CLRS Introduction to algorithms notes

Nathan Warner

Computer Science
Northern Illinois University
United States

Contents

The role of algorithms in computing

1.1 Algorithms

Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output

We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

1.1.1 The sorting problem

For example, we might need to sort a sequence of numbers into nondecreasing order. Here is how we formally define the sorting problem

Input: A sequence of n numbers $\langle a_1, a_2, ..., a_n \rangle$

Output: A permutation $\langle a'_1, a'_2, ..., a'_n \rangle$ such that $a'_1 \leqslant a'_2 \leqslant ... \leqslant a'_n$

An **instance of a problem** consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

An algorithm is said to be correct if, for every input instance, it halts with the correct output. We say that a correct algorithm solves the given computational problem.

1.2 Data structures

A data structure is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them.

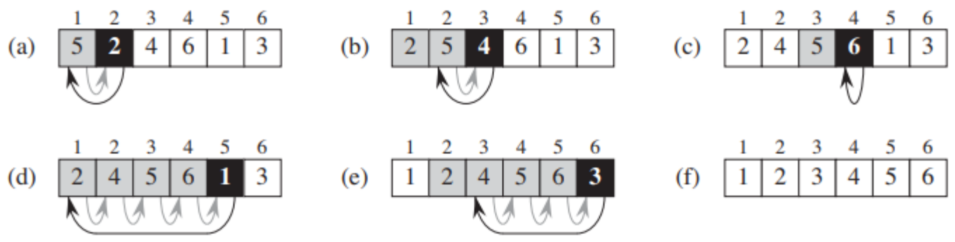
Prelude

2.1 Insertion sort

Our first algorithm, *insertion sort*, solves the sorting problem. The numbers that we wish to sort are also known as the keys. Although conceptually we are sorting a sequence, the input comes to us in the form of an array with n elements.

We start with insertion sort, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left, as illustrated in Figure 2.1. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table

We present our pseudocode for insertion sort as a procedure called INSERTION-SORT, which takes as a parameter an array $A[1...n]$ containing a sequence of length n that is to be sorted. (In the code, the number n of elements in A is denoted by $A.length$.) The algorithm sorts the input numbers ***in place:*** it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time. The input array A contains the sorted output sequence when the INSERTION-SORT procedure is finished



```
INSERTION-SORT(A)
    for j = 2 to A.length
        key = A[j]
        // Insert A[j] into the sorted sequence A[1...j-1]
        i = j-1
        while i > 0 and A[i] > key
            A[i+1] = A[i]
            i = i-1
        A[i+1] = key
```

2.2 Loop invariants and the correctness of insertion sort

At the beginning of each iteration of the `for` loop, which is indexed by j, the subarray consisting of elements $A[1 \ldots j-1]$ constitutes the currently sorted hand, and the remaining subarray $A[j+1 \ldots n]$ corresponds to the pile of cards still on the table. In fact, elements $A[1 \ldots j-1]$ are the elements *originally* in positions 1 through $j-1$, but now in sorted order. We state these properties of $A[1 \ldots j-1]$ formally as a ***loop invariant***:

> At the start of each iteration of the `for` loop of lines 1–8, the subarray $A[1 \ldots j-1]$ consists of the elements originally in $A[1 \ldots j-1]$, but in sorted order.

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

1. **Initialization:** It is true prior to the first iteration of the loop.
2. **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
3. **Termination:** When the loop terminates, the invariant gives us a useful property to show the algorithm is correct.

When the first two properties hold, the loop invariant is true prior to every iteration of the loop

Note the similarity to mathematical induction, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration corresponds to the base case, and showing that the invariant holds from iteration to iteration corresponds to the inductive step

The third property is perhaps the most important one, since we are using the loop invariant to show correctness. Typically, we use the loop invariant along with the condition that caused the loop to terminate. The termination property differs from how we usually use mathematical induction, in which we apply the inductive step infinitely; here, we stop the "induction" when the loop terminates.

Let us see how these properties hold for insertion sort.

1. **Initialization**: We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$. The subarray $A[1 : j-1]$, therefore, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.
2. **Maintenance**: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the `for` loop works by moving $A[j-1], A[j-2], A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4–7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1 : j]$ then consists of the elements originally in $A[1 : j]$, but in sorted order. Incrementing j for the next iteration of the `for` loop then preserves the loop invariant.
3. **Termination**: Finally, we examine what happens when the loop terminates. The condition causing the `for` loop to terminate is that $j >$ A.length $= n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time.

Substituting $n+1$ for j in the wording of the loop invariant, we have that the subarray $A[1:n]$ consists of the elements originally in $A[1:n]$, but in sorted order. Observing that the subarray $A[1:n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

2.3 Pseudocode conventions

- Indentation indicates block structure.
- The looping constructs while, for, and repeat-until and the if-else conditional construct have interpretations similar to those in C, C++, Java, Python, and Pascal

  The loop counter retains its value after exiting the loop, unlike some situations that arise in C++, Java, and Pascal. Thus, immediately after a for loop, the loop counter's value is the value that first exceeded the for loop bound. We used this property in our correctness argument for insertion sort.
- We use the keyword **to** when a for loop increments its loop. counter in each iteration, and we use the keyword **downto** when a for loop decrements its loop counter. When the loop counter changes by an amount greater than 1, the amount of change follows the optional keyword **by**.
- The symbol "//" indicates that the remainder of the line is a comment
- A multiple assignment of the form $i = j = e$ assigns to both variables i and j the value of expression e; it should be treated as equivalent to the assignment $j = e$ followed by the assignment $i = j$.
- Variables (such as i, j , and key) are local to the given procedure. We shall not use global variables without explicit indication.
- We access array elements by specifying the array name followed by the index in square brackets. For example, $A[i]$ indicates the i-th element of the array A. The notation "::" is used to indicate a range of values within an array. Thus, $A[1:j]$ indicates the subarray of A consisting of the j elements $A[1], A[2], \ldots, A[j]$.

  Note: The notation $A[1..j]$ and $A[1:j]$ is used interchangably.
- We typically organize compound data into objects, which are composed of attributes. We access a particular attribute using the syntax found in many object-oriented programming languages: the object name, followed by a dot, followed by the attribute name. For example, we treat an array as an object with the attribute length indicating how many elements it contains. To specify the number of elements in an array A, we write $A.length$.

  We treat a variable representing an array or object as a pointer to the data representing the array or object. For all attributes f of an object x, setting $y = x$ causes $y.f$ to equal $x.f$. Moreover, if we now set $x.f = 3$, then afterward not only does $x.f$ equal 3, but $y.f$ equals 3 as well. In other words, x and y point to the same object after the assignment $y = x$.

  Our attribute notation can "cascade." For example, suppose that the attribute f is itself a pointer to some type of object that has an attribute g. Then the notation $x.f.g$ is implicitly parenthesized as $(x.f).g$. In other words, if we had assigned $y = x.f$, then $x.f.g$ is the same as $y.g$.

Sometimes, a pointer will refer to no object at all. In this case, we give it the special value `NIL`.

- We pass parameters to a procedure *by value*: the called procedure receives its own copy of the parameters, and if it assigns a value to a parameter, the change is *not* seen by the calling procedure. When objects are passed, the pointer to the data representing the object is copied, but the object's attributes are not. For example, if x is a parameter of a called procedure, the assignment $x = y$ within the called procedure is not visible to the calling procedure. The assignment $x.f = 3$, however, is visible. Similarly, arrays are passed by pointer, so that a pointer to the array is passed, rather than the entire array, and changes to individual array elements are visible to the calling procedure.

- A `return` statement immediately transfers control back to the point of call in the calling procedure. Most `return` statements also take a value to pass back to the caller. Our pseudocode differs from many programming languages in that we allow multiple values to be returned in a single `return` statement.

- The boolean operators "`and`" and "`or`" are *short circuiting.* That is, when we evaluate the expression "x `and` y" we first evaluate x. If x evaluates to `FALSE`, then the entire expression cannot evaluate to `TRUE`, and so we do not evaluate y. If, on the other hand, x evaluates to `TRUE`, we must evaluate y to determine the value of the entire expression. Similarly, in the expression "x `or` y" we evaluate the expression y only if x evaluates to `FALSE`. Short-circuiting operators allow us to write boolean expressions such as "$x \neq$ `NIL` `and` $x.f = y$" without worrying about what happens when we try to evaluate $x.f$ when x is `NIL`.

- The keyword `error` indicates that an error occurred because conditions were wrong for the procedure to have been called. The calling procedure is responsible for handling the error, and we do not specify what action to take.

2.4 Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process.

Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs.

we shall assume a generic oneprocessor, **random-access machine (RAM)** model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations

Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and would yield little insight into algorithm design and analysis. Yet we must be careful not to abuse the RAM model. For example, what if a RAM had an instruction that sorts? Then we could sort in just one instruction. Such a RAM would be unrealistic, since real computers do not have such instructions. Our guide, therefore, is how real computers are designed. The RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time.

The data types in the RAM model are integer and floating point (for storing real numbers). Although we typically do not concern ourselves with precision in this book, in some applications precision is crucial. We also assume a limit on the size of each word of data. For example, when working with inputs of size n, we typically assume that integers are represented by $clg\ n$ bits for some constant $c \geqslant 1$ We require $c \geqslant 1$ so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily.

Note: We will treat computation of 2^k as a constant-time operation when k is a small enough positive integer

RAM-model analyses are usually excellent predictors of performance on actual machines.

2.4.1 Analysis of insertion sort

The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers. Moreover, INSERTION-SORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are. In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input. To do so, we need to define the terms "running time" and "size of input" more carefully

The best notion for **input size** depends on the problem being studied. The most natural measure is the number of items in the input, for example, the array size n for sorting. For many other problems, such as multiplying two integers, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation. Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph. We shall indicate which input size measure is being used with each problem we study

The **running time** of an algorithm on a particular input is the number of primitive operations or "steps" executed. It is convenient to define the notion of step so that it is as machine-independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the ith line takes time c_i, where c_i is a constant. This viewpoint is in keeping with the RAM model, and it also reflects how the pseudocode would be implemented on most actual computers

In the following discussion, our expression for the running time of INSERTIONSORT will evolve from a messy formula that uses all the statement costs c_i to a much simpler notation that is more concise and more easily manipulated. This simpler notation will also make it easy to determine whether one algorithm is more efficient than another.

We start by presenting the **INSERTION-SORT** procedure with the time "cost" of each statement and the number of times each statement is executed. For each $j = 2, 3, \ldots, n$, where $n =$ A.length, we let t_j denote the number of times the `while` loop test in line 5 is executed for that value of j. When a `for` or `while` loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body. We assume that comments are not executable statements, and so they take no time.

```
 0  INSERTION-SORT(A)                           cost     times
 1      for j = 2 to A.length                   $c_1$      $n$
 2          key = A[j]                          $c_2$      $n-1$
 3                                              0        $n-1$
 4          // Insert A[j] into the
 5          // sorted sequence A[1...j-1]
 6          i = j-1                             $c_4$      $n-1$
 7          while i > 0 and A[i] > key          $c_5$      $\sum_{j=2}^{n} t_j$
 8              A[i+1] = A[i]                   $c_6$      $\sum_{j=2}^{n}(t_j-1)$
 9              i = i-1                         $c_7$      $\sum_{j=2}^{n}(t_j-1)$
10          A[i+1] = key                        $c_8$      $n-1$
```

The running time of the algorithm is the sum of the running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.

