DATA STRUCTURES

Abstraction and Design Using Java

THIRD EDITION

ELLIOT B. KOFFMAN
Temple University

PAUL A. T. WOLFGANG
Temple University

WILEY

First, we will discuss algorithm efficiency and how to characterize the efficiency of an algorithm. You will learn about big-O notation, which you can use to compare the relative efficiency of different algorithms.

Lists and the Collections Framework

2.1 Algorithm Efficiency and Big-O

Whenever we write a new class, we will discuss the efficiency of its methods so that you know how they compare to similar methods in other classes. You can't easily measure the amount of time it takes to run a program with modern computers. When you issue the command

```
java MyProgram
```

(or click the Run button of your integrated development environment [IDE]), the operating system first loads the Java Virtual Machine (JVM). The JVM then loads the .class file for MyProgram, it then loads other .class files that MyProgram references, and finally your program executes. (If the .class files have not yet been created, the Java IDE will compile the source file before executing the program.) Most of the time it takes to run your program is occupied with the first two steps. If you run your program a second time immediately after the first, it may seem to take less time. This is because the operating system may have kept the files in a local memory area called a cache. However, if you have a large enough or complicated enough problem, then the actual running time of your program will dominate the time required to load the JVM and .class files.

Because it is very difficult to get a precise measure of the performance of an algorithm or program, we normally try to approximate the effect of a change in the number of data items, n, that an algorithm processes. In this way, we can see how an algorithm's execution time increases with respect to n, so we can compare two algorithms by examining their growth rates.

For many problems, there are algorithms that are relatively obvious but inefficient. Although every day computers are getting faster, with larger memories, there are algorithms whose growth rate is so large that no computer, no matter how fast or with how much memory, can solve the problem above a certain size. Furthermore, if a problem that has been too large to be solved can now be solved with the latest, biggest, and fastest supercomputer, adding a few more inputs may make the problem impractical, if not impossible, again. Therefore, it is important to have some idea of the relative efficiency of different algorithms. Next, we see how we might obtain such an idea by examining three methods in the following examples.

EXAMPLE 2.1 Consider the following method, which searches an array for a value:

```
public static int search(int[] x, int target) {
    for (int i = 0; i < x.length; i++) {
         if (x[i] == target)
             return i;
    }
    // target not found
    return -1;
}
```

If the target is not present in the array, then the **for** loop body will be executed `x.length` times. If the target is present, it could be anywhere. If we consider the average over all cases where the target is present, then the loop body will execute `x.length/2` times. Therefore, the total execution time is directly proportional to `x.length`. If we doubled the size of the array, we would expect the time to double (not counting the overhead discussed earlier).

EXAMPLE 2.2 Now let us consider another problem. We want to find out whether two arrays have no common elements. We can use our `search` method to search one array for values that are in the other.

```
/** Determine whether two arrays have no common elements.
    @param x One array
    @param y The other array
    @return true if there are no common elements
*/
public static boolean areDifferent(int[] x, int[] y) {
    for (int i = 0; i < x.length; i++) {
        if (search(y, x[i]) != -1)
            return false;
    }
    return true;
}
```

The loop body will execute at most `x.length` times. During each iteration, it will call method `search` to search for current element, `x[i]`, in array `y`. The loop body in `search` will execute at most `y.length` times. Therefore, the total execution time would be proportional to the product of `x.length` and `y.length`.

EXAMPLE 2.3 Let us consider the problem of determining whether each item in an array is unique. We could write the following method:

```
/** Determine whether the contents of an array are all unique.
    @param x The array
    @return true if all elements of x are unique
*/
public static boolean areUnique(int[] x) {
    for (int i = 0; i < x.length; i++) {
        for (int j = 0; j < x.length; j++) {
            if (i != j && x[i] == x[j])
                return false;
        }
    }
    return true;
}
```

If all values are unique, the outer loop will execute `x.length` times. For each iteration of the outer loop, the inner loop will also execute `x.length` times. Thus, the total number of times the body of the inner loop will execute is $(x.length)^2$.

