

Chapter 1

Execution Time Measurement

In comparing two different implementations of the same ADT, an important question to address is their relative execution times. It might appear, at first, that describing the execution time of a computer algorithm should be easy. One merely runs the algorithm with a stopwatch in hand (or any number of technological variations of a similar theme) and records the results. And indeed, such a value, called a *benchmark*, is occasionally useful. We will examine benchmarks later in this chapter.

Most of the time, however, a benchmark is not a good characterization of a computer algorithm. A benchmark only describes the actual performance of a program on a specific machine on a given day. Different processors can have dramatically different performances (computer manufacturers go to great lengths to remind users of this fact in hopes of enticing them to purchase faster and more costly machines). Even working on a single machine, there may be a selection of many alternative compilers for the same programming language. These compilers will produce slightly different machine code instructions for the same source language, and thus have varying execution times. Finally, execution timings for multiprocessing computers (computers that can perform many tasks simultaneously) will be altered by the other tasks being performed at the same time.

In this chapter we introduce the concept of *algorithmic analysis* (or *analysis of algorithms*), a technique used to characterize the execution behavior of algorithms in a manner independent of a particular platform, compiler, or language. Many of the data structures described in this book provide similar functionality, but differ in their execution times for various operations. Thus, selecting the appropriate type of container for any particular problem involves not only an understanding of a container's capabilities, but also an understanding of how execution times are determined and contrasted.

1.1 Algorithmic Analysis and Big-Oh Notation

Rather than measuring the execution time of computer programs, we will henceforth speak of measuring computer *algorithms*. An algorithm will generally be embodied in one or more methods. Thus, our measurements will be based on an examination of the execution time for the invocation of a single method. Of course, making individual methods faster can only help to improve the execution speed of entire programs.

We would like a technique to abstract away small variations, and describe the performance of algorithms in a more idealized, processor-independent fashion. It is clear that alternative approaches for the same problem can have dramatically different execution times. Ask yourself, for example, why dictionaries list words in alphabetical order. The reason, we know, is that doing so permits the reader to discover a word by repeatedly moving back and forth over increasingly smaller sections of the book (a process we will later label *binary search*). If dictionaries were not ordered, then to discover whether a particular word appeared in the collection the reader would have to examine the list in sequence from one end to the other, comparing the test word against each entry in turn; obviously this process would be considerably slower than finding a word in a normal dictionary.

The technique we use to capture this more-or-less intuitive notion of complexity is an idea called *algorithmic analysis*. In algorithmic analysis (sometimes termed *asymptotic analysis*) we gain power by, ironically, losing precision. By systematically ignoring constant amounts, and instead concentrating on an abstract characterization of the *size* of a problem, we can make more intelligent comparisons.

Determining whether a word appears in an unorganized list of n alternatives requires, for example, comparing each word against the target, and therefore uses approximately n comparisons. We assume each of these comparisons can be performed in a constant amount of time (the particular constant is unimportant, being greater or smaller depending upon the machine being used). In total, the running time of the process is therefore $c \times n$ for some unknown constant c . On the other hand, if the list is ordered and we can easily move to any position in the collection we will see (later in this chapter) that the search can be performed using only $\log n$ comparisons. Thus the real running time is some function $d \times \log n$ for some unknown value d .

Because the actual constants involved are unknown and, largely, irrelevant, we can eliminate them from our description. We use a notation, called “big-Oh” notation, which captures this intuitive idea. We say that searching an unorganized list requires $O(n)$ steps (read “big-Oh of n steps”), whereas searching a sorted and randomly accessible dictionary can be performed in $O(\log n)$ steps (read “big-Oh of $\log n$ steps”). We call the big-Oh characterization the *asymptotic execution time* for the operation.

The formal definition of big-Oh states that if $f(n)$ is a function that represents the *actual* execution time of an algorithm, then the algorithm is $O(g(n))$ if, for all values n larger than some fixed constant n_0 , there exists some constant c , such that $f(n)$ is always bounded by (smaller than or equal to) the quantity $c \times g(n)$. Although this formal definition may be of critical importance to theoreticians, an intuitive feeling for algorithmic execution is of

more practical importance, and thus the remainder of the first section of this chapter will be devoted to providing this understanding. (For those readers who insist on a more formal approach, several of the exercises at the end of the chapter are designed to place the concept of algorithmic analysis on a firm foundation.)

1.2 Execution Time of Programming Constructs

In the following sections we will discuss the most common programming constructs and illustrate the analysis of the running times using several algorithms.

1.2.1 Constant Time

The Java language provides a number of primitive data types, such as integers and floating-point numbers. We begin by assuming that operations on such values (addition, subtraction and the like) can be performed within a constant amount of time. Assignment also requires only constant time. A fixed sequence of constant time operations, such as a sequence of assignments, still requires only constant time. The following is from a playing card abstraction, a class we will use in many of our examples:

```
public class Card {
    Card (int is, int ir) { // initialize a new Card
        suit = is; // suit
        rank = ir; // rank
    }

    public final int suit; // suit
    public final int rank; // rank

    public static final int diamond = 1;
    public static final int club = 2;
    public static final int spade = 3;
    public static final int heart = 4;
    ...
}
```

The maximum time it takes to perform a conditional if statement is the sum of the times it takes to perform a test, and the maximum time required by either alternative. If all three can be performed in constant time, then the time to execute the if statement can still be bounded by some constant value:

```
public class Card {
    ...
    public Color color () {
        if ((suit == diamond) || (suit == heart))
```

```

        return Color.red;
    else
        return Color.black;
    }
}

```

Even when procedures are quite lengthy, if they do not include loops or procedure calls, then the total execution time must remain constant.