To compute $T(n)$, the running time of **INSERTION-SORT** on an input of n values, we sum the products of the cost and times columns, obtaining:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1).$$

Even for inputs of a given size, an algorithm's running time may depend on *which input of that size is given*. For example, in **INSERTION-SORT**, the best-case occurs if the array is already sorted. For each $j = 2, 3, \ldots, n$, we then find that $A[i] \leqslant$ key in line 5 when i has its initial value of $j-1$. Thus $t_j = 1$ for $j = 2, 3, \ldots, n$, and the best-case running time is

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

We can express this running time as $an + b$ for constants a and b that depend on the statement costs c_i; it is thus a *linear function of* n.

If the array is in reverse sorted order—that is, in decreasing order—the worst-case results. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1 \ldots j-1]$, and so $t_j = j$ for $j = 2, 3, \ldots, n$. Noting that

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (j-1) = \frac{n(n+1)}{2}$$

.

we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned}
T(n) = {} & c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
& + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
= {} & \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 \\
& + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
& - \left(c_2 + c_4 + c_5 + c_8\right) ..
\end{aligned}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i; it is thus a *quadratic function of* n.

Typically, as in insertion sort, the running time of an algorithm is fixed for a given input, although in later chapters we shall see some interesting "randomized" algorithms whose behavior can vary even for a fixed input.

2.5 Worst-case and average-case analysis

In our analysis of insertion sort, we looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. For the remainder of this book, though, we shall usually concentrate on finding only the worst-case running time, that is, the longest running time for any input of size n. We give three reasons for this orientation

- The worst-case running time of an algorithm gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. We need not make some educated guess about the running time and hope that it never gets much worse.
- For some algorithms, the worst case occurs fairly often
- The "average case" is often roughly as bad as the worst case.

In some particular cases, we shall be interested in the average-case running time of an algorithm

2.6 Order of growth

We used some simplifying abstractions to ease our analysis of the **INSERTION-SORT** procedure. First, we ignored the actual cost of each statement, using the constants c_i to represent these costs. Then, we observed that even these constants give us more detail than we really need: we expressed the worst-case running time as

$$an^2 + bn + c$$

for some constants a, b, and c that depend on the statement costs c_i. We thus ignored not only the actual statement costs, but also the abstract costs c_i.

We shall now make one more simplifying abstraction: it is the ***rate of growth***, or ***order of growth***, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large values of n. We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.

For insertion sort, when we ignore the lower-order terms and the leading term's constant coefficient, we are left with the factor of n^2 from the leading term. We write that insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of n-squared").

We usually consider one algorithm to be more efficient than another if its worstcase running time has a lower order of growth. Due to constant factors and lowerorder terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth. But for large enough inputs, a $\Theta(n^2)$ algorithm, for example, will run more quickly in the worst case than a $\Theta(n^3)$ algorithm.

2.7 Designing algorithms

We can choose from a wide range of algorithm design techniques. For insertion sort, we used an ***incremental approach***

we examine an alternative design approach, known as "divide-and-conquer"

2.7.1 The divide-and-conquer approach, mergesort

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a ***divide-and-conquer approach***: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem

The divide-and-conquer paradigm involves three steps at each level of the recursion:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows

1. **Divide:** Divide the n-element sequence to be sorted into two subsequences of $\frac{n}{2}$ elements each.
2. **Conquer:** Sort the two subsequences recursively using merge sort.
3. **Combine:** Merge the two sorted subsequences to produce the sorted answer

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. We merge by calling an auxiliary procedure **MERGE**(A, p, q, r), where A is an array and p, q, and r are indices into the array such that $p \leqslant q < r$. The procedure assumes that the subarrays $A[p \dots q]$ and $A[q+1 \dots r]$ are in sorted order. It *merges* them to form a single sorted subarray that replaces the current subarray $A[p \dots r]$.

Our **MERGE** procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the total number of elements being merged, and it works as follows. Returning to our card-playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto the sorted output pile. the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile. Computationally, each basic step takes constant time, since we are comparing just the two top cards. Since we perform at most n basic steps, merging takes $\Theta(n)$ time.

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. We place on the bottom of each pile a ***sentinel*** card, which contains a special value that we use to simplify our code. Here, we use ∞ as the sentinel value, so that whenever a card with ∞ is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly $r - p + 1$ cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

```
0   MERGE(A,p,q,r)
1       $n_1 = q - p + 1$
2       $n_2 = r - q$
3       let $L[1..n_1 + 1]$ and  $R[1..n_2 + 1]$ be new arrays
4       for $i = 1$ to $n_1$
5           $L[i] = A[p + i - 1]$
6       for $j = 1$ to $n_2$
7           $R[j] = A[q + j]$
8       $L[n_1 + 1] = \infty$
9       $R[n_2 + 1] = \infty$
10      $i = 1$
11      $j = 1$
12      for $k = p$ to $r$
13          if $L[i] \leqslant R[j]$
14              $A[k] = L[i]$
15              $i = i + 1$
16          else
17          $A[k] = R[j]$
18          $j = j + 1$
```

In detail, the MERGE procedure works as follows. Line 1 computes the length n_1 of the subarray $A[p \dots q]$, and line 2 computes the length n_2 of the subarray $A[q+1 \dots r]$. We create arrays L and R (”left” and ”right”), of lengths $n_1 + 1$ and $n_2 + 1$, respectively, in line 3; the extra position in each array will hold the sentinel. The **for** loop of lines 4–5 copies the subarray $A[p \dots q]$ into $L[1 \dots n_1]$, and the **for** loop of lines 6–7 copies the subarray $A[q+1 \dots r]$ into $R[1 \dots n_2]$. Lines 8–9 put the sentinels at the ends of the arrays L and R. Lines 10–17, illustrated in Figure 2.3, perform the $r - p + 1$ basic steps by maintaining the following loop invariant:

> At the start of each iteration of the **for** loop of lines 12–17, the subarray $A[p \dots k-1]$ contains the $k-p$ smallest elements of $L[1 \dots n_1+1]$ and $R[1 \dots n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

1. **Initialization:** Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p \dots k-1]$ is empty. This empty subarray contains the $k - p = 0$ smallest elements of L and R, and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

2. **Maintenance:** To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leqslant R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p \dots k-1]$ contains the $k-p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p \dots k]$ will contain the $k-p+1$ smallest elements. Incrementing k (in the **for** loop update) and i (in line 15) reestablishes the loop invariant for the

next iteration. If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

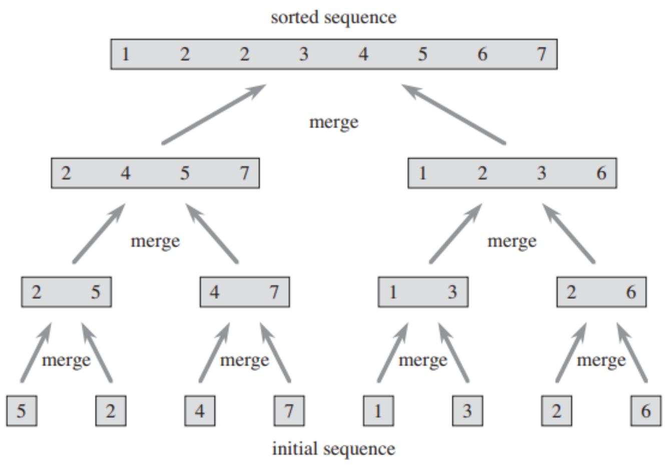
3. **Termination:** At termination, $k = r + 1$. By the loop invariant, the subarray $A[p \dots k-1]$, which is $A[p \dots r]$, contains the $k - p = r - p + 1$ smallest elements of $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$, in sorted order. The arrays L and R together contain $n_1 + n_2 + 2 = r - p + 3$ elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

To see that the MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$, observe that each of lines 1–3 and 8–11 takes constant time, the **for** loops of lines 4–7 take $\Theta(n_1 + n_2) = \Theta(n)$ time, and there are n iterations of the **for** loop of lines 12–17, each of which takes constant time.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT(A, p, r) sorts the elements in the subarray $A[p \dots r]$. If $p \geqslant r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p \dots r]$ into two subarrays: $A[p \dots q]$, containing $\lfloor n/2 \rfloor$ elements, and $A[q + 1 \dots r]$, containing $\lceil n/2 \rceil$ elements.

```
MERGE-SORT(A,p,r)
    if $p < r$
        $q = \lfloor (p+r)/2 \rfloor$
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1, r)
        MERGE(A,p,q,r)
```

To sort the entire sequence $A = \langle A[1], A[2], \dots, A[n] \rangle$, we make the initial call MERGE-SORT$(A, 1, A.\text{length})$, where once again $A.\text{length} = n$. Figure 2.4 illustrates the operation of the procedure bottom-up when n is a power of 2. The algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length $n/2$ are merged to form the final sorted sequence of length n.



2.7.2 Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call to itself, we can often describe its running time by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic paradigm. As before, we let $T(n)$ be the running time on a problem of size n. If the problem size is small enough, say $n \leqslant c$ for some constant c, the straightforward solution takes constant time, which we write as $\Theta(1)$. Suppose that our division of the problem yields a subproblems, each of which is $1/b$ the size of the original. (For merge sort, both a and b are 2, but we shall see many divide-and-conquer algorithms in which $a \neq b$.) It takes time $T(n/b)$ to solve one subproblem of size n/b, and so it takes time $aT(n/b)$ to solve a of them. If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leqslant c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}.$$

2.7.3 Analysis of merge sort

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of 2. Each divide step then yields two subsequences of size exactly $\frac{n}{2}$. we shall see that this assumption does not affect the order of growth of the solution to the recurrence.

We reason as follows to set up the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic paradigm. As before, we let $T(n)$ be the running time on a problem of size n. If the problem size is small enough, say $n \leqslant c$ for some constant c, the straightforward solution takes constant time, which we write as $\Theta(1)$. Suppose that our division of the problem yields a subproblems, each of which is $1/b$ the size of the original. (For merge sort, both a and b are 2, but we shall see many divide-and-conquer algorithms in which $a \neq b$.) It takes time $T(n/b)$ to solve one subproblem of size n/b, and so it takes time $aT(n/b)$ to solve a of them. If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leqslant c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

- **Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

- **Conquer:** We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
- **Combine:** We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of n, that is, $\Theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

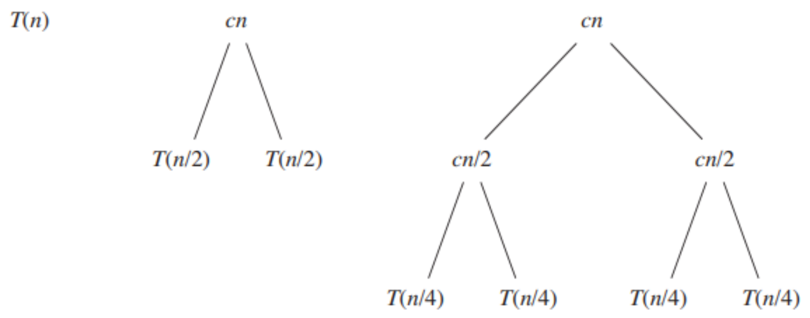
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases} \tag{2.1}$$

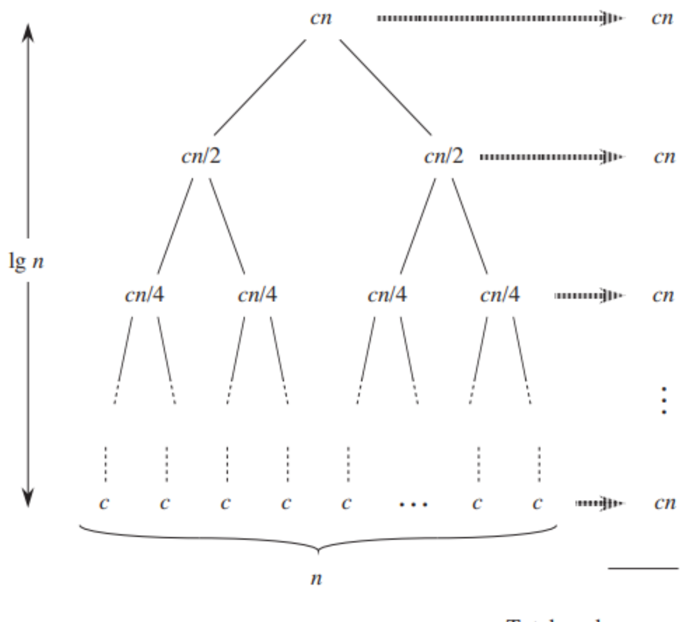
In the future, we shall see the "master theorem," which we can use to show that $T(n)$ is $\Theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$. Because the logarithm function grows more slowly than any linear function, for large enough inputs, merge sort, with its $\Theta(n \lg n)$ running time, outperforms insertion sort, whose running time is $\Theta(n^2)$, in the worst case.