EXAMPLE 2.4 The method we showed in Example 2.3 is very inefficient because we do approximately twice as many tests as necessary. We can rewrite the inner loop as follows:

```
/** Determine whether the contents of an array are all unique.
    @param x The array
    @return true if all elements of x are unique
*/
public static boolean areUnique(int[] x) {
    for (int i = 0; i < x.length; i++) {
        for (int j = i + 1; j < x.length; j++) {
            if (x[i] == x[j])
                return false;
        }
    }
    return true;
}
```

We can start the inner loop index at i + 1 because we have already determined that elements preceding this one are unique. The first time, the inner loop will execute `x.length`–1 times. The second time, it will execute `x.length`–2 times, and so on. The last time, it will execute just once. The total number of times it will execute is

```
x.length-1 + x.length-2 +. . .+ 2 + 1
```

The series $1 + 2 + 3 + \cdots + (n - 1)$ is a well-known series that has a value of

$$\frac{n \times (n-1)}{2} \text{ or } \frac{n^2 - n}{2}$$

Therefore, this sum is

$$\frac{x.length^2 - x.length}{2}$$

Big-O Notation

Today, the type of analysis just illustrated is more important to the development of efficient software than measuring the milliseconds in which a program runs on a particular computer. Understanding how the execution time (and memory requirements) of an algorithm grows as a function of increasing input size gives programmers a tool for comparing various algorithms and how they will perform. Computer scientists have developed a useful terminology and notation for investigating and describing the relationship between input size and execution time. For example, if the time is approximately doubled when the number of inputs, n, is doubled, then the algorithm grows at a linear rate. Thus, we say that the growth rate has an order of n. If, however, the time is approximately quadrupled when the number of inputs is doubled, then the algorithm grows at a quadratic rate. In this case, we say that the growth rate has an order of n^2.

In the previous section, we looked at four methods: one whose execution time was related to `x.length`, another whose execution time was related to `x.length` times `y.length`, one whose execution time was related to `(x.length)`^{2}, and one whose execution time was related to `(x.length)`^{2} and `x.length`. Computer scientists use the notation **O**(n) to represent the first case, **O**($n \times m$) to represent the second, and **O**(n^2) to represent the third and fourth, where n is `x.length` and m is `y.length`. The symbol **O** (which you will see in a variety of typefaces and styles in computer science literature) can be thought of as an abbreviation for "order of magnitude." This notation is called *big-O notation*.

Often, a simple way to determine the big-**O** of an algorithm or program is to look at the loops and to see whether the loops are nested. Assuming that the loop body consists only of simple statements, a single loop is **O**(n), a pair of nested loops is **O**(n^2), a nested loop pair inside another is **O**(n^3), and so on. However, you also must examine the number of times the loop executes.

Consider the following:

```
for (i = 1; i < x.length; i *= 2) {
    // Do something with x[i]
}
```

The loop body will execute $k - 1$ times, with `i` having the following values: 1, 2, 4, 8, 16, 32, . . ., 2^k until 2^k is greater than `x.length`. Since $2^{k-1} =$ `x.length` $< 2^k$ and $\log_2 2^k$ is k, we know that $k - 1 = \log_2$`(x.length)` $< k$. Thus, we say that this loop is **O**(log n). The logarithm function grows slowly. The log to the base 2 of 1,000,000 is approximately 20. Typically, in analyzing the running time of algorithms, we use logarithms to the base 2.

