > N_2$. Now put $N = \max\{N_1, N_2\}$ and $A = A_1 A_2$. Then for $n > N$ we have $|f(n)| \leq A_1 \cdot |g(n)| \leq A_1 \cdot A_2 \cdot |h(n)| = A \cdot |h(n)|$. Therefore $f(n)$ is $O(h(n))$.
- To show that $f(n) \prec h(n)$, it remains to show that $h(n)$ is not $O(f(n))$. Suppose to the contrary that $h(n)$ is $O(f(n))$. Then there are positive numbers N and A for which $|h(n)| \leq A \cdot |f(n)|$ for all $n > N$. In particular, for $n > \max\{N, N_1\}$ we have $|h(n)| \leq A \cdot |f(n)| \leq AA_1 \cdot |g(n)|$. This means that $h(n)$ is $O(g(n))$, a contradiction.