We do not need the master theorem to intuitively understand why the solution to the recurrence (2.1) is $T(n) = \Theta(n \lg n)$. Let us rewrite recurrence (2.1) as

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{if } n > 1, \end{cases} \tag{2.2}$$

where the constant c represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps.




cn
cn
$cn/2$
$cn/2$
cn
$\lg n$
$cn/4$
$cn/4$
$cn/4$
$cn/4$
cn
c c c c c … c c
cn
n
Total: $cn \lg n + cn$

(d)

Growth of Functions

Although we can sometimes determine the exact running time of an algorithm, as we did for insertion sort, the extra precision is not usually worth the effort of computing it. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.

When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the asymptotic efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

3.1 Asymptotic notation

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $\mathbb{N}$

3.2 Asymptotic notation, functions, and running times

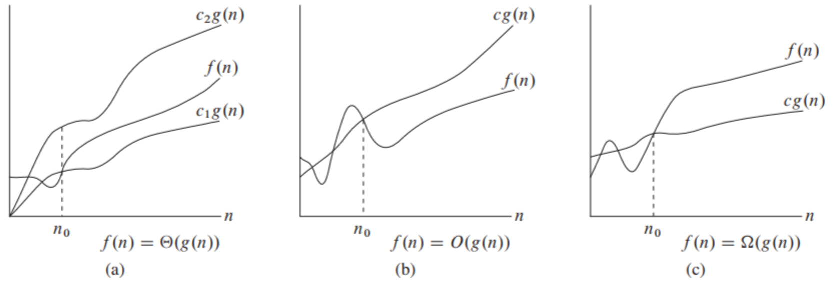
The functions to which we apply asymptotic notation will usually characterize the running times of algorithms. But asymptotic notation can apply to functions that characterize some other aspect of algorithms (the amount of space they use, for example), or even to functions that have nothing whatsoever to do with algorithms

Even when we use asymptotic notation to apply to the running time of an algorithm, we need to understand *which* running time we mean. Sometimes we are interested in the worst-case running time. Often, however, we wish to characterize the running time no matter what the input. In other words, we often wish to make a blanket statement that covers all inputs, not just the worst case. We shall see asymptotic notations that are well suited to characterizing running times no matter what the input.

3.2.1 Θ-notation

For a given function $g(n)$ we denote by $\Theta(g(n))$ the *set of functions*

$$\Theta(g(n)) = \{f(n) : \exists\ c_1, c_2, n_0 > 0\ s.t\ 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\ \forall\ n \geqslant n_0\}.$$



Because $\Theta(g(n))$ is a set, we could write "$f(n) \in \Theta(g(n))$" to indicate that $f(n)$ is a member of $\Theta(g(n))$. Instead, we will usually write "$f(n) = \Theta(g(n))$" to express the same notion. You might be confused because we abuse equality in this way, but we shall see later in this section that doing so has its advantages.

For all $n \geqslant n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

The definition of $\Theta(g(n))$ requires that every member $f(n) \in \Theta(g(n))$ be asymptotically nonnegative, that is, that $f(n)$ be nonnegative whenever n is sufficiently large. (An asymptotically positive function is one that is positive for all sufficiently large n.) Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty. We shall therefore assume that every function used within Θ-notation is asymptotically nonnegative.

Earlier we introduced an informal notion of Θ-notation that amounted to throwing away lower-order terms and ignoring the leading coefficient of the highest-order term. Let us briefly justify this intuition by using the formal definition to show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. To do so, we must determine positive constants $c_1, c_2, and n_0$ such that

$$c_1 n^2 \leqslant \frac{1}{2}n^2 - 3n \leqslant c_2 n^2.$$

For all $n \geqslant n_0$. We have

$$\begin{aligned} c_1 n^2 \leqslant& \frac{1}{2}n^2 - 3n \leqslant c_2 n^2 \\ \implies c_1 \leqslant& \frac{1}{2} - \frac{3}{n} \leqslant c_2. \end{aligned}$$

Analyzing the right hand inequality, we see that as $n \to \infty$, $\frac{1}{2} - \frac{3}{n} \to \frac{1}{2}$. Thus, the right hand inequality holds for all $n \geqslant 1$ when $c_2 \geqslant \frac{1}{2}$. The first value of n that makes $\frac{1}{2} - \frac{3}{n} > 0$ is when $n = 7$, we see

$$\frac{1}{2} - \frac{3}{7} = \frac{1}{14}.$$

Thus, we can make the left hand inequality hold for any $n \geqslant 7$ by choosing $c_1 \leqslant \frac{1}{14}$. Therefore, by choosing $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$, and $n_0 = 7$, we can verify that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Certainly, other choices for the constants exist, but the important thing is that *some* choice exists.

Note that a different function belonging to $\Theta(n^2)$ would usually require different constants

We can also use the formal definition to verify that $6n^3 \neq \Theta(n^2)$. Suppose for the sake of contradiction that such c_1, c_2, and n_0 exist. Then,

$$\begin{aligned} c_1 n^2 \leqslant 6n^3 \leqslant c_2 n^2 \\ \implies \frac{c_1}{n} \leqslant 6 \leqslant \frac{c_2}{n}. \end{aligned}$$

If we examine the right hand inequality, we see that as $n \to \infty$, $\frac{c_2}{n} \to 0$ for all choices of c_2. Thus, the right side cannot possibly hold for large n, since c_2 is constant

Intuitively, the lower-order terms of an asymptotically positive function can be ignored in determining asymptotically tight bounds because they are insignificant for large n. When n is large, even a tiny fraction of the highest-order term suffices to dominate the lower-order terms. Thus, setting c_1 to a value that is slightly smaller than the coefficient of the highest-order term and setting c_2 to a value that is slightly larger permits the inequalities in the definition of Θ-notation to be satisfied. The coefficient of the highest-order term can likewise be ignored, since it only changes c_1 and c_2 by a constant factor equal to the coefficient.

Since any constant is a degree-0 polynomial, we can express any constant function as $\Theta(n^0)$, or $\Theta(1)$

3.2.2 O-notation

The Θ-notation asymptotically bounds a function from above and below. When we have only an asymptotic upper bound, we use O-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the set of functions.

$$O(g(n)) = \{f(n) :\ \exists\ c, n_0 > 0\ s.t\ 0 \leqslant f(n) \leqslant cg(n)\ \forall\ n \geqslant n_0\}.$$

We use O-notation to give an upper bound on a function, to within a constant factor.

We write $f(n) = O(g(n))$ to indicate that a function $f(n)$ is a member of the set $O(g(n))$. Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since Θ-notation is a stronger notion than O-notation. Written set-theoretically, we have

$$\Theta(g(n)) \subseteq O(g(n)).$$

Thus, our proof that any quadratic function an^2+bn+c, where $a > 0$, is in $\Theta(n^2)$ also shows that any such quadratic function is in $O(n^2)$. What may be more surprising is that when $a > 0$, any *linear* function $an + b$ is in $O(n^2)$, which is easily verified by taking $c = a + |b|$ and $n_0 = \max(1, -b/a)$.

Since O-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, we have a bound on the running time of the algorithm on every input—the blanket statement we discussed earlier. Thus, the $O(n^2)$ bound on the worst-case running time of insertion sort also applies to its running time on every input. The $\Theta(n^2)$ bound on the worst-case running time of insertion sort, however, does not imply a $\Theta(n^2)$ bound on the running time of insertion sort on every input. For example, we saw in Chapter 2 that when the input is already sorted, insertion sort runs in $\Theta(n)$ time.

Technically, it is an abuse to say that the running time of insertion sort is $O(n^2)$, since for a given n, the actual running time varies, depending on the particular input of size n. When we say "the running time is $O(n^2)$," we mean that there is a function $f(n)$ that is $O(n^2)$ such that for any value of n, no matter what particular input of size n is chosen, the running time on that input is bounded from above by the value $f(n)$. Equivalently, we mean that the worst-case running time is $O(n^2)$.

3.2.3 Ω-notation

Just as O-notation provides an asymptotic upper bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions.

$$\Omega(g(n)) = \{f(n) : \ \exists\ c, n_0 > 0 \ s.t \ 0 \leqslant cg(n) \leqslant f(n) \ \forall\ n \geqslant n_0\}.$$

When we say that the running time (no modifier) of an algorithm is $\Omega(g(n))$, we mean that no matter what particular input of size n is chosen for each value of n, the running time on that input is at least a constant times $g(n)$, for sufficiently large n. Equivalently, we are giving a lower bound on the best-case running time of an algorithm. For example, the best-case running time of insertion sort is $\Omega(n)$, which implies that the running time of insertion sort is $\Omega(n)$.

The running time of insertion sort therefore belongs to both $\Omega(n)$ and $O(n^2)$, since it falls anywhere between a linear function of n and a quadratic function of n. Moreover, these bounds are asymptotically as tight as possible: for instance, the running time of insertion sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time (e.g., when the input is already sorted). It is not contradictory, however, to say that the worst-case running time of insertion sort is $\Omega(n^2)$, since there exists an input that causes the algorithm to take $\Omega(n^2)$ time.

3.2.4 Theta-Characterization Theorem

Theorem. For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

In practice, rather than using this theorem to obtain asymptotic upper and lower bounds from asymptotically tight bounds, we usually use it to prove asymptotically tight bounds from asymptotic upper and lower bounds.

3.3 Asymptotic notation in equations and inequalities

We have already seen how asymptotic notation can be used within mathematical formulas. For example, in introducing O-notation, we wrote "$n = O(n^2)$." We might also write $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. How do we interpret such formulas?

When the asymptotic notation stands alone (that is, not within a larger formula) on the right-hand side of an equation (or inequality), as in $n = O(n^2)$, we have already defined the equal sign to mean set membership: $n \in O(n^2)$. In general, however, when asymptotic notation appears in a formula, we interpret it as stand- ing for some anonymous function that we do not care to name. For example, the formula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$, where $f(n)$ is some function in the set $\Theta(n)$. In this case, we let $f(n) = 3n + 1$, which indeed is in $\Theta(n)$.