Formal Definition of Big-O

Consider a program that is structured as follows:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
         Simple Statement
    }
}
for (int k = 0; i < n; k++) {
     Simple Statement 1
     Simple Statement 2
     Simple Statement 3
     Simple Statement 4
     Simple Statement 5
}
Simple Statement 6
Simple Statement 7
...
Simple Statement 30
```

Let us assume that each *Simple Statement* takes one unit of time and that the **for** statements are free. The nested loop executes a *Simple Statement* n^2 times. Then five *Simple Statements* are executed n times in the loop with control variable k. Finally, 25 *Simple Statements* are executed after this loop. We would then conclude that the expression

$$T(n) = n^2 + 5n + 25$$

shows the relationship between processing time and n (the number of data items processed in the loop), where T(n) represents the processing time as a function of n. It should be clear that the n^2 term dominates as n becomes large.

In terms of T(n), formally, the big-O notation

$$T(n) = O(f(n))$$

means that there exist two constants, n_0 and c, greater than zero, and a function, f(n), such that for all $n > n_0$, $cf(n) \geq T(n)$. In other words, as n gets sufficiently large (larger than n_0), there is some constant c for which the processing time will always be less than or equal to $cf(n)$, so $cf(n)$ is an upper bound on the performance. The performance will never be worse than $cf(n)$ and may be better.

If we can determine how the value of f(n) increases with n, we know how the processing time will increase with n. The growth rate of f(n) will be determined by the growth rate of the fastest-growing term (the one with the largest exponent), which in this case is the n^2 term. This means that the algorithm in this example is an $O(n^2)$ algorithm rather than an $O(n^2 + 5n + 25)$ algorithm. In general, it is safe to ignore all constants and drop the lower-order terms when determining the order of magnitude for an algorithm.

EXAMPLE 2.5 Given $T(n) = n^2 + 5n + 25$, we want to show that this is indeed $O(n^2)$. Thus, we want to show that there are constants n_0 and c such that for all $n > n_0$, $cn^2 > n^2 + 5n + 25$.

One way to do this is to find a point where

$$cn^2 = n^2 + 5n + 25$$

If we let n be n_0 and solve for c, we get

$$c = 1 + 5/n_0 + 25/{n_0}^2$$

We can evaluate the expression on the right easily when n_0 is 5(1 + 5/5 + 25/25). This gives us a c of 3. So $3n^2 > n^2 + 5n + 25$ for all n greater than 5, as shown in Figure 2.1.

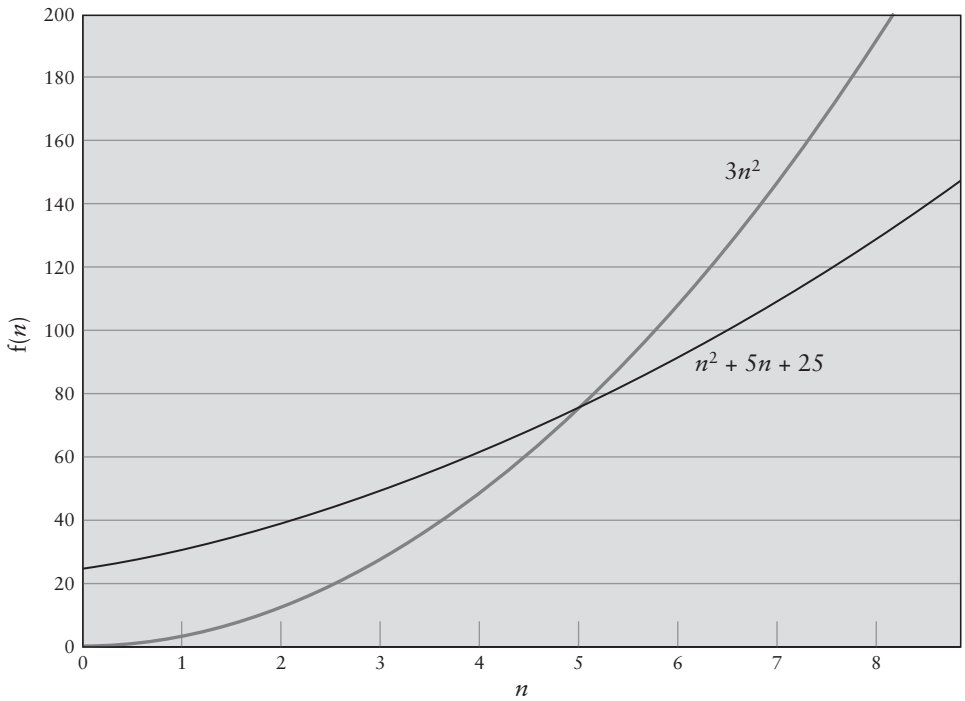


FIGURE 2.1
$3n^2$ versus $n^2 + 5n + 25$

EXAMPLE 2.6 Consider the following program loop:

```
for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
        3 simple statements
    }
}
```

The first time through the outer loop, the inner loop is executed $n - 1$ times; the next time, $n - 2$; and the last time, once. The outer loop is executed n times. So we get the following expression for T(n):

$$3(n-1) + 3(n-2) + \cdots + 3(2) + 3(1)$$

We can factor out the 3 to get

$$3((n-1) + (n-2) + \cdots + 2 + 1)$$

The sum $1 + 2 + \cdots + (n-2) + (n-1)$ (in parentheses above) is equal to

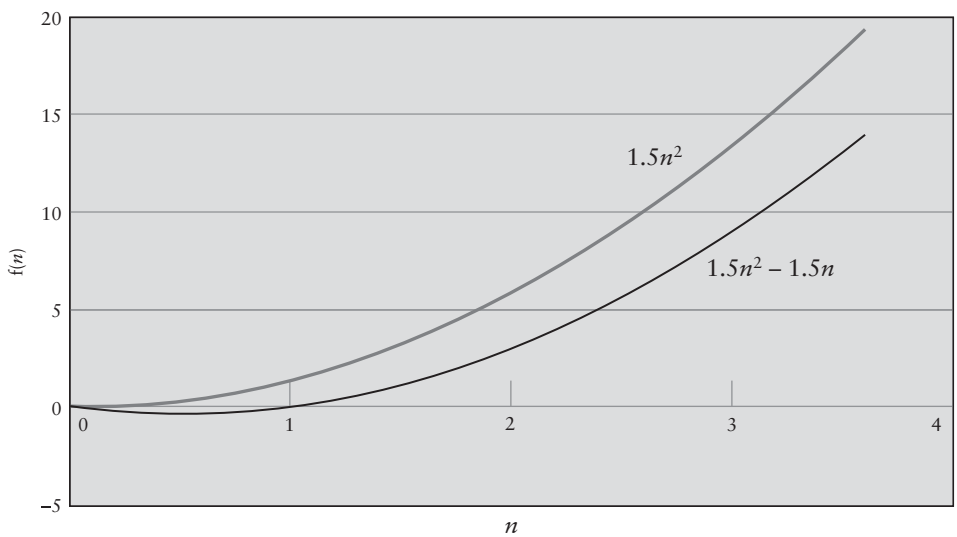
$$\frac{n^2 - n}{2}$$

Thus, our final $T(n)$ is

$$T(n) = 1.5n^2 - 1.5n$$

This polynomial is zero when n is 1. For values greater than 1, $1.5n^2$ is always greater than $1.5n^2 - 1.5n$. Therefore, we can use 1 for n_0 and 1.5 for c to conclude that our T(n) is O(n^2) (see Figure 2.2).

FIGURE 2.2
$1.5n^2$ versus $1.5n^2 - 1.5n$



If T(n) is the form of a polynomial of degree d (the highest exponent), then it is O(n^d). A mathematically rigorous proof of this is beyond the scope of this text. An intuitive proof is demonstrated in the previous two examples. If the remaining terms have positive coefficients, find a value of n where the first term is equal to the remaining terms. As n gets bigger than this value, the n^d term will always be bigger than the remaining terms.

We use the expression **O**(1) to represent a constant growth rate. This is a value that doesn't change with the number of inputs. The simple steps all represent **O**(1). Any finite number of **O**(1) steps is still considered **O**(1).

Summary of Notation

In this section, we have used the symbols T(n), f(n), and **O**(f(n)). Their meaning is summarized in Table 2.1.

TABLE 2.1
Symbols Used in Quantifying Software Performance

Symbol	Meaning
T(n)	The time that a method or program takes as a function of the number of inputs, n. We may not be able to measure or determine this exactly
f(n)	Any function of n. Generally, f(n) will represent a simpler function than T(n), for example, n^2 rather than $1.5n^2 - 1.5n$
O(f(n))	Order of magnitude. **O**(f(n)) is the set of functions that grow no faster than f(n). We say that T(n) = **O**(f(n)) to indicate that the growth of T(n) is bounded by the growth of f(n)

Comparing Performance

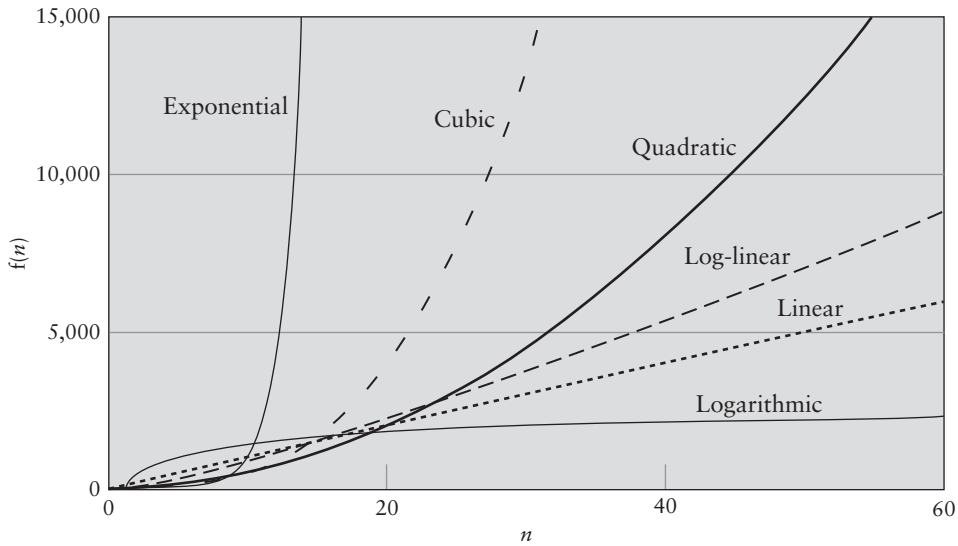
Throughout this text, as we discuss various algorithms, we will discuss how their execution time or storage requirements grow as a function of the problem size using this big-O notation. Several common growth rates will be encountered and are summarized in Table 2.2.