1.2.2 Simple Loops

A loop that performs a constant number of iterations is still considered to be executing in constant time. An example might be the following, which creates a deck of playing cards, and places them into an array:

```

public class CardDeck {
    CardDeck () {
        int topCard = 0;
        for (int i = 1; i <= 13; i++) {
            cards[topCard++] = new Card(Card.diamond, i);
            cards[topCard++] = new Card(Card.club, i);
            cards[topCard++] = new Card(Card.spade, i);
            cards[topCard++] = new Card(Card.heart, i);
        }
    }

    private Card [ ] cards = new Card[52];
    ...
}

```

Such loops, however, are relatively rare in comparison to loops in general. More commonly, the number of iterations a loop will perform is determined in some fashion by the input values. To describe the running time of a loop we need to characterize how many iterations the loop will perform, and then multiply this value by the running time of the body of the loop. The simplest case occurs when the limits of the loop are fixed by the input value, and the body of the loop uses constant time. An example in which this occurs is a procedure that takes as arguments a array of values and returns the smallest value in the collection:

```

public double minimum (double [ ] values) {
    // pre: values has at least one element
    // post: returns smallest value in collection
    int n = values.length;
    double minValue = values[0];

    for (int i = 1; i < n; i++) {

```

The Array in Java

The *array* is the only data structure provided by the Java language that is designed specifically to hold a collection of values. (Other data structures are provided by the standard Java *library*, but they are not part of the basic Java language).

While many computer languages have an array data type, the array in Java is slightly more general than most. For example, when declaring a variable as an array the programmer need not specify an upper bound. This can be seen, for example, in the declaration of command line arguments that must be present in the main procedure in any program:

```
public class AnApplication {
    public static void main (String [ ] args) {
        ...
    }
}
```

The square brackets signify that `args` is an array of `String` values, yet do not specify how many elements the array will hold. Instead, the size of an array is established when the value of the variable is first created. Here, too, Java is different than most other computer languages. In Java creating an array is a two step process. First, the variable that will hold the array must be declared. Second, the space for the variable must be allocated using the `new` operator. This can be seen in the `CardDeck` example:

```
Card [ ] cards = new Card[52]; // create an array with 52 elements
```

Values in an array are accessed using the subscript operator. The index values used with this operator must be greater than equal to zero, and less than the number of elements in the array. Again, Java is unlike some other languages (such as C++) in that index bounds *are* checked, and an exception will be raised if they are not valid.

```

        // if current value is less than minimum so far
        if (values[i] < minValue)
            minValue = values[i]; // then save it
    }
    return minValue;
}

```

Here the loop will clearly execute n times. Because at each iteration we are performing at most one comparison and one assignment, the execution time for the body of the loop is constant. The total running time for the procedure is therefore $O(n)$. (Note that the n in $O(n)$ need not correspond to any variable in the program, it is just a convenient way to indicating the execution time is proportional to the size of the input. Even if we had used a different variable in the program we would still say the execution time was $O(n)$.)

Loops need not always have a simple terminating condition, and therefore a careful analysis of the algorithm may be necessary in order to characterize the number of iterations a loop will perform. A good example of this is a procedure to determine if an integer value represents a prime number. To do this we need only test integers that are smaller than the square root of the value, because if a value has any factors it must have at least one factor smaller than this amount. This can be accomplished by the following procedure, which returns a true value if the integer argument is prime, and false otherwise:

```

public boolean isPrime (int n) {
    // pre: n is greater than or equal to 2
    // post: true if n is prime, false otherwise
    for (int i = 2; i * i <= n; i++) {
        // if i is a factor, then not prime
        if (0 == n % i) return false;
    }
    // if we end loop without finding factor
    // then n must be prime
    return true;
}

```

In this case, the loop terminates when the value of the loop variable exceeds the square root of the original value. Thus, the number of iterations is at most \sqrt{n} . Because each iteration of the loop is performing only a constant amount of work, the entire algorithm is said to be $O(\sqrt{n})$. Notice in this case that the loop could very well terminate early (it will almost always do so, unless the number is prime). We say in this case that the algorithmic execution time represents a bound on the *worst case* execution time. When it is difficult to characterize the execution time for every instance, a worst case bound is often the next best thing. Between these two lies a third possibility, a characterization of the *average* (or *expected*) execution time. We will later see examples where the average execution time is the characterization of greatest interest.

1.2.3 Nested Loops

As noted earlier, the execution time of a loop is the number of iterations of the loop multiplied by the execution time for the body of the loop. This becomes more complex for nested loops.

The simplest case occurs when the limits of the loops are independent of one other. In this situation, the execution time is the product of the values representing the number of times each loop will iterate multiplied by the execution time of the body. An example of this is the classic algorithm for multiplying two n by n matrices, to produce a new n by n matrix product.

```
void matprod (double [ ][ ] a, double [ ][ ] b, double [ ][ ] c) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            c[i][j] = 0.0;
            for (int k = 0; k < n; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

The number of iterations in each loop is n . The body of the innermost loop, which performs only a multiplication, an addition, and an assignment, is constant time. Therefore the total running time is $O(n^3)$.

A more complex analysis is required when the limits of iteration for the inner loops are linked to an outer loop. An example of this behavior is found in the procedure `bubbleSort`, which places the values of an array into sorted order. An outer loop provides the index for the value to be placed. An inner loop then compares elements against each other, “bubbling” larger elements to the top of the array. By the end of each iteration of the inner loop the largest remaining value will have been moved into position.

```
void bubbleSort (double [ ] v) {
    int n = v.length;
    // find the largest remaining value
    // and place into v[i]
    for (int i = n - 1; i > 0; i--) {
        // move large values to the top
        for (int j = 0; j < i; j++) {
            if (v[j] > v[j+1]) { // if out of order
                double temp = v[j]; // then swap
                v[j] = v[j + 1];
                v[j + 1] = temp;
            }
        }
    }
}
```