Using asymptotic notation in this manner can help eliminate inessential detail and clutter in an equation. For example, in Chapter 2 we expressed the worst-case running time of merge sort as the recurrence

$$T(n) = 2T(n/2) + \Theta(n).$$

If we are interested only in the asymptotic behavior of $T(n)$, there is no point in specifying all the lower-order terms exactly; they are all understood to be included in the anonymous function denoted by the term $\Theta(n)$.

The number of anonymous functions in an expression is understood to be equal to the number of times the asymptotic notation appears. For example, in the expression

$$\sum_{i=1}^{n} O(i).$$

there is only a single anonymous function (a function of i). This expression is thus not the same as

$$O(1) + O(2) + ... + O(n).$$

Which doesn't really have a clean interpretation. In some cases, asymptotic notation appears on the left-hand side of an equation, as in

$$2n^2 + \Theta(n) = \Theta(n^2).$$

We interpret such equations using the following rule: *No matter how the anony- mous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid.* Thus, our example means that for any function $f(n) \in \Theta(n)$, there is some func- tion $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$ for all n. In other words, the right-hand side of an equation provides a *coarser* level of detail than the left-hand side. We can chain together a number of such relationships, as in

$$2n^2 + 3n + 1 \quad = \quad 2n^2 + \Theta(n) \quad = \quad \Theta(n^2)\,.$$

We can interpret each equation separately by the rules above. The first equa- tion says that there is *some* function $f(n) \in \Theta(n)$ such that $2n^2 + 3n + 1 = 2n^2 + f(n)$ for all n. The second equation says that for *any* function $g(n) \in \Theta(n)$ (such as the $f(n)$ just mentioned), there is some function $h(n) \in \Theta(n^2)$ such that $2n^2 + g(n) = h(n)$ for all n. Note that this interpretation implies that $2n^2 + 3n + 1 = \Theta(n^2)$, which is what the chaining of equations intuitively gives us.

3.4 *o*-notation

The asymptotic upper bound provided by O-notation may or may not be asymp- totically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymp- totically tight. We formally define $o(g(n))$ ("little-oh of g of n") as the set

$$\begin{aligned} o(g(n)) = \{f(n) : &\text{ for any positive constant } c > 0, \text{ there exists a constant} \\ & n_0 > 0 \text{ such that } 0 \leqslant f(n) < cg(n) \text{ for all } n \geqslant n_0\}. \end{aligned}$$

For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \leqslant f(n) \leqslant cg(n)$ holds for *some* con- stant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \leqslant f(n) < cg(n)$ holds for *all* constants $c > 0$. Intuitively, in o-notation, the function $f(n)$ becomes insignifi- cant relative to $g(n)$ as n approaches infinity; that is,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

3.5 ω-notation

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight. One way to define it is by

$$f(n) \in \omega(g(n)) \text{ if and only if } g(n) \in o(f(n)) \,.$$

Formally, however, we define $\omega(g(n))$ ("little-omega of g of n") as the set

$\omega(g(n)) = \{f(n) :$ for any positive constant $c > 0,$ there exists a constant $n_0 > 0$ such that $0 \leqslant cg(n) < f(n)$ for all

For example, $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that

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

if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

3.6 Properties of asymptotic notation

3.6.1 Transitivity

$$\begin{aligned}
f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad & \Longrightarrow \; f(n) = \Theta(h(n)) \\
f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \quad & \Longrightarrow \; f(n) = O(h(n)) \\
f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad & \Longrightarrow \; f(n) = \Omega(h(n)) \\
f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \quad & \Longrightarrow \; f(n) = o(h(n)) \\
f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \quad & \Longrightarrow \; f(n) = \omega(h(n)).
\end{aligned}$$

3.6.2 Reflexivity

$$\begin{aligned}
f(n) &= \Theta(f(n)) \\
f(n) &= O(f(n)) \\
f(n) &= \Omega(f(n)).
\end{aligned}$$

3.6.3 Symmetry

$$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n)).$$

3.6.4 Transpose symmetry

$$\begin{aligned}
f(n) = O(g(n)) &\iff g(n) = \Omega(f(n)) \\
f(n) = o(g(n)) &\iff g(n) = \omega(f(n)).
\end{aligned}$$

3.7 Comparing functions, asymptotically smaller, and asymptotically larger

Because these properties hold for asymptotic notations, we can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

$$\begin{aligned}
f(n) &= O(g(n)) && \text{is like} \quad a \leqslant b\,, \\
f(n) &= \Omega(g(n)) && \text{is like} \quad a \geqslant b\,, \\
f(n) &= \Theta(g(n)) && \text{is like} \quad a = b\,, \\
f(n) &= o(g(n)) && \text{is like} \quad a < b\,, \\
f(n) &= \omega(g(n)) && \text{is like} \quad a > b\,.
\end{aligned}$$

We say that $f(n)$ is *asymptotically smaller* than $g(n)$ if $f(n) = o(g(n))$, and $f(n)$ is *asymptotically larger* than $g(n)$ if $f(n) = \omega(g(n))$.

One property of real numbers, however, does not carry over to asymptotic nota- tion:

Trichotomy: For any two real numbers a and b, exactly one of the following must hold: $a < b$, $a = b$, or $a > b$.

Although any two real numbers can be compared, not all functions are asymptot- ically comparable. That is, for two functions $f(n)$ and $g(n)$, it may be the case that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds. For example, we cannot compare the functions n and $n^{1+\sin n}$ using asymptotic notation, since the value of the exponent in $n^{1+\sin n}$ oscillates between 0 and 2, taking on all values in between.

3.8 Standard notations and common functions

3.8.1 Monotonicity

A function $f(n)$ is *monotonically increasing* if $m \leqslant n$ implies $f(m) \leqslant f(n)$. Similarly, it is *monotonically decreasing* if $m \leqslant n$ implies $f(m) \geqslant f(n)$. A function $f(n)$ is *strictly increasing* if $m < n$ implies $f(m) < f(n)$ and *strictly decreasing* if $m < n$ implies $f(m) > f(n)$.

3.8.2 Floors and ceilings

For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the floor of x") and the least integer greater than or equal to x by $\lceil x \rceil$ (read "the ceiling of x"). For all real x,

$$x - 1 < \lfloor x \rfloor \leqslant x \leqslant \lceil x \rceil < x + 1\,. \tag{3.3}$$

For any integer n,

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n\,,$$

and for any real number $x \geqslant 0$ and integers $a, b > 0$,

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor\,, \tag{3.4}$$

$$\left\lceil \frac{\lfloor x/a \rfloor}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil\,, \tag{3.5}$$

$$\left\lfloor \frac{a}{b} \right\rfloor \leqslant \frac{a + (b-1)}{b}\,, \tag{3.6}$$

$$\left\lceil \frac{a}{b} \right\rceil \geqslant \frac{a - (b-1)}{b}\,. \tag{3.7}$$

The floor function $f(x) = \lfloor x \rfloor$ is monotonically increasing, as is the ceiling function $f(x) = \lceil x \rceil$.

3.8.3 Modular arithmetic

For any integer a and any positive integer n, the value $a \mod n$ is the remainder (or residue) of the quotient $\frac{a}{n}$

$$a \mod n = a - n \left\lfloor \frac{a}{n} \right\rfloor\,. \tag{3.8}$$

It follows that

$$0 \leqslant a \mod n < n. \tag{3.9}$$

Given a well-defined notion of the remainder of one integer when divided by another, it is convenient to provide special notation to indicate equality of remainders. If $(a \mod n) = (b \mod n)$, we write $a \equiv b \pmod{n}$ and say that a is *equivalent* to b, modulo n. In other words, $a \equiv b \pmod{n}$ if a and b have the same remainder when divided by n. Equivalently, $a \equiv b \pmod{n}$ if and only if n is a divisor of $b - a$. We write $a \not\equiv b \pmod{n}$ if a is not equivalent to b, modulo n.

3.8.4 Polynomials

Given a nonnegative integer d, a polynomial in n of degree d is a function $p(n)$ of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i.$$

where the constants $a_0, a_1, \ldots, a_d$ are the *coefficients* of the polynomial and $a_d \neq 0$. A polynomial is asymptotically positive if and only if $a_d > 0$. For an asymptotically positive polynomial $p(n)$ of degree d, we have $p(n) = \Theta(n^d)$. For any real constant $a \geqslant 0$, the function n^a is monotonically increasing, and for any real constant $a \leqslant 0$, the function n^a is monotonically decreasing. We say that a function $f(n)$ is *polynomially bounded* if $f(n) = O(n^k)$ for some constant k.

3.8.5 Exponentials

For all real $a > 0$, m, and n, we have the following identities:

$$\begin{aligned} a^0 &= 1 \\ a^1 &= a \\ a^{-1} &= \frac{1}{a} \\ (a^m)^n &= a^{mn} \\ (a^m)^n &= (a^n)^m \\ a^m a^n &= a^{m+n}. \end{aligned}$$

For all n and $a \geqslant 1$, the function a^n is monotonically increasing in n. When convenient, we shall assume $0^0 = 1$

We can relate the rates of growth of polynomials and exponentials by the following fact. For all real constants a and b such that $a > 1$,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0.$$

From which we can conclude that

$$n^b = o(a^n).$$

Thus, any exponential function with a base strictly greater than 1 grows faster than any polynomial function

Regarding e, recall

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots.$$

For all real x, we have

$$e^x \geqslant 1 + x.$$

With equality only when $x = 0$. When $|x| \leqslant 1$, we have the approximation

$$1 + x \leqslant e^x \leqslant 1 + x + x^2.$$

When $x \to 0$, the approximation of e^x by $1 + x$ is quite good

$$e^x = 1 + x + \Theta(x^2).$$

Further recall

$$\lim_{n\to\infty}\left(1 + \frac{x}{n}\right)^n = e^x.$$

3.8.6 Logarithms

We shall use the following notations:

$$\begin{aligned} \lg\ n &= \log_2(n) \\ \ln(n) &= \log_e(n) \\ \lg^x\ n &= (\lg\ n)^k \\ \operatorname{lglg}\ n &= \lg(\lg\ n). \end{aligned}$$

An important notational convention we shall adopt is that *logarithm functions will apply only to the next term in the formula*, so that $\lg n + k$ will mean $(\lg n) + k$ and not $\lg(n + k)$. If we hold $b > 1$ constant, then for $n > 0$, the function $\log_b n$ is strictly increasing.

For all real $a > 0$, $b > 0$, $c > 0$, and n,

$$\begin{aligned} a &= b^{\log_b a}, \\ \log_c(ab) &= \log_c a + \log_c b, \\ \log_b a^n &= n \log_b a, \\ \log_b a &= \frac{\log_c a}{\log_c b}, && (3.15) \\ \log_b(1/a) &= -\log_b a, \\ \log_b a &= \frac{1}{\log_a b}, \\ a^{\log_b c} &= c^{\log_b a}, && (3.16) \end{aligned}$$

where, in each equation above, logarithm bases are not 1.

There is a simple series expansion for $\ln(1 + x)$ when $|x| < 1$:

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots .$$

We also have the following inequalities for $x > -1$:

$$\frac{x}{1 + x} \leqslant \ln(1 + x) \leqslant x, \tag{3.17}$$

where equality holds only for $x = 0$.

