Module 1

Algorithm Analysis

Objectives:

1. Understand why algorithm analysis is useful.
2. Understand Big-Oh, Big-Omega, and Big-Theta.
3. Be familiar with the general categories of growth-rate functions used in determining an algorithm's complexity.
4. Be able to estimate the complexity of an algorithm, based on its code.
5. Know how to compare algorithms based on Big-Oh analysis.

Textbook:

Chapter 4, sections 4.1 - 4.22.

Sections 4.23 - 4.27 are optional; those sections are dependent on the ADT Bag, which is presented in chapters 1-3. A bag is simply a generalized structure that contains a collection of objects, and is often referred to as a collection. We won't cover specific data structures until Module 5.

Lab:

1. Do the Maximum Contiguous Subsequence (MCS) lab for module 1.

Quiz:

Complete the quiz for Module 1.

Assignment:

Complete the assignment for Module 1, posted as a separate content item in the section for Module 1.

1.1 Why analyze algorithm efficiency?

Not all algorithms for accomplishing a particular task are equivalent. Any given task can be accomplished using any of a number of different algorithms. Some of these algorithms may perform better than others. Why bother going to the trouble of analyzing an algorithm's efficiency? If an algorithm works, especially if a great deal of effort was put into writing it, isn't that good enough? Computers are becoming faster as improvements are made in hardware technology. As computers become faster, algorithms—even inefficient ones, will run faster. Multi-processor systems can be employed to solve larger problems that single-processor systems can't handle. So, why do extra work to optimize our algorithms? As it turns out, the reasons vary, depending on the problem and the need.

- Some algorithms will not be able to solve problems beyond a trivial size in a reasonable amount of time, even with the latest hardware advancements. In some cases, this is simply a characteristic of the problem itself, but in other cases, an algorithm may be performing a lot of unnecessary or redundant work. Optimizing an algorithm to avoid this extra work may make the problem manageable.

- Sometimes it is necessary to optimize an algorithm even to gain relatively small improvements in running time. For example, if you are doing a Google search, you may be able to live with a 10-second search time, but not a 20-second search time.

- Even when there are no complaints about an algorithm's performance, it is useful at least in a purist sense to try to find the most efficient algorithm for solving a problem. In some cases, this is relatively easy, while in other cases it is not. But if it can be done, the algorithm can be documented and reused, eliminating the need for further optimization.

- One side effect of faster hardware is that it may encourage software developers to become sloppier with their code. The problem here is that even if a sloppy algorithm will work fine on the latest computer systems, not everyone can afford or have access to the fastest, state-of-the-art computers on the market. Trying to run sloppily-developed code on slower systems can discourage use of the software, and even impact the software's marketability.

1.2 Running Time vs. Space Consumption

Generally speaking, an algorithm's efficiency is most often tied to its running time - the faster an algorithm runs, the more efficient it is. This does not necessarily mean that a given algorithm is bad - in fact, it may be the best algorithm possible for the problem at hand. But if a way can be found to improve the algorithm such that its running time is faster, the improved algorithm will be a better choice when the number of input items is large. When the number of input items is small, it may not make much difference which algorithm is used.

Space consumption is often overlooked when determining an algorithm's efficiency, which may no cause problems on modern systems with gigabytes of RAM. But space consumption should not be ignored. Many end users of a program may have limited RAM, but still need to use the program. Embedded software used in electronic devices must be able to function under very strict space requirements. Space consumption also tends to be important when data must be retrieved or written to relatively slow devices, such as a hard drive or a network share. In these cases even a fast algorithm that requires enormous amounts of space may run very poorly, or even lock up the system.

Running time and space consumption are independent of each other. That is, you can have a very fast algorithm that consumes a lot of space, and you can have a very slow algorithm that uses very little space. For the sake of simplicity, I will often refer to running time in this lecture, but keep in mind that the same principles also apply to space consumption.

1.3 Growth Rate Functions