```

    }
  }
}
```

To determine the running time of this procedure we can simulate execution on a few values. The pattern that quickly becomes apparent is that on the first iteration of the outermost loop, the inner loop will execute $n - 1$ times. On the second iteration of the outermost loop, the inner loop will execute $n - 2$ times, and so on until on the final iteration the inner loop executes 1 time. The number of iterations is therefore the sum of $(n - 1) + (n - 2) + \dots + 1$.

Mathematical Induction

In order to determine the size of the sum $(n-1)+(n-2)+\dots+1$, we use a powerful technique, called *mathematical induction*. We will encounter mathematical induction (or induction, for short) in many different places and guises in our investigations of data structures, and thus it is useful to be well versed in the technique.

To apply mathematical induction one first forms a *hypothesis*, a statement of the result you think will hold. In this case, our hypothesis will be that the sum of the values from 1 to n is given by the formula $\frac{n(n+1)}{2}$. Discovering a hypothesis is sometimes the most difficult part of a mathematical induction proof. Sometimes the hypothesis will be provided for you naturally in the problem statement; other times it can only be discovered by, for example, looking for patterns in several different cases.

The next step is to verify this formula for one or more *base cases*. If we select 1 for n , for example, we have 1 for the sum, and $\frac{2}{2}$ for the fraction, so the result holds. If we select 2 for n , we have 3 for the sum, and $\frac{2 \times 3}{2}$ for the fraction, so the result again holds.

The final step is to verify the formula for all remaining integers. We do this by *assuming* the hypothesis holds for all integers smaller than or equal to some indeterminate value n , then proving it must therefore hold for the value $n + 1$. Doing so generally requires understanding how the hypotheses for n and $n + 1$ are linked, and *reducing* the $n + 1$ case to the size n situation.¹

For example, in our present problem we assume that the summation of values from 1 to n is $\frac{n(n+1)}{2}$. We then inquire as to the sum of the values from 1 to $n + 1$. But this can be written as $(1 + 2 + \dots + n) + (n + 1)$. Our *induction hypothesis* tells us that we can substitute $\frac{n(n+1)}{2}$ for the first term. The resulting expression is $\frac{n(n+1)}{2} + (n + 1)$, or $\frac{n(n+1)}{2} + \frac{2(n+1)}{2}$, which simplifies to $\frac{(n+1)(n+2)}{2}$. Because this matches our induction hypothesis for $n + 1$, we are done.

From this analysis, we can deduce that the number of iterations of the body of the loop in the bubble sort algorithm is $\frac{(n-1)n}{2}$. This is $\frac{n^2+n}{2}$. As we will see when we discuss the

¹Sometimes it is easier to *assume* the hypothesis holds for integers smaller than $n - 1$, and then *prove* it must therefore hold for n .

Mathematical Induction and Recursion

There is a close relationship between the technique of mathematical induction and the use of recursion as a programming technique.

- Both begin by identifying one or more *base cases* that are handled using some other means.
- Both proceed by showing how a large problem can be *reduced* to a slightly smaller problem *of the same form*.
- The analysis then proceeds by showing first that the base cases are correct, and then a conditional argument of the form that *if* the induction formula (or recursive function call) is correct, *then* the larger expression must be correct.

We will see more of both mathematical induction and recursive algorithms in later chapters.

addition of terms in algorithmic analysis, this therefore shows that the running time of this algorithm is $O(n^2)$.

1.2.4 While Loops

The analysis of while loops is similar to that of for loops. The key is to determine the number of iterations the loop will perform. If the while loop is doing a linear traversal over some range, this may be straightforward. An example is the insertion sort algorithm, which can be written as follows:

```
void insertionSort (double [ ] v) {
    int n = v.length;
    for (int i = 1; i < n; i++) {
        // move element v[i] into place
        double element = v[i];
        int j = i - 1;
        while (j >= 0 && element < v[j]) {
            v[j+1] = v[j]; // slide old value up
            j = j - 1;
        }
        // place element into position
        v[j+1] = element;
    }
}
```

```
}

```

Unlike the bubble sort algorithm, the insertion sort algorithm places the lower portion of the array into sequence first. Each new value is inserted into place, sliding elements over until the proper location for each new value is established.

The outer loop clearly executes $n - 1$ times. The inner loop *may* terminate early, but in the worst case must shift elements all the way to the bottom. (This worst case occurs if the input is initially sorted backwards, and thus each value in turn is swapped until it reaches the zeroth element). We see, therefore, that in the worst case the number of iterations of the inner loop follows the pattern $1 + 2 + 3 + 4 + \dots + (n - 1)$. As we have seen, the sum of this series is $\frac{(n-1)n}{2}$, and thus the algorithm is $O(n^2)$. (An optimist might argue that this worst case performance is rare, citing that in the best case the inner loop only executes one step, and therefore in this situation the insertion sort algorithm is $O(n)$. Unfortunately, though the mathematics is more complicated than we can present here, it is possible to show that the average case behavior of insertion sort is still $O(n^2)$.)

More complicated uses of the while loop occur when the variables involved are not simply tracing out an arithmetic progression. A classic example of a nontrivial while loop occurs in the binary search algorithm. Here we assume that the input array is an already ordered collection of values. The task is to determine if a particular value occurs in the list and, if not, the position immediately above to the location where the element would be placed.

```
int binarySearch (double [ ] data, double testValue) {
    // pre: elements in argument data are ordered smallest to largest
    // post: returns the index in data where testValue is found,
    // or index of next larger element if not in collection
    int low = 0;
    int high = data.length;

    // repeatedly reduce the area of search
    // until it is just one value
    while (low < high) {
        mid = (low + high) / 2;
        if (data[mid] < testValue)
            low = mid + 1;
        else
            high = mid;
    }
    return low;
}

```

The program works by repeatedly dividing in half the range of values being searched. Because there were originally n values, we know the number of times the collection can be subdivided is no larger than roughly $\log n$ (see nearby box on logarithms). This is sufficient to tell us that the entire procedure is $O(\log n)$.

Logarithms

To the mathematician, a *logarithm* is generally envisioned as the inverse of the exponential function, or perhaps it is associated with the integral calculus (that is, $\log_e a = \int_1^a \frac{1}{x} dx$). To a computer scientist, the intuition concerning the log function should be something very different.

The log (base n) of a positive value x is *approximately* equal to the number of times that x can be divided by n .

Most often the log function will arise when quantities are repeatedly split in half. This is why logarithms in computer science are, almost invariably, used with a base value two.

The log (base 2) of a positive value x is *approximately* equal to the number of times that x can be divided in half.

The word “approximately” is used, because the log function yields a fractional value, and the exact figure can be as much as one larger than the integer ceiling of the log. But, as we have already noted, integer constants can be safely ignored when discussing asymptotic bounds.

1.2.5 Function Calls

When function or procedure calls occur, the running time of the call is taken to be the running time of the associated procedure. For example, suppose we wished to print the values of all prime numbers less than n . We could use an algorithm such as the following:

```
public void printPrimes (int n) {
    for (int i = 2; i <= n; i++) {
        if (isPrime(i))
            System.out.println("value " + i + " is prime");
        else
            System.out.println("value " + i + " is not prime");
    }
}
```

We know the execution time of the `isPrime` routine is $O(\sqrt{n})$. (In fact, we can make the even stronger statement that it is $O(\sqrt{i})$, but doing so results in a summation that is difficult to analyze, so we will bound each call by the larger limit.) Because we are making roughly n calls, the total running time of the procedure is no greater than $O(n\sqrt{n})$.

1.3 Summing Algorithmic Execution Times

Consider the following procedure for initializing an n by n matrix as an identity matrix.

```
public void makeIdentityMatrix (double [ ] [ ] m) {
    int n = m.length;
        // first make matrix of all zeros
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            m[i, j] = 0.0;
        }
    }

        // then place ones along diagonal
    for (i = 0; i < n; i++) {
        m[i, i] = 1.0;
    }
}
```

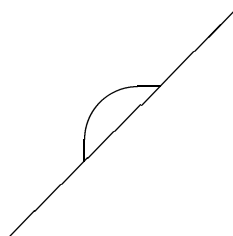
Clearly the first set of loops has an $O(n^2)$ execution time, and just as clearly the latter loop has $O(n)$ behavior. One might be tempted to assert that the entire process therefore has $O(n^2 + n)$ performance. Instead, in this section we will argue for a simpler rule, namely, *when adding algorithmic complexities, the larger value dominates*. Therefore, the entire algorithm is $O(n^2)$.

To know what functions will dominate others, we must have some sort of ranking. The following table gives one such ordering, with the fastest growing functions listed near the top, and the slower growing functions listed below. Formally, we say that a function $f(n)$ dominates a function $g(n)$ if there exists a constant value n_0 such that for all values $n > n_0$, it is the case that $g(n) < f(n)$.

Also shown in this table is the common name used to describe the algorithmic behavior. We use such names when we say, for example, that the matrix multiplication algorithm given in Section ?? is cubic, or the bubble sort algorithm is quadratic.

<i>Function</i>	<i>Common name</i>
$n!$	Factorial
2^n	Exponential
$n^d, d > 3$	Polynomial
n^3	Cubic
n^2	Quadratic
$n\sqrt{n}$	
$n \log n$	
n	Linear
\sqrt{n}	Root- n
$\log n$	Logarithmic
1	Constant

In the following discussion we will motivate this rule in a variety of fashions. The first is intended merely as an intuitive illustration. The reader has probably had the experience of sitting in a car during a rainstorm, and may have noted that small raindrops will stay fixed on the angled front window, even if the car remains at rest. If more water collects in the drop, however, it eventually falls off. There is a certain limit in size beyond which drops seemingly cannot remain fixed on the window.



The force pulling the drop down is gravity, while the force permitting the drop to remain on the windscreen is friction, or the surface tension between the drop and the glass. We can idealize the situation slightly and consider the drop to be a perfect hemisphere. If r represents the radius of the drop, the surface area between the drop and the plane is πr^2 . Gravity, on the other hand, operates on the entire volume of the drop, which is proportional to r^3 . Thus, the force of gravity can be described by $c \times r^3$, for some unknown constant c , and the force of surface tension is similarly described by $d \times r^2$, making use of some unknown constant d .

The observed phenomenon is that small drops will remain on the surface. This is the situation when the force of the surface tension is greater than that of gravity. As the drop becomes larger, its radius increases. Eventually, no matter what the constants may be, a cubic (r^3) function will always become larger than a quadratic (r^2) function. Thus, we would predict that large drops must always fall off the surface, because gravity prevails over surface tension. This, of course, matches what we observe.