We say that a function $f(n)$ is *polylogarithmically bounded* if $f(n) = O(\lg^k n)$ for some constant k. We can relate the growth of polynomials and polylogarithms by substituting $\lg n$ for n and 2^a for a in equation (3.10), yielding

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0.$$

From this limit, we can conclude that

$$\lg^b n = o(n^a)$$

for any constant $a > 0$. Thus, any positive polynomial function grows faster than any polylogarithmic function

3.8.7 Functional iteration

We use the notation $f^{(i)}(n)$ to denote the function $f(n)$ iteratively applied i times to an initial value of n. Formally, let $f(n)$ be a function over the reals. For non-negative integers i, we recursively define

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0, \\ f(f^{(i-1)}(n)) & \text{if } i > 0. \end{cases}$$

For example, if $f(n) = 2n$, then $f^{(i)}(n) = 2^i n$.

3.8.8 The iterated logarithm function

We use the notation $\lg^* n$ (read "log star of n") to denote the iterated logarithm, defined as follows. Let $\lg^{(i)} n$ be as defined above, with $f(n) = \lg n$. Because the logarithm of a nonpositive number is undefined, $\lg^{(i)} n$ is defined only if $\lg^{(i-1)} n > 0$. Be sure to distinguish $\lg^{(i)} n$ (the logarithm function applied i times in succession, starting with argument n) from $\lg^i n$ (the logarithm of n raised to the i-th power). Then we define the iterated logarithm function as

$$\lg^* n = \min\{i \geqslant 0 : \lg^{(i)} n \leqslant 1\}.$$

The iterated logarithm is a *very* slowly growing function:

$$\begin{aligned} \lg^* 2 &= 1, \\ \lg^* 4 &= 2, \\ \lg^* 16 &= 3, \\ \lg^* 65536 &= 4, \\ \lg^*(2^{65536}) &= 5. \end{aligned}$$

Divide and conquer

Recall that in divide-and-conquer, we solve a problem recursively, applying three steps at each level of the recursion

- **Divide**: the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer**: the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- **Combine**: the solutions to the subproblems into the solution for the original problem.

When the subproblems are large enough to solve recursively, we call that the ***recursive case***. Once the subproblems become small enough that we no longer recurse, we say that the recursion "bottoms out" and that we have gotten down to the ***base case***. Sometimes, in addition to subproblems that are smaller instances of the same problem, we have to solve subproblems that are not quite the same as the original problem. We consider solving such subproblems as part of the combine step

4.1 Recurrences

A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs. we described the worst-case running time $T(n)$ of the MERGE-SORT procedure by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if} n = 1 \\ 2T(n/2) + \Theta(n) & \text{if} n > 1 \end{cases}.$$

Whose solution we claimed to be $T(n) = \Theta(n \lg n)$

Recurrences can take many forms. For example, a recursive algorithm might divide subproblems into unequal sizes, such as a 2/3-to-1/3 split. If the divide and combine steps take linear time, such an algorithm would give rise to the recurrence

$$T(n) = T(2n/3) + T(n/3) + \Theta(n).$$

Subproblems are not necessarily constrained to being a constant fraction of the original problem size. For example, a recursive version of linear search (see Exercise 2.1-3) would create just one subproblem containing only one element fewer than the original problem. Each recursive call would take constant time plus the time for the recursive calls it makes, yielding the recurrence

$$T(n) = T(n-1) + \Theta(1).$$

This chapter offers three methods for solving recurrences—that is, for obtaining asymptotic "Θ" or "O" bounds on the solution:

- In the *substitution method*, we guess a bound and then use mathematical induction to prove our guess correct.
- The *recursion-tree method* converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.

- The *master method* provides bounds for recurrences of the form

$$T(n) = aT(n/b) + f(n), \tag{4.2}$$

where $a \geqslant 1$, $b > 1$, and $f(n)$ is a given function. Such recurrences arise frequently. A recurrence of the form in equation (4.2) characterizes a divide-and-conquer algorithm that creates a subproblems, each of which is $1/b$ the size of the original problem, and in which the divide and combine steps together take $f(n)$ time.

To use the master method, you will need to memorize three cases, but once you do that, you will easily be able to determine asymptotic bounds for many simple recurrences

Occasionally, we shall see recurrences that are not equalities but rather inequalities, such as $T(n) \leqslant 2T(n/2) + \Theta(n)$. Because such a recurrence states only an upper bound on $T(n)$, we will couch its solution using O-notation rather than Θ-notation. Similarly, if the inequality were reversed to $T(n) \geqslant 2T(n/2) + \Theta(n)$, then because the recurrence gives only a lower bound on $T(n)$, we would use Ω-notation in its solution.

4.2 Technicalities in recurrences

In practice, we neglect certain technical details when we state and solve recurrences. For example, if we call MERGE-SORT on n elements when n is odd, we end up with subproblems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Neither size is actually $n/2$, because $n/2$ is not an integer when n is odd. Technically, the recurrence describing the worst-case running time of MERGE-SORT is really

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1. \end{cases} \tag{4.3}$$

Boundary conditions represent another class of details that we typically ignore. Since the running time of an algorithm on a constant-sized input is a constant, the recurrences that arise from the running times of algorithms generally have $T(n) = \Theta(1)$ for sufficiently small n. Consequently, for convenience, we shall generally omit statements of the boundary conditions of recurrences and assume that $T(n)$ is constant for small n. For example, we normally state recurrence (4.1) as

$$T(n) = 2T(n/2) + \Theta(n), \tag{4.4}$$

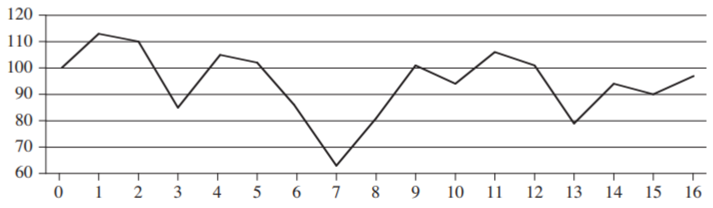
without explicitly giving values for small n. The reason is that although changing the value of $T(1)$ changes the exact solution to the recurrence, the solution typically doesn't change by more than a constant factor, and so the order of growth is unchanged.

When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions. We forge ahead without these details and later determine whether or not they matter. They usually do not, but you should know when they do. Experience helps, and so do some theorems stating that these details do not affect the asymptotic bounds of many recurrences characterizing divide-and-conquer algorithms

4.3 The maximum-subarray problem

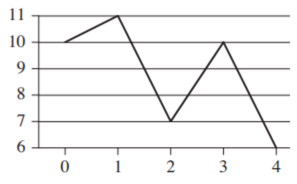
Suppose that you been offered the opportunity to invest in the Volatile Chemical Corporation. Like the chemicals the company produces, the stock price of the Volatile Chemical Corporation is rather volatile. You are allowed to buy one unit of stock only one time and then sell it at a later date, buying and selling after the close of trading for the day. To compensate for this restriction, you are allowed to learn what the price of the stock will be in the future. Your goal is to maximize your profit

You might think that you can always maximize profit by either buying at the lowest price or selling at the highest price. For example, in Figure 4.1, we would maximize profit by buying at the lowest price, after day 7. If this strategy always worked, then it would be easy to determine how to maximize profit: find the highest and lowest prices, and then work left from the highest price to find the lowest prior price, work right from the lowest price to find the highest later price, and take the pair with the greater difference. Figure 4.2 shows a simple counterexample, demonstrating that the maximum profit sometimes comes neither by buying at the lowest price nor by selling at the highest price.



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Figure 4.1 Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

Figure 4.2 An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of $3 per share would be earned by buying after day 2 and selling after day 3. The price of $7 after day 2 is not the lowest price overall, and the price of $10 after day 3 is not the highest price overall.

4.3.1 A brute-force solution

We can easily devise a brute-force solution to this problem: just try every possible pair of buy and sell dates in which the buy date precedes the sell date. A period of n days has $\binom{n}{2}$ such pairs of dates. Since $\binom{n}{2}$ is $\Theta(n^2)$, and the best we can hope for is to evaluate each pair of dates in constant time, this approach would take $\Omega(n^2)$ time. Can we do better?

4.3.2 A transformation

In order to design an algorithm with an $o(n^2)$ running time, we will look at the input in a slightly different way. We want to find a sequence of days over which the net change from the first day to the last is maximum. Instead of looking at the daily prices, let us instead consider the daily change in price, where the change on day i is the difference between the prices after day $i-1$ and after day i. The table in Figure 4.1 shows these daily changes in the bottom row. If we treat this row as an array A, shown in Figure 4.3, we now want to find the nonempty, contiguous subarray of A whose values have the largest sum. We call this contiguous subarray the *maximum subarray*. For example, in the array of Figure 4.3, the maximum subarray of $A[1 \ldots 16]$ is $A[8 \ldots 11]$, with the sum 43. Thus, you would want to buy the stock just before day 8 (that is, after day 7) and sell it after day 11, earning a profit of \$43 per share.

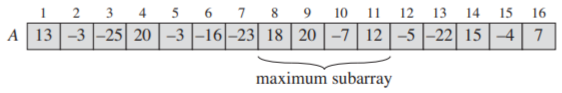


Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8\,..\,11]$, with sum 43, has the greatest sum of any contiguous subarray of array A.

At first glance, this transformation does not help. We still need to check $\binom{n-1}{2} = \Theta(n^2)$ subarrays for a period of n days

4.3.3 A solution using divide-and-conquer

Let's think about how we might solve the maximum-subarray problem using the divide-and-conquer technique. Suppose we want to find a maximum subarray of the subarray $A[low \ldots high]$. Divide-and-conquer suggests that we divide the subarray into two subarrays of as equal size as possible. That is, we find the midpoint, say *mid*, of the subarray, and consider the subarrays $A[low \ldots mid]$ and $A[mid+1 \ldots high]$. As Figure 4.4(a) shows, any contiguous subarray $A[i \ldots j]$ of $A[low \ldots high]$ must lie in exactly one of the following places:

- entirely in the subarray $A[low \ldots mid]$, so that $low \leqslant i \leqslant j \leqslant mid$,
- entirely in the subarray $A[mid+1 \ldots high]$, so that $mid < i \leqslant j \leqslant high$, or
- crossing the midpoint, so that $low \leqslant i \leqslant mid < j \leqslant high$.