TABLE 2.2
Common Growth Rates

Big-O	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Log-linear
O(n^2)	Quadratic
O(n^3)	Cubic
O(2^n)	Exponential
O(n!)	Factorial

Figure 2.3 shows the growth rate of a logarithmic, a linear, a log-linear, a quadratic, a cubic, and an exponential function by plotting f(n) for a function of each type. Note that for small values of n, the exponential function is smaller than all of the others. As shown, it is not until n reaches 20 that the linear function is smaller than the quadratic. This illustrates two points. For small values of n, the less efficient algorithm may be actually more efficient. If you know that you are going to process only a limited amount of data, the **O**(n^2) algorithm may be much more appropriate than the **O**(n log n) algorithm that has a large constant factor. However, algorithms with exponential growth rates can start out small but very quickly grow to be quite large.

FIGURE 2.3
Different Growth Rates



The raw numbers in Figure 2.3 can be deceiving. Part of the reason is that big-O notation ignores all constants. An algorithm with a logarithmic growth rate **O**(*log n*) may be more complicated to program, so it may actually take more time per data item than an algorithm with a linear growth rate **O**(*n*). For example, at $n = 25$, Figure 2.3 shows that the processing time is approximately 1800 units for an algorithm with a logarithmic growth rate and 2500 units for an algorithm with a linear growth rate. Comparisons of this sort are pretty meaningless. The logarithmic algorithm may actually take more time to execute than the linear algorithm for this relatively small data set. Again, what is important is the growth rate of these two kinds of algorithms, which tells you how the performance of each kind of algorithm changes with *n*.

EXAMPLE 2.7 Let's look at how growth rates change as we double the value of *n* (say, from $n = 50$ to $n = 100$). The results are shown in Table 2.3. The third column gives the ratio of processing times for the two different data sizes. For example, it shows that it will take 2.35 times as long to process 100 numbers as it would to process 50 numbers with an **O**($n \log n$) algorithm.

TABLE 2.3
Effects of Different Growth Rates

O(f(*n*))	f(50)	f(100)	f(100)/f(50)
O(1)	1	1	1
O(log *n*)	5.64	6.64	1.18
O(*n*)	50	100	2
O($n \log n$)	282	664	2.35
O(n^2)	2500	10,000	4
O(n^3)	12,500	100,000	8
O(2^n)	1.126×10^{15}	1.27×10^{30}	1.126×10^{15}
O(*n*!)	3.0×10^{64}	9.3×10^{57}	3.1×10^{93}