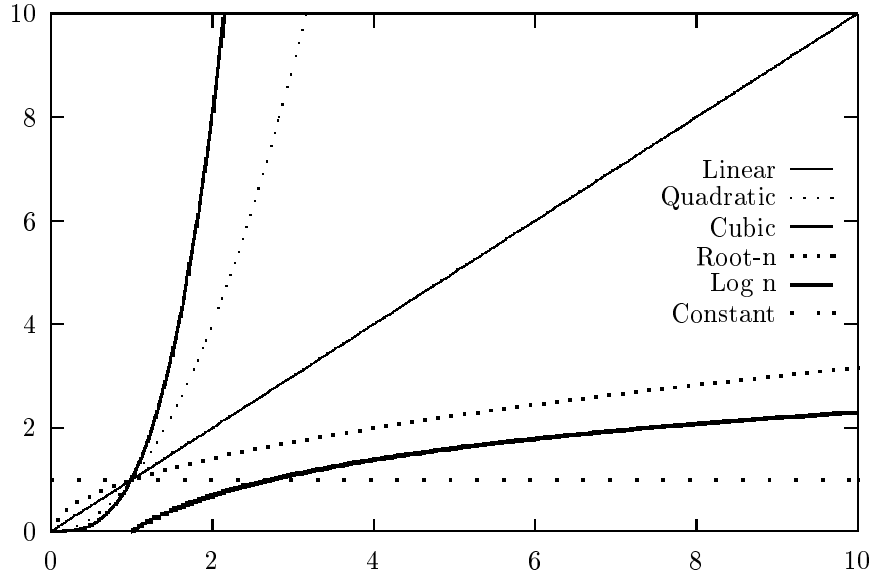


Figure 1.1: Characteristic curves for various functions

We can see this behavior in a graphical fashion by noting the curves given by various functions. All functions possess a certain characteristic curve. Figure ?? illustrates these curves for various functions. An n^3 function will always grow faster than, and thus always eventually surpass an n^2 function, regardless of the constant values involved either as coefficient on the function or as an additive amount. This is shown in Figure ??, where the function $\frac{n^2}{4}$ is compared against the amount $200 + n \times \log n$. Although the latter is initially much larger, the n^2 function must always eventually dominate.

When two functions of different orders of magnitude are combined, the larger function will dominate the smaller one, so that the characteristic shape of the result matches the larger, and not the smaller. This is shown in Figure ??, which compares the functions n , n^2 , and the combination $n^2 + n$.

Another way to understand this is to consider a few actual values for the various functions. Assume, for example, that we can perform one operation every microsecond (that is, 10^6 operations per second), and we have some task that requires an input of size 10^5 . The figures in Table ?? illustrate how long it would take to perform this task, assuming various different running times. The multiplication of the dominant function by a constant will, of course, change the running time, but only by a constant factor. Consider a task that requires $n^2 + n \log n$ steps. The n^2 part will mean the task will require several hours to complete, while the $n \log n$ component will add an insignificant few additional seconds to the execution time. Taking all of these arguments into consideration, it seems intuitive that

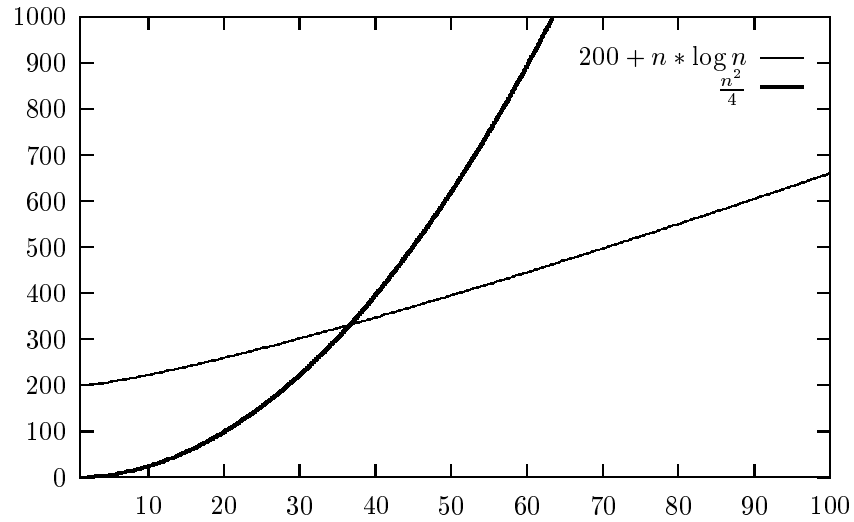


Figure 1.2: Comparing $n \log n$ and n^2 growth

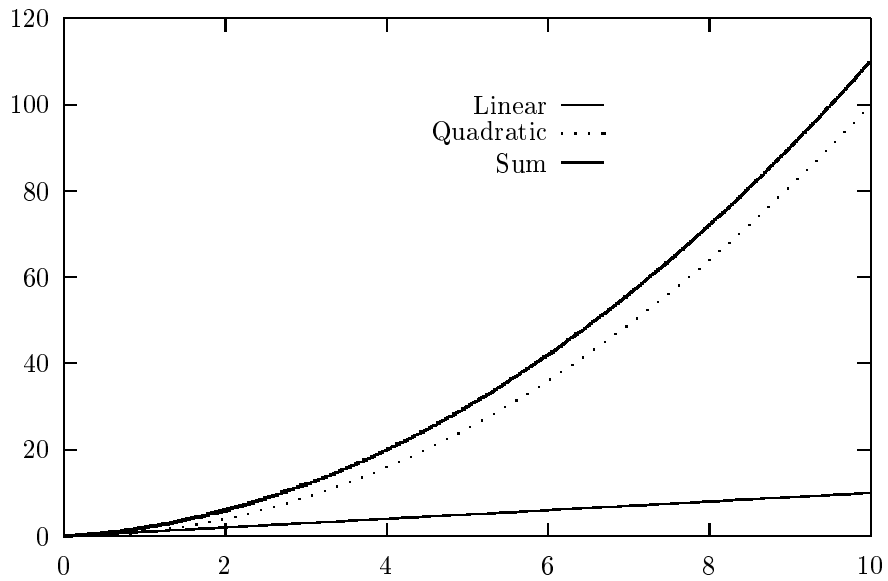


Figure 1.3: Addition of a linear and a quadratic function

function	Running time
2^n	More than a century
n^3	31.7 years
n^2	2.8 hours
$n\sqrt{n}$	31.6 seconds
$n \log n$	1.2 seconds
n	0.1 seconds
\sqrt{n}	3.2×10^{-4} seconds
$\log n$	1.2×10^{-5} seconds

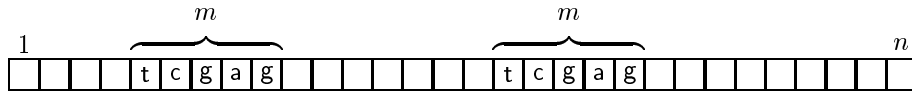
Table 1.1: Execution times for a task using $n = 10^5$ input values, assuming 10^6 operations can be performed per second

when we add algorithmic execution times, it is safe to ignore all terms except the dominant function. (For those who have a natural dislike for intuitive arguments, some of the exercises at the end of the chapter will place this assertion on a more formal grounding.)

1.4 The Importance of Fast Algorithms

The following story will help illustrate both the importance of finding fast algorithms and data structures, and that the most obvious approach to solving a problem may not necessarily be the best.

Several years ago a student working in genetic research was faced with a task involved in the analysis of DNA sequences. The problem could be reduced to a relatively simple form. The DNA is represented as a array of n integer values, where n is very large (on the order of tens of thousands). The problem was to discover whether any pattern of length m , where m was a fixed and small constant (say five or ten) is ever repeated in the array of values.



The programmer dutifully sat down and wrote a simple and straightforward program, something like the following:

```
for (int i = 0; i < (n-m); i++)
  for (int j = i+1; j < (n-m); j++) {
    boolean found = true;
    for (k = 0; k < m; k++) {
      if (x[i+k] != x[j+k])
```