Therefore, a maximum subarray of $A[low \ldots high]$ must lie in exactly one of these places. In fact, a maximum subarray of $A[low \ldots high]$ must have the greatest sum over all subarrays entirely in $A[low \ldots mid]$, entirely in $A[mid+1 \ldots high]$, or crossing the midpoint. We can find maximum subarrays of $A[low \ldots mid]$ and $A[mid+1 \ldots high]$ recursively, because these two subproblems are smaller instances of the problem of finding a maximum subarray. Thus, all that is left to do is find a maximum subarray that crosses the midpoint, and take a subarray with the largest sum of the three

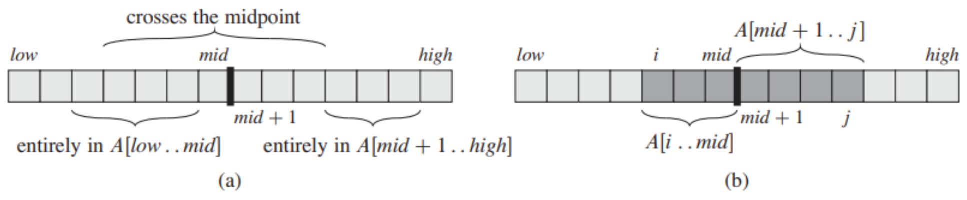


Figure 4.4 **(a)** Possible locations of subarrays of $A[low\,..\,high]$: entirely in $A[low\,..\,mid]$, entirely in $A[mid+1\,..\,high]$, or crossing the midpoint *mid*. **(b)** Any subarray of $A[low\,..\,high]$ crossing the midpoint comprises two subarrays $A[i\,..\,mid]$ and $A[mid+1\,..\,j]$, where $low \le i \le mid$ and $mid < j \le high$.

```
0   FIND-MAX-CROSSING-SUBARRAY($A$, low,mid,high)
1       left-sum = $-\infty$
2       sum = 0
3       for i = mid downto low
4           sum = sum + A[i]
5           if sum > left-sum
6               left-sum = sum
7               max-left = i
8       right-sum = $-\infty$
9       sum = 0
10      for j = mid+1 to high
11          sum = sum + A[j]
12          if sum > right-sum
13              right-sum = sum
14              max-right = j
15      return (max-left, max-right, left-sum + right-sum)
```

If the subarray $A[low \ldots high]$ contains n entries (so that $n = high - low + 1$), we claim that the call FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$ takes $\Theta(n)$ time. Since each iteration of each of the two **for** loops takes $\Theta(1)$ time, we just need to count up how many iterations there are altogether. The **for** loop of lines 3–7 makes $mid - low + 1$ iterations, and the **for** loop of lines 10–14 makes $high - mid$ iterations, and so the total number of iterations is

$$(mid - low + 1) + (high - mid) = high - low + 1 = n.$$

With a linear-time FIND-MAX-CROSSING-SUBARRAY procedure in hand, we can write pseudocode for a divide-and-conquer algorithm to solve the maximum-subarray problem:

```
0  FIND-MAXIMUM-SUBARRAY($A, low, high$)
1      if high == low
2          return ($low, high, A[low]$)
3      else mid = $\lfloor (low + high)/2 \rfloor$
4          (left-low,left-high, left-sum) =
5              FIND-MAXIMUM-SUBARRAY(A,low,mid)
6          (right-low,right-high, right-sum) =
7              FIND-MAXIMUM-SUBARRAY(A,mid+1,high)
8          (cross-low,cross-high, cross-sum) =
9              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
10         if left-sum $\geqslant$ right-sum and left-sum $\geqslant$ cross-sum
11             return (left-low, left-high, left-sum)
12         else if right-sum $\geqslant$ left-sum and right-sum $\geqslant$ cross-sum
13             return (right-low, right-high, right-sum)
14         else return (cross-low, cross-high, cross-sum)
```

The initial call FIND-MAXIMUM-SUBARRAY($A, 1, 1, A.length$) will find a maximum subarray of $A[1 \ldots n]$.

Similar to FIND-MAX-CROSSING-SUBARRAY, the recursive procedure FIND-MAXIMUM-SUBARRAY returns a tuple containing the indices that demarcate a maximum subarray, along with the sum of the values in a maximum subarray. Line 1 tests for the base case, where the subarray has just one element. A subarray with just one element has only one subarray—itself—and so line 2 returns a tuple with the starting and ending indices of just the one element, along with its value. Lines 3–11 handle the recursive case. Line 3 does the divide part, computing the index *mid* of the midpoint. Let's refer to the subarray $A[low \ldots mid]$ as the *left subarray* and to $A[mid + 1 \ldots high]$ as the *right subarray*. Because we know that the subarray $A[low \ldots high]$ contains at least two elements, each of the left and right subarrays must have at least one element. Lines 4 and 5 conquer by recursively finding maximum subarrays within the left and right subarrays, respectively. Lines 6–11 form the combine part. Line 6 finds a maximum subarray that crosses the midpoint. (Recall that because line 6 solves a subproblem that is not a smaller instance of the original problem, we consider it to be in the combine part.) Line 7 tests whether the left subarray contains a subarray with the maximum sum, and line 8 returns that maximum subarray. Otherwise, line 9 tests whether the right subarray contains a subarray with the maximum sum, and line 10 returns that maximum subarray. If neither the left nor right subarrays contain a subarray achieving the maximum sum, then a maximum subarray must cross the midpoint, and line 11 returns it.

4.3.4 Analyzing the divide-and-conquer algorithm

Next we set up a recurrence that describes the running time of the recursive FINDMAXIMUM-SUBARRAY procedure. we make the simplifying assumption that the original problem size is a power of 2, so that all subproblem sizes are integers.

We denote by $T(n)$ the running time of FIND-MAXIMUM-SUBARRAY on a subarray of n elements. For starters, line 1 takes constant time. The base case, when $n = 1$, is easy: line 2 takes constant time, and so

$$T(1) = \Theta(1).$$

The recursive case occurs when $n > 1$. Lines 1 and 3 take constant time. Each of the subproblems solved in lines 4 and 5 is on a subarray of $n/2$ elements (our assumption that the original problem size is a power of 2 ensures that $n/2$ is an integer), and so we spend $T(n/2)$ time solving each of them. Because we have to solve two subproblems—for the left subarray and for the right subarray—the contribution to the running time from lines 4 and 5 comes to $2T(n/2)$. As we have already seen, the call to FIND-MAX-CROSSING-SUBARRAY in line 6 takes $\Theta(n)$ time. Lines 7–11 take only $\Theta(1)$ time. For the recursive case, therefore, we have

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n). \end{aligned} \tag{4.6}$$

Combining equations (4.5) and (4.6) gives us a recurrence for the running time $T(n)$ of FIND-MAXIMUM-SUBARRAY:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases} \tag{4.7}$$

This recurrence is the same as recurrence (4.1) for merge sort. As we shall see from the master method in the future, this recurrence has the solution $T(n) = \Theta(n \lg n)$. You might also revisit the recursion tree in Figure 2.5 to understand why the solution should be $T(n) = \Theta(n \lg n)$.

4.4 Strassen's algorithm for matrix multiplication

4.4.1 The standard $\Theta(n^3)$ approach

If $A = (a_{ij})$ and $B = (b_{ij})$ are square $n \times n$ matrices, then in the product $C = A \cdot B$, we define the entry c_{ij}, for $i, j = 1, 2, \ldots, n$, by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}. \tag{4.8}$$

We must compute n^2 matrix entries, and each is the sum of n values. The following procedure takes $n \times n$ matrices A and B and multiplies them, returning their $n \times n$ product C. We assume that each matrix has an attribute *rows*, giving the number of rows in the matrix.

Using the definition of matrix multiplication, we have the simple algorithm

```
0  SQUARE-MATRIX-MULTIPLY(A,B)
1      n = A.rows
2      let $C$ be a new $n \times n$ matrix
3      for i = 1 to n
4          for j = 1 to n
5              $c_{ij}$ = 0
6              for k = 1 to n
7                  $c_{ij}$ = $c_{ij} + a_{ik} \cdot b_{kj}$
8      return $C$
```

Because each of the triply-nested for loops runs exactly n iterations, and each execution of line 7 takes constant time, the SQUARE-MATRIX-MULTIPLY procedure takes $\Theta(n^3)$ time

4.4.2 A simple divide-and-conquer algorithm

To keep things simple, when we use a divide-and-conquer algorithm to compute the matrix product $C = A \cdot B$, we assume that n is an exact power of 2 in each of the $n \times n$ matrices. We make this assumption because in each divide step, we will divide $n \times n$ matrices into four $n/2 \times n/2$ matrices, and by assuming that n is an exact power of 2, we are guaranteed that as long as $n \geqslant 2$, the dimension $n/2$ is an integer.

Suppose that we partition each of A, B, and C into four $n/2 \times n/2$ matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{4.10}$$

Equation (4.10) corresponds to the four equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \tag{4.11}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \tag{4.12}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \tag{4.13}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \tag{4.14}$$

Each of these four equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products. We can use these equations to create a straightforward, recursive, divide-and-conquer algorithm:

```
0  SQUARE-MATRIX-MULTIPLY-RECURSIVE(A,B)
1      n = A.rows
2      Let $C$ be a new $n \times n$ matrix
3      if n == 1
4          $c_{11} = a_{11} \cdot b_{11}$
5      else partition $A, B,$ and $C$ as in equations (4.9)
6          $C_{11}$ = SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{11}, B_{11}$)
7              +SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{12}, B_{21}$)
8          $C_{12}$ = SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{11}, B_{12}$)
9              +SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{12}, B_{22}$)
10         $C_{21}$ = SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{21}, B_{11}$)
11             +SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{22}, B_{21}$)
12         $C_{22}$ = SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{21}, B_{12}$)
13             +SQUARE-MATRIX-MULTIPLY-RECURSIVE($A_{22}, B_{22}$)
14     return $C$
```

This pseudocode glosses over one subtle but important implementation detail. How do we partition the matrices in line 5? If we were to create 12 new $n/2 \times n/2$ matrices, we would spend $\Theta(n^2)$ time copying entries. In fact, we can partition the matrices without copying entries. The trick is to use index calculations. We identify a submatrix by a range of row indices and a range of column indices of the original matrix. We end up representing a submatrix a little differently from how we represent the original matrix, which is the subtlety we are glossing over. The advantage is that, since we can specify submatrices by index calculations, executing line 5 takes only $\Theta(1)$ time (although we shall see that it makes no difference asymptotically to the overall running time whether we copy or partition in place).

Now, we derive a recurrence to characterize the running time of SQUARE-MATRIX-MULTIPLY-RECURSIVE. Let $T(n)$ be the time to multiply two $n \times n$ matrices using this procedure. In the base case, when $n = 1$, we perform just the one scalar multiplication in line 4, and so

$$T(1) = \Theta(1). \tag{4.15}$$

The recursive case occurs when $n > 1$. As discussed, partitioning the matrices in line 5 takes $\Theta(1)$ time, using index calculations. In lines 6–9, we recursively call SQUARE-MATRIX-MULTIPLY-RECURSIVE a total of eight times. Because each recursive call multiplies two $n/2 \times n/2$ matrices, thereby contributing $T(n/2)$ to the overall running time, the time taken by all eight recursive calls is $8T(n/2)$. We also must account for the four matrix additions in lines 6–9. Each of these matrices contains $n^2/4$ entries, and so each of the four matrix additions takes $\Theta(n^2)$ time. Since the number of matrix additions is a constant, the total time spent adding ma-

trices in lines 6–9 is $\Theta(n^2)$. (Again, we use index calculations to place the results of the matrix additions into the correct positions of matrix C, with an overhead of $\Theta(1)$ time per entry.) The total time for the recursive case, therefore, is the sum of the partitioning time, the time for all the recursive calls, and the time to add the matrices resulting from the recursive calls:

$$\begin{aligned} T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\ &= 8T(n/2) + \Theta(n^2). \end{aligned} \tag{4.16}$$

Notice that if we implemented partitioning by copying matrices, which would cost $\Theta(n^2)$ time, the recurrence would not change, and hence the overall running time would increase by only a constant factor.

