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.

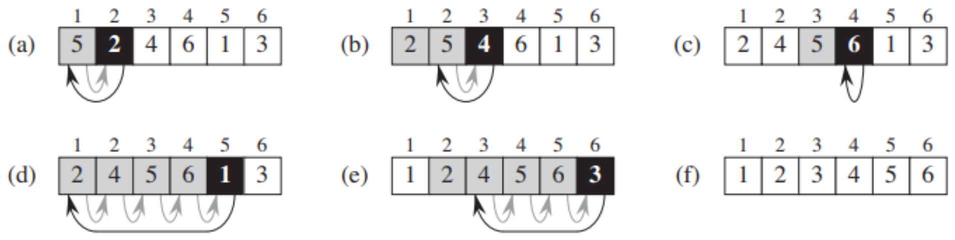
Prelude

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

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

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.

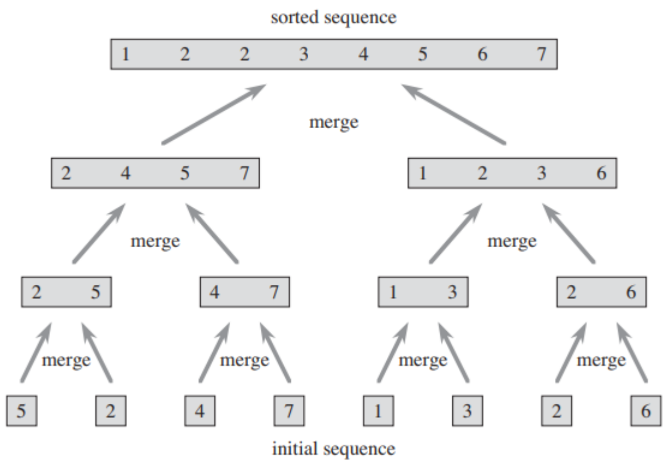
3. **Termination:** At termination, $k = r + 1$. By the loop invariant, the subarray $A[p \ldots k-1]$, which is $A[p \ldots r]$, contains the $k - p = r - p + 1$ smallest elements of $L[1 \ldots n_1 + 1]$ and $R[1 \ldots n_2 + 1]$, in sorted order. The arrays L and R together contain $n_1 + n_2 + 2 = r - p + 3$ 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 $A[p \ldots r]$. 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 $A[p \ldots r]$ into two subarrays: $A[p \ldots q]$, containing $\lfloor n/2 \rfloor$ elements, and $A[q+1 \ldots r]$, 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 $A = \langle A[1], A[2], \ldots, A[n] \rangle$, 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:

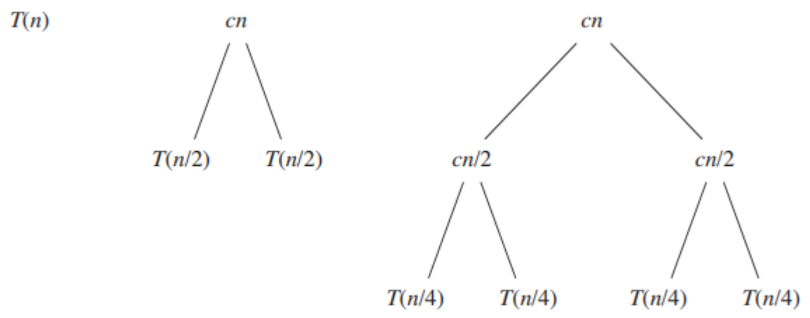
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases} \tag{2.1}$$

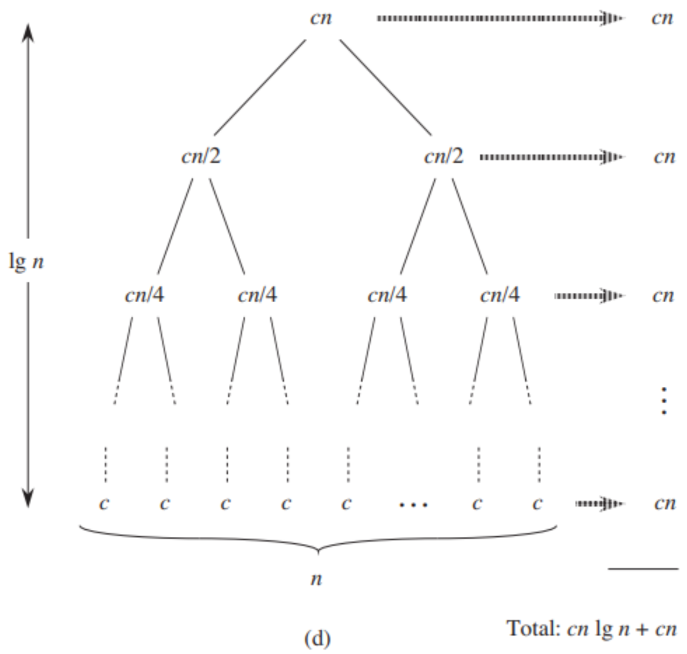
In the future, we shall see the "master theorem," which we can use to show that $T(n)$ is $\Theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$. Because the logarithm function grows more slowly than any linear function, for large enough inputs, merge sort, with its $\Theta(n \lg n)$ running time, outperforms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

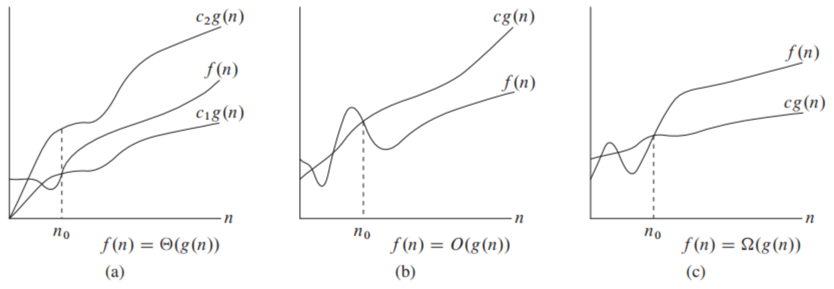
The functions to which we apply asymptotic notation will usually characterize the running times of algorithms. But asymptotic notation can apply to functions that characterize some other aspect of algorithms (the amount of space they use, for example), or even to functions that have nothing whatsoever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

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

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

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