```

        found = false;
    }
    if (found) {
        ...
    }
}

```

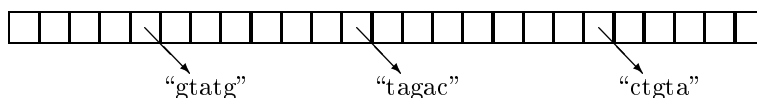
The student was somewhat disappointed when trial runs indicated his program would need several days to complete. He discussed his problem with a second student, who suggested a different approach. Rather than working with a array of n elements, the second student reorganized the data into a matrix with roughly n rows and m columns:

x_1	x_2	...	x_m
x_2	x_3	...	x_{m+1}
...			...
x_{n-m}	x_{n-1}
$x_{n-(m-1)}$...	x_{n-1}	x_n

The second student then wrote a program to sort this matrix by rows. If any pattern was repeated, then two adjacent rows in the sorted matrix would have identical values. It was a trivial matter to check for this condition.

As we will see in later chapters, sorting can be made an $O(n \log n)$ time operation. Sorting a matrix can be performed by an algorithm that is $O(m \times n \log n)$, whereas the original program was $O(m \times n^2)$. This was a considerable improvement, reducing the estimated execution time from several days to slightly less than one day.

At this point a third student entered the room, and said that she had a technique that could reduce the execution time even further. The approach described by the third student was to take each section of m values, and reduce it to a single integer value, for example by adding the elements. This is a technique called *hashing*, and we will examine it in detail in Chapter ???. The student then constructed another array of roughly n elements. For each hashed entry, she placed a string representing the run of m characters at the location given by the hash value taken mod the size of the array:



If two elements were equal, they would have the same hash value. It was therefore a simple matter to examine the array prior to placing each value to see if the same value had already been seen. Since all this could be performed in one pass through the original data array, it was $O(m \times n)$, which was even faster than the $O(m \times n \log n)$ solution. Using this technique, the student was able to reduce the data analysis phase of his work from several days to well under one day of execution time.

```

public class TaskTimer {

    public TaskTimer (Reporter ioutput) { output = ioutput; }
    private Reporter output;

    public void initialize (int i) { } // default is nothing

    public void doTask (int i) { } // override this method

    public void run (int start, int stop, int step) {
        for(int i = start; i <= stop; i += step) {
            initialize(i);
            System.gc(); // do garbage collection
            long startTime = System.currentTimeMillis();
            doTask(i);
            long stopTime = System.currentTimeMillis();
            int time = (int) (stopTime - startTime);
            output.addPoint(i, time);
        }
    }
}

```

Figure 1.4: The class TaskTimer

1.5 Benchmarking Actual Execution Times

While an asymptotic characterization is normally the first and best way to describe the execution time behavior of a complex algorithm, it is not the final word. For example, how do we compare the execution time of bubble sort and insertion sort, once we have determined that they are both $O(n^2)$ algorithms? For that, we must compare the two algorithms by their actual running times on some machine.

The library of classes described in this book contains a simple utility, `GraphMaker`, that we will use to perform timings of algorithms. A `GraphMaker` is a `Canvas`, which means it can be used as a standard Java graphical object. Typically the `GraphMaker` is a central portion of an application display. The constructor for the class `GraphMaker` takes two integer arguments, which represent the maximum x and maximum y values for the graph to be displayed. Elements are added to a graph by means of a `Reporter`, a helper class that accepts x and y pairs.

Another utility class, `TaskTimer`, will simplify the creation of graphs. This class, shown in Figure ??, provides three methods in addition to the constructor. The constructor requires a reporter that will be used to plot the values. The methods `initialize` and `doTask` are

The Java Big Bang

In Chapter ?? we noted that an object-oriented program can be envisioned as a universe of interacting objects. It is helpful to keep this image in mind when considering how Java program typically start execution. The procedure `main` must be declared both `public` and `static`. The latter characteristic is necessary since only `static` methods can be executed prior to the creation of any instances of a class.

Often, as in Figure ??, the most important action the `main` program must perform is to create the first object that will occupy the object-oriented universe. (Sometimes it is the *only* action). The constructor for this class will then proceed to create other objects, until all the necessary elements in the universe are formed. In this program we create a `GraphMaker` and instances of two different subclasses of `TaskTimer`. The `GraphMaker` will, in turn, create instances of `Reporter`. Finally, the program is set in motion, and the output is displayed.

overwritten by the user to specify the task to be performed. The method `run` repeatedly executes the given tasks, computing the execution time, and recording the execution times using the reporter.

The code in Figure ?? shows how we can create a subclass of `TaskTimer` to compute the running time of the bubble sort algorithm described earlier in Section ?. This figure also shows how to create the main application that combines the `GraphMaker` and the `TaskTimers`. Running the `BubbleExperiment` program produces output such as that shown in Figure ?. Note that although the execution timings for the two algorithms show clear trends, they are not precise. This is characteristic of actual execution timings. Features such as memory management, the need for the hardware to respond to other processes, and other uncertain factors can combine to make small (or sometimes, very large) deviations in execution timings. Thus no single timing should ever be taken to be representative of the performance of an algorithm. Instead, a number of repeated timings should be obtained, so as to smooth out any transient effects.

We can see from Figure ?? that although Bubble Sort and Insertion Sort have the same asymptotic execution time (they are both $O(n^2)$), in practice insertion sort tends to be slightly faster.

```
class BubbleTime extends TaskTimer {
    BubbleTime (Reporter out) { super(out); }

    public void initialize (int n) {
        data = new double[n]; // create new array
        for (int i = 0; i < n; i++) // fill with random values
            data[i] = n * Math.random();
    }

    public void doTask (int n) { bubbleSort(data); }

    private double [ ] data;

    private void bubbleSort (double [ ] v) { ... }
}

public class BubbleExperiment extends Frame {

    public static void main (String [ ] args) {
        BubbleExperiment world = new BubbleExperiment();
        world.show(); world.run();
    }

    public BubbleExperiment () {
        setTitle("Bubble Sort and Insertion Sort Comparison");
        setSize(500, 300);
        add("Center", graph);
    }

    private GraphMaker graph = new GraphMaker(5000, 3000);