Combining equations (4.15) and (4.16) gives us the recurrence for the running time of SQUARE-MATRIX-MULTIPLY-RECURSIVE:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1. \end{cases} \tag{4.17}$$

As we shall see from the master method in Section 4.5, recurrence (4.17) has the solution $T(n) = \Theta(n^3)$. Thus, this simple divide-and-conquer approach is no faster than the straightforward SQUARE-MATRIX-MULTIPLY procedure.

4.4.3 Strassen's method

The key to Strassen's method is to make the recursion tree slightly less bushy. That is, instead of performing eight recursive multiplications of $n/2 \times n/2$ matrices, it performs only seven. The cost of eliminating one matrix multiplication will be several new additions of $n/2 \times n/2$ matrices, but still only a constant number of additions. As before, the constant number of matrix additions will be subsumed by Θ-notation when we set up the recurrence equation to characterize the running time.

Strassen's method is not at all obvious.

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.

2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.

3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. Each matrix P_i is $n/2 \times n/2$.

4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

We shall see the details of steps 2–4 in a moment, but we already have enough information to set up a recurrence for the running time of Strassen's method. Let us assume that once the matrix size n gets down to 1, we perform a simple scalar multiplication, just as in line 4 of SQUARE-MATRIX-MULTIPLY-RECURSIVE. When $n > 1$, steps 1, 2, and 4 take a total of $\Theta(n^2)$ time, and step 3 requires us to perform seven multiplications of $n/2 \times n/2$ matrices. Hence, we obtain the following recurrence for the running time $T(n)$ of Strassen's algorithm:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 7T(n/2) + \Theta(n^2), & \text{if } n > 1. \end{cases}$$

We have traded off one matrix multiplication for a constant number of matrix additions. Once we understand recurrences and their solutions, we shall see that this tradeoff actually leads to a lower asymptotic running time. By the master method in Section 4.5, recurrence (4.18) has the solution $T(n) = \Theta(n^{\log_2 7})$.

We now proceed to describe the details. In step 2, we create the following 10 matrices:

$$\begin{aligned} S_1 &= B_{12} - B_{22}, \\ S_2 &= A_{11} + A_{12}, \\ S_3 &= A_{21} + A_{22}, \\ S_4 &= B_{21} - B_{11}, \\ S_5 &= A_{11} + A_{22}, \\ S_6 &= B_{11} + B_{22}, \\ S_7 &= A_{12} - A_{22}, \\ S_8 &= B_{21} + B_{22}, \\ S_9 &= A_{11} - A_{21}, \\ S_{10} &= B_{11} + B_{12}. \end{aligned}$$

Since we must add or subtract $n/2 \times n/2$ matrices 10 times, this step does indeed take $\Theta(n^2)$ time.

In step 3, we recursively multiply $n/2 \times n/2$ matrices seven times to compute the following $n/2 \times n/2$ matrices, each of which is the sum or difference of products of A and B submatrices:

$$\begin{aligned} P_1 &= A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22}, \\ P_2 &= S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22}, \\ P_3 &= S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11}, \\ P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11}, \\ P_5 &= S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}, \\ P_6 &= S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}, \\ P_7 &= S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}. \end{aligned}$$

Expanding out the right-hand side, with the expansion of each P_i on its own line and vertically aligning terms that cancel out, we see that C_{11} equals

$$\begin{aligned}
&A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
&- A_{11} \cdot B_{11} - A_{22} \cdot B_{21} \\
&- A_{11} \cdot B_{22} - A_{12} \cdot B_{22} \\
&- A_{22} \cdot B_{22} - A_{22} \cdot B_{21} + A_{12} \cdot B_{22} + A_{12} \cdot B_{21},
\end{aligned}$$

which simplifies to

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21},$$

which corresponds to equation (4.11).

Similarly, we set

$$C_{12} = P_1 + P_2,$$

and so C_{12} equals

$$\begin{aligned}
&A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
&+ A_{11} \cdot B_{22} + A_{12} \cdot B_{22},
\end{aligned}$$

which simplifies to

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22},$$

corresponding to equation (4.12).

Setting

$$C_{21} = P_3 + P_4,$$

makes C_{21} equal

$$\begin{aligned}
&A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
&- A_{22} \cdot B_{11} + A_{22} \cdot B_{21},
\end{aligned}$$

which simplifies to

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21},$$

corresponding to equation (4.13).

$$C_{22} = P_5 + P_1 - P_3 - P_7,$$

so that C_{22} equals

$$\begin{aligned}
&A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
&- A_{11} \cdot B_{12} + A_{11} \cdot B_{11} \\
&- A_{21} \cdot B_{11} - A_{21} \cdot B_{12} \\
&- A_{11} \cdot B_{12} + A_{21} \cdot B_{11} + A_{21} \cdot B_{12},
\end{aligned}$$

which simplifies to

$$C_{22} = A_{22} \cdot B_{22} + A_{21} \cdot B_{12}.$$

which corresponds to equation (4.14). Altogether, we add or subtract $n/2 \times n/2$ matrices eight times in step 4, and so this step indeed takes $\Theta(n^2)$ time.

Thus, we see that Strassen's algorithm, comprising steps 1–4, produces the correct matrix product and that recurrence (4.18) characterizes its running time. Since we shall see in Section 4.5 that this recurrence has the solution $T(n) = \Theta(n^{\log_2 7})$, Strassen's method is asymptotically faster than the straightforward SQUARE-MATRIX-MULTIPLY procedure. The notes at the end of this chapter discuss some further details.

4.5 The substitution method for solving recurrences

Now that we have seen how recurrences characterize the running times of divideand-conquer algorithms, we will learn how to solve recurrences. We start in this section with the "substitution" method.

The substitution method for solving recurrences comprises two steps:

1. Guess the form of the solution
2. Use mathematical induction to find the constants and show that the solution works.

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values; hence the name "substitution method." This method is powerful, but we must be able to guess the form of the answer in order to apply it.

We can use the substitution method to establish either upper or lower bounds on a recurrence. As an example, let us determine an upper bound on the recurrence

$$T(n) = 2T\left(\lfloor n/2 \rfloor\right) + n.$$

We guess that the solution is $T(n) = O(n \lg n)$. The substitution method requires us to prove that

$$T(n) \leqslant cn \lg n$$

for an appropriate choice of the constant $c > 0$. We start by assuming that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding

$$T(\lfloor n/2 \rfloor) \leqslant c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor).$$

Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leqslant 2\big(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)\big) + n, \\ &\leqslant cn \lg(n/2) + n, \\ &= cn \lg n - cn \lg 2 + n, \\ &= cn \lg n - cn + n, \\ &\leqslant cn \lg n.. \end{aligned}$$

Where the last step holds as long as $c \geqslant 1$.

Mathematical induction now requires us to show that our solution holds for the boundary conditions. Typically, we do so by showing that the boundary conditions are suitable as base cases for the inductive proof. For the recurrence (4.19), we must show that we can choose the constant c large enough so that the bound

$$T(n) \leqslant cn \lg n$$

works for the boundary conditions as well. This requirement can sometimes lead to problems. Let us assume, for the sake of argument, that $T(1) = 1$ is the sole boundary condition of the recurrence. Then for $n = 1$, the bound

$$T(n) \leqslant cn \lg n$$

yields $T(1) \leqslant c \cdot 1 \cdot \lg 1 = 0$, which is at odds with $T(1) = 1$. Consequently, the base case of our inductive proof fails to hold.

We can overcome this obstacle in proving an inductive hypothesis for a specific boundary condition with only a little more effort. In the recurrence (4.19), for example, we take advantage of asymptotic notation requiring us only to prove

$$T(n) \leqslant cn \lg n \quad \text{for} \quad n \geqslant n_0,$$

where n_0 is a constant that we get to choose. We keep the troublesome boundary condition $T(1) = 1$, but remove it from consideration in the inductive proof. We do so by first observing that for $n > 3$, the recurrence does not depend directly on $T(1)$. Thus, we can replace $T(1)$ by $T(2)$ and $T(3)$ as the base cases in the inductive proof, letting $n_0 = 2$. Note that we make a distinction between the base case of the recurrence ($n = 1$) and the base cases of the inductive proof ($n = 2$ and $n = 3$). With $T(1) = 1$, we derive from the recurrence that $T(2) = 4$ and $T(3) = 5$. Now we can complete the inductive proof that

$$T(n) \leqslant cn \lg n$$

for some constant $c \geqslant 1$ by choosing c large enough so that

$$T(2) \leqslant c \cdot 2 \lg 2 \quad \text{and} \quad T(3) \leqslant c \cdot 3 \lg 3.$$

As it turns out, any choice of $c \geqslant 2$ suffices for the base cases of $n = 2$ and $n = 3$ to hold. For most of the recurrences we shall examine, it is straightforward to extend boundary conditions to make the inductive assumption work for small n, and we shall not always explicitly work out the details.

4.5.1 Iterative approach

Consider the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + 2 \quad T(1) = 1.$$

If $T(n) = 2T\left(\frac{n}{2}\right) + n$, then

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}.$$

Further,

$$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}.$$

Thus,

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \\
&= 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n \\
&= 2^2T\left(\frac{n}{2^2}\right) + n + n \\
&= 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + n + n \\
&= 2^3T\left(\frac{n}{2^3}\right) + n + n + n.
\end{aligned}$$

Thus, we have a general iterative form

$$2^iT\left(\frac{n}{2^i}\right) + in.$$

Since we have a base case $T(1) = 1$, we require

$$\begin{aligned}
&\frac{n}{2^i} = 1 \\
\implies &i = \lg(n).
\end{aligned}$$

Thus, we have

$$\begin{aligned}&2^{\lg n}T(1) + \lg(n)n\\&= n^{\lg 2}(1) + \lg(n)n\\&= n\lg + n.\end{aligned}$$

Which is precisely $\Theta(n \lg n)$

4.5.2 Making a good guess

Unfortunately, there is no general way to guess the correct solutions to recurrences. Guessing a solution takes experience and, occasionally, creativity.