Algorithms with Exponential and Factorial Growth Rates

Algorithms with exponential and factorial (even faster) growth rates have an effective practical upper limit on the size of problem they can be used for, even with faster and faster computers. For example, if we have an $O(2^n)$ algorithm that takes an hour for 100 inputs, adding the 101st input will take a second hour, adding 5 more inputs will take 32 hours (more than a day!), and adding 14 inputs will take 16,384 hours, which is almost 2 years!

This relation is the basis for cryptographic algorithms—algorithms that *encrypt* text using a special key to make it unreadable by anyone who intercepts it and does not know the key. Encryption is used to provide security for sensitive data sent over the Internet. Some cryptographic algorithms can be broken in $O(2^n)$ time, where n is the number of bits in the key. A key length of 40 bits is considered breakable by a modern computer, but a key length of 100 (60 bits longer) is not because the key with a length of 100 bits will take approximately a billion–billion (10^{18}) times as long as the 40-bit key to crack.

EXERCISES FOR SECTION 2.1

SELF-CHECK

1. Determine how many times the output statement is executed in each of the following fragments. Indicate whether the algorithm is $O(n)$ or $O(n^2)$.

a.
```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        System.out.println(i + "  " + j);
```

b.
```
for (int i = 0; i < n; i++)
    for (int j = 0; j < 2; j++)
        System.out.println(i + "  " + j);
```

c.
```
for (int i = 0; i < n; i++)
    for (int j = n - 1; j >= i; j--)
        System.out.println(i + "  " + j);
```

d.
```
for (int i = 1; i < n; i++)
    for (int j = 0; j < i; j++)
        if (j % i == 0)
            System.out.println(i + "  " + j);
```

2. For the following $T(n)$, find values of n_0 and c such that cn^3 is larger than $T(n)$ for all n larger than n_0.

$T(n) = n^3 - 5n^2 + 20n - 10$

3. How does the performance grow as n goes from 2000 to 4000 for the following? Answer the same question as n goes from 4000 to 8000. Provide tables similar to Table 2.3.

a. $O(\log n)$

b. $O(n)$

c. $O(n \log n)$

d. $O(n^2)$

e. $O(n^3)$

4. According to the plots in Figure 2.3, what are the processing times at $n = 20$ and at $n = 40$ for each of the growth rates shown?

PROGRAMMING

1. Write a program that compares the values of y1 and y2 in the following expressions for values of n up to 100 in increments of 10. Does the result surprise you?

```
y1 = 100 * n + 10
y2 = 5 * n * n + 2
```