    public void run() {
        BubbleTime bTimer = new BubbleTime(graph.getReporter("Bubble"));
        bTimer.run(0, 5000, 100);
        InsertionTime iTimer =
            new InsertionTime(graph.getReporter("Insertion"));
        iTimer.run(0, 5000, 100);
    }
}
```

Figure 1.5: The class BubbleExperiment

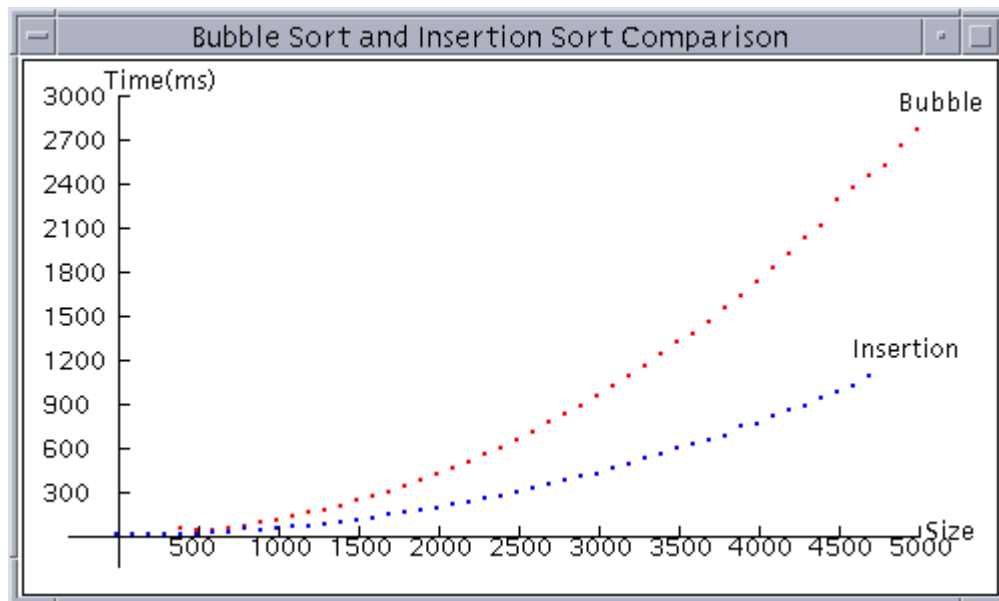


Figure 1.6: Execution Timings for Bubble and Insertion Sort

Experiment: Changing the Input Distribution

Throughout the book we will suggest various experiments that the reader may wish to perform in order to gain a deeper understanding of different topics. One interesting experiment relevant to the output shown in Figure ?? is to consider the impact of the distribution of values in the original unsorted array, and whether an alternative distribution will produce radically different results. We noted, for example, that the execution time of the `InsertionSort` algorithm will vary considerably depending upon the characteristics of the input values.

We can create an “almost sorted” input array by changing the way that we initialize the data array in Figure ?. Consider the following:

```
public void initialize (int n) {
    data = new double[n]; // create new array
    for (int i = 0; i < n; i++)
        data[i] = i + 3 * Math.random();
}
```

Since `Math.random` returns a value larger than or equal to 0 and smaller than 1, if we multiply this by 3 we get a value between 0 and 3. If we add this to i we get a value between i and $i + 3$. Since the values are random, we are not guaranteed that they will be monotonically increasing, but they should generally get larger. Will this change the result we see in Figure ??? Can you figure out how to create a array that is “almost reverse sorted”?

1.6 Chapter Summary

On a coarse level, algorithms can be compared by their algorithmic complexity. The algorithmic characterization of an algorithm is usually given by a “big-Oh” expression. This expression characterizes the behavior of the algorithm as input values grow ever larger. However, even with relatively small inputs, an algorithm with a smaller algorithmic complexity will usually execute more quickly than one with a larger algorithmic complexity.

Mathematical induction is a powerful analysis technique used in proving properties of general formulas. In computer science, mathematical induction is used in the analysis of program fragments that are executed repeatedly, either as loops or as recursive algorithms. Mathematical induction is used in proving termination, analyzing execution time, and proving correctness.

Mathematical induction and the development of recursive algorithms are very similar. Some of the common features of both include:

- The discovery of base cases that can be handled by other mechanisms, and to which all larger problems will ultimately be reduced.
- The discovery of how a larger problem can be reduced to a slightly smaller problem

of the same form.

In comparing two algorithms with the same algorithmic complexity, actual running times, or benchmarks, may be employed. While a benchmark can provide an accurate time estimate for a particular machine and particular input conditions, it is sometimes difficult (frequently impossible) to extrapolate such a number to new machines or different input values.

Key Concepts

- Algorithmic complexity
- Big-Oh notation
- Recursion
- Mathematical induction
- Benchmarks

Further Reading

The basic concepts of computational complexity were originally developed by Juris Hartmanis and Richard E. Stearns [?], for which they were awarded the ACM Turing award in 1993. As we noted in Chapter ??, the study of algorithms was popularized by Donald Knuth in his three-part series [?, ?, ?].

Study Questions

1. Why is a benchmark not generally a good characterization of the running time of an algorithm?
2. What does it mean to say that an algorithm is $O(f(n))$ for some function $f(n)$?
3. What is the formula that describes the algorithmic execution time of a conditional (if) statement? How about a `switch` statement?
4. What is the algorithmic running time of the procedure named `power`, used to raise a double precision value to an integer exponent, that we earlier described in Section ???
5. What are the parts of an argument that uses mathematical induction?
6. What is the best and worst case algorithmic running time of the insertion sort algorithm?
7. To a computer scientist, what should be the intuitive meaning of the log function?

8. What are the two pieces of information needed to characterize the algorithmic running time of a recursive procedure?
9. What rule is used in adding two algorithmic complexity formulas?
10. What are some situations where a benchmark provides more useful information than an algorithmic complexity characterization?

Exercises

1. The three parts of this exercise are intended to place the concept of algorithmic analysis on a slightly more firm theoretical foundation. In particular, these exercises will help support the idea that (a) additive constants can be ignored, (b) constant coefficients can be ignored, and (c) when adding two algorithmic complexities, only the dominant function need be considered.

Recall that the definition of big-Oh notation said that a function $f(n)$ is $O(g(n))$ if there exist constants n_0 and c such that for all values $n > n_0$, $f(n) < c \times g(n)$.

- (a) Assume that $f(n)$ is $O(g(n))$, where $f(n)$ and $g(n)$ are both functions of n , $g(n) > 1$ for all n . Demonstrate that $f(n) + c$, for any constant c , is still $O(g(n))$.
Hint: Show that there exists some new constant c_2 , which bounds $f(n) + c$.
 - (b) Assume that $f(n)$ is $O(g(n))$. Demonstrate that $c \times f(n)$, for any constant c , is still $O(g(n))$.
 - (c) Assume $f_1(n) < f_2(n)$ for all values larger than some n_0 , and that $f_2(n)$ is $O(g(n))$. Show that $f_1(n) + f_2(n)$ is $O(g(n))$.
2. Prove that the function ax^i is $O(x^{i+j})$ for any value $j \geq 0$.
 3. Using the proof from the previous question, prove that any polynomial $a + bx + cx^2 + \dots + hx^i$ is $O(x^i)$, that is, that the largest polynomial term will dominate.
 4. Prove that the function $\log_a n$ is $O(\log_2 n)$ for any constant value a (hence, we need not state the base of a log when using big-Oh notation).
 5. Prove, using mathematical induction, that a common multiplier can be factored out of a summation. That is, for any constant c and nonnegative integer n :

$$\sum_{i=1}^n c \times f(i) = c \times \sum_{i=1}^n f(i)$$

6. Prove, by mathematical induction, that the sum of powers of 2 is one less than the next higher power. That is, for any nonnegative integer n :

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1$$

7. Suppose by careful measurement we have discovered a function that describes the precise running time of some algorithm. For each of the following such functions, describe the algorithmic running time in big-Oh notation.

- (a) $3n^2 + 3n + 7$
- (b) $(5 * n) * (3 + \log n)$
- (c) $\frac{5n+4}{6}$
- (d) $1 + 2 + 3 + 4 + \dots + n$
- (e) $n + \log n^2$
- (f) $\frac{(n+1) \log n}{2}$

8. For each of the following program skeletons, describe the algorithmic execution time as a function of n . You can assume the remaining portions of the loops require only constant execution time.

- (a)