If a recurrence is similar to one you have seen before, then guessing a similar solution is reasonable. As an example, consider the recurrence

$$T(n) = 2T\left(\lfloor n/2 \rfloor + 17\right) + n.$$

which looks difficult because of the added “17” in the argument to T on the righthand side. Intuitively, however, this additional term cannot substantially affect the

The solution to the recurrence. When n is large, the difference between $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 17$ is not that large: both cut n nearly evenly in half. Consequently, we make the guess that $T(n) = O(n \lg n)$, which you can verify as correct by using the substitution method

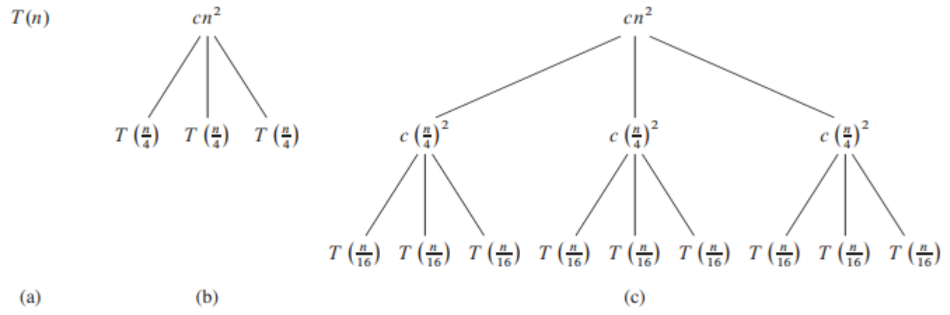
4.6 The recursion-tree method for solving recurrences

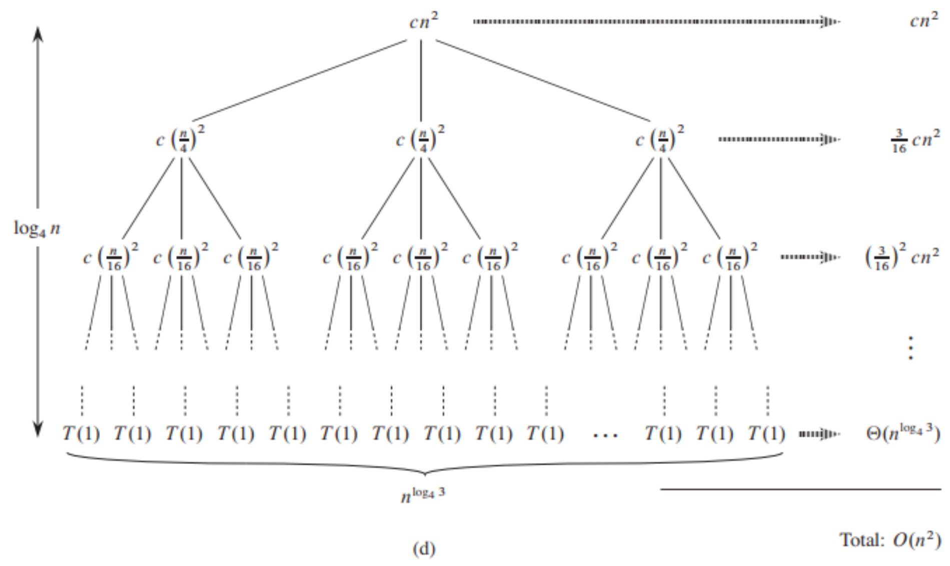
Although you can use the substitution method to provide a succinct proof that a solution to a recurrence is correct, you might have trouble coming up with a good guess. Drawing out a recursion tree, as we did in our analysis of the merge sort recurrence, serves as a straightforward way to devise a good guess. In a recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.

per-level costs to determine the total cost of all levels of the recursion. A recursion tree is best used to generate a good guess, which you can then verify by the substitution method. When using a recursion tree to generate a good guess, you can often tolerate a small amount of “sloppiness,” since you will be verifying your guess later on. If you are very careful when drawing out a recursion tree and summing the costs, however, you can use a recursion tree as a direct proof of a solution to a recurrence.

For example, let us see how a recursion tree would provide a good guess for the recurrence $T(n) = 3T\left(\lfloor n/4 \rfloor\right) + \Theta(n^2)$. We start by focusing on finding an upper bound for the solution. Because we know that floors and ceilings usually do not matter when solving recurrences (here’s an example of sloppiness that we can tolerate), we create a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. For convenience, we assume that n is an exact power of 4 (another example of tolerable sloppiness) so that all subproblem sizes are integers.

Part (a) of the figure shows $T(n)$, which we expand in part (b) into an equivalent tree representing the recurrence The cn^2 term at the root represents the cost at the top level of recursion, and the three subtrees of the root represent the costs incurred by the subproblems of size $n/4$. Part (c) shows this process carried one step further by expanding each node with cost $T(n/4)$ from part (b). The cost for each of the three children of the root is $c(n/4)^2$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence.


$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
(a)
(b)
(c)



Because subproblem sizes decrease by a factor of 4 each time we go down one level, we eventually must reach a boundary condition. How far from the root do we reach one? The subproblem size for a node at depth i is $n/4^i$. Thus, the subproblem size hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$. Thus, the tree has $\log_4 n + 1$ levels (at depths $0, 1, 2, \ldots, \log_4 n$).

Next we determine the cost at each level of the tree. Each level has three times more nodes than the level above, and so the number of nodes at depth i is 3^i. Because subproblem sizes reduce by a factor of 4 for each level we go down from the root, each node at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, has a cost of $c(n/4^i)^2$. Multiplying, we see that the total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, is:

$$3^i c(n/4^i)^2 = \left(\frac{3}{16}\right)^i cn^2.$$

The bottom level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each contributing cost $T(1)$, for a total cost of:

$$n^{\log_4 3}T(1),$$

which is $\Theta(n^{\log_4 3})$, since we assume that $T(1)$ is a constant.

Now we add up the costs over all levels to determine the cost for the entire tree:

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}).$$

The sum can be written as:

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}).$$

This geometric series simplifies to:

$$T(n) = \frac{\left(\frac{3}{16}\right)^{\log_4 n} - 1}{\frac{3}{16} - 1} cn^2 + \Theta(n^{\log_4 3}).$$

Using equation (A.5), this becomes:

$$T(n) = cn^2 + \Theta(n^{\log_4 3}).$$

This last formula looks somewhat messy until we realize that we can again take advantage of small amounts of sloppiness and use an infinite decreasing geometric series as an upper bound. Backing up one step and applying equation (A.6), we have:

$$T(n) = cn^2 + O(n^{\log_4 3}).$$

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}), \\ &= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}), \\ &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}), \\ &= O(n^2). \end{aligned}$$

Thus, we have derived a guess of $T(n) = O(n^2)$ for our original recurrence:

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2).$$

In this example, the coefficients of cn^2 form a decreasing geometric series, and by equation (A.6), the sum of these coefficients is bounded from above by the constant $\frac{16}{13}$. Since the root's contribution to the total cost is cn^2, the root contributes a constant fraction of the total cost. In other words, the cost of the root dominates the total cost of the tree.

In fact, if $O(n^2)$ is indeed an upper bound for the recurrence (as we shall verify in a moment), then it must be a tight bound. Why? The first recursive call contributes a cost of $\Theta(n^2)$, and so $\Omega(n^2)$ must be a lower bound for the recurrence.

A.5:

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}.$$

A.6: For an infinite geometric sum with $|x| < 1$, we have the infinite decreasing geometric series

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}.$$

Now we can use the substitution method to verify that our guess was correct, that is, $T(n) = O(n^2)$ is an upper bound for the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. We want to show that $T(n) \leqslant dn^2$ for some constant $d > 0$. Using the same constant $c > 0$ as before, we have:

$$\begin{aligned} T(n) &\leqslant 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leqslant 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\leqslant 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\leqslant dn^2, . \end{aligned}$$

where the last step holds as long as $d > \frac{16}{13}c$.

In another, more intricate, example, Figure 4.6 shows the recursion tree for

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n).$$

(Again, we omit floor and ceiling functions for simplicity.) As before, we let c represent the constant factor in the $O(n)$ term. When we add the values across the levels of the recursion tree shown in the figure, we get a value of cn for every level.

The longest simple path from the root to a leaf is

$$n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1.$$

Since $(2/3)^k n = 1$ when $k = \log_{3/2} n$, the height of the tree is $\log_{3/2} n$.

Intuitively, we expect the solution to the recurrence to be at most the number of levels times the cost of each level, or $O(cn \log_{3/2} n) = O(n \lg n)$. Figure 4.6 shows only the top levels of the recursion tree, however, and not every level in the tree contributes a cost of cn. Consider the cost of the leaves. If this recursion tree were a complete binary tree of height $\log_{3/2} n$, there would be $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ leaves. Since the cost of each leaf is a constant, the total cost of all leaves would then be

$$\Theta(n^{\log_{3/2} 2}),$$

which, since $\log_{3/2} 2$ is a constant strictly greater than 1, is $\omega(n \lg n)$. This recursion tree is not a complete binary tree, however, as it has fewer than $n^{\log_{3/2} 2}$ leaves. Moreover, as we go down from the root, more and more internal nodes are absent. Consequently, levels toward the bottom of the recursion tree contribute less than cn to the total cost.

We could work out an accurate accounting of all costs, but remember that we are just trying to come up with a guess to use in the substitution method. Let us tolerate the sloppiness and attempt to show that a guess of $O(n \lg n)$ for the upper bound is correct.

$$\begin{aligned}
T(n) &\leqslant T(n/3) + T(2n/3) + cn \\
&\leqslant d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (d(n/3)\lg n - d(n/3)\lg 3) \\
&+ (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d\left((n/3)\lg 3 + (2n/3)\lg(3/2)\right) + cn \\
&= dn\lg n - d\left((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2\right) + cn \\
&= dn\lg n - dn\left(\lg 3 - 2/3\right) + cn \\
&\leqslant dn\lg n, .
\end{aligned}$$

as long as $d \geqslant c/(\lg 3 - (2/3))$. Thus, we did not need to perform a more accurate accounting of costs in the recursion tree.

4.7 The master method for solving recurrences

The master method provides a "cookbook" method for solving recurrences of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

Where $a \geqslant 1$, and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. To use the master method, you will need to memorize three cases, but then you will be able to solve many recurrences quite easily, often without pencil and paper.

The recurrence (4.20) describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b, where a and b are positive constants. The a subproblems are solved recursively, each in time $T(n/b)$. The function $f(n)$ encompasses the cost of dividing the problem and combining the results of the subproblems. For example, the recurrence arising from Strassen's algorithm has $a = 7$, $b = 2$, and $f(n) = \Theta(n^2)$.

As a matter of technical correctness, the recurrence is not actually well defined, because n/b might not be an integer. Replacing each of the a terms $T(n/b)$ with either $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ will not affect the asymptotic behavior of the recurrence, however. (We will prove this assertion in the next section.) We normally find it convenient, therefore, to omit the floor and ceiling functions when writing divide-and-conquer recurrences of this form.

4.7.1 The master theorem

Theorem. Let $a \geqslant 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $af(n/b) \leqslant cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Before applying the master theorem to some examples, let's spend a moment trying to understand what it says. In each of the three cases, we compare the function $f(n)$ with the function $n^{\log_b a}$. Intuitively, the larger of the two functions determines the solution to the recurrence. If, as in case 1, the function $n^{\log_b a}$ is the larger, then the solution is $T(n) = \Theta(n^{\log_b a})$. If, as in case 3, the function $f(n)$ is the larger, then the solution is $T(n) = \Theta(f(n))$. If, as in case 2, the two functions are the same size, we multiply by a logarithmic factor, and the solution is

$$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n).$$

Beyond this intuition, you need to be aware of some technicalities. In the first case, not only must $f(n)$ be smaller than $n^{\log_b a}$, it must be *polynomially smaller*. That is, $f(n)$ must be asymptotically smaller than $n^{\log_b a}$ by a factor of n^ϵ for some constant $\epsilon > 0$. In the third case, not only must $f(n)$ be larger than $n^{\log_b a}$, it also must be polynomially larger and in addition satisfy the "regularity" condition that

$$af(n/b) \leqslant cf(n).$$

This condition is satisfied by most of the polynomially bounded functions that we shall encounter.

Note that the three cases do not cover all the possibilities for $f(n)$. There is a gap between cases 1 and 2 when $f(n)$ is smaller than $n^{\log_b a}$ but not polynomially smaller. Similarly, there is a gap between cases 2 and 3 when $f(n)$ is larger than $n^{\log_b a}$ but not polynomially larger. If the function $f(n)$ falls into one of these gaps, or if the regularity condition in case 3 fails to hold, you cannot use the master method to solve the recurrence.

4.7.2 Using the master method

To use the master method, we simply determine which case (if any) of the master theorem applies and write down the answer.

As a first example, consider

$$T(n) = 9T\left(\frac{n}{3}\right) + n.$$