```
for (int i = 0; i < n; i++) {
    ...
}
for (int j = n; j >= 0; j--) {
    ...
}
```
- (b)

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        ...
    }
}
```
- (c)

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        ...
    }
}
```
- (d)

```
for (int i = n; i > 0; i = i / 2) {
    ...
}
```
- (e)

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j * j < n; j++) {
        ...
    }
}
```

```
(f) for (int i = n; i > 0; i = i >> 1) {
    for (int j = 0; j < n; j++) {
        ...
    }
}
```

9. Suppose we have an n^2 algorithm that for $n = 80$ runs slightly longer than one hour. One day we discover an alternative algorithm that runs in time $n \log n$. If we assume the constants of proportionality are about the same, about how long would we expect the new program to run?
10. Simulate the execution of bubble sort (Section ??) on the array of value 7 2 3 9 4. Show the state of the array at the end of each iteration of the outermost loop.
11. Simulate the execution of insertion sort (Section ??) on the array of value 7 2 3 9 4. Show the state of the array at the end of each iteration of the outermost loop.

Programming Projects

1. Can you rewrite the binary search algorithm as a recursive procedure? Perform execution timings on the resulting algorithm to compare the actual running time to that of the original.
2. The *Selection Sort* algorithm is similar to bubble sort, but rather than swapping elements in the inner loop, it merely locates the position of the currently largest value, then performs one swap in the outer loop. Implement this algorithm, and compare the running time to that of bubble sort and insertion sort. Is it better than bubble sort? Is it faster than insertion sort?
3. Convert the insertion sort algorithm so that it will work on an array of `Object` values, using a `Comparator` (see Section ??) to determine the ordering of the elements.
4. Using the `TaskTimer` class, plot the execution time for the method `isPrime` for all integer values between 1 and 1000. Does the resulting curve appear to be \sqrt{n} ? (Because the total execution time is so small, you may need to repeatedly execute the function call 100 times to get a meaningful value).
5. Using the `TaskTimer` class, plot the number of moves required to solve the Hanoi puzzle as a function of n . What type of curve do you think this represents?
6. Using the `TaskTimer` class, provide evidence that `binarySearch` is an $O(\log n)$ algorithm. To do this, in the initialization phase of `TaskTimer` create an array of n random integers between 1 and n and sort them, so that they will be in order. Then in the timed section search for a random value between 1 and n . Perform this task for a sequence of increasing values of n , and plot the execution times.

7. The Fibonacci numbers are described by the formula $F(0) = 0$, $F(1) = 1$, and the recursive formula $F(n) = F(n - 2) + F(n - 1)$. Write two routines for computing the n^{th} Fibonacci number. The first routine should compute the value iteratively; given an argument value n , it first computes $F(0)$, then $F(1)$, and from these each value $F(i)$ until i reaches n . The second algorithm should work recursively. Given an argument n , if n is larger than 1 it will recursively call itself to compute the two earlier Fibonacci numbers, then return their sum. Calculate and compare execution timings for these two algorithms. Can you explain the difference in execution times?
8. The `GraphMaker` utility can be used to plot any type of graph, not simply those generated using the `TaskTimer`. Write an application that will take an array of integer values, determine the size of the array and the maximum and minimum values in the array, then create a graph of the appropriate size and plot the values.
9. A number of interesting mathematical problems involve random walks. Here is a simple one-dimensional example that can be solved using an array. Imagine a drunken cockroach is placed in the middle square of an array of n elements. The bug wanders randomly back and forth. Assuming at each step the bug will move with *equal probability* to either the next higher or the next lower square, how long will it take for the bug to reach the end of the the array? How many times will he cross each square?

While difficult to solve mathematically, this type of problem is easily solved using a simulation. Program a simulation for the one-dimensional drunken cockroach problem for an array of size n . Then, using the plotting utility you developed in the previous programming project, display the number of times the bug stepped on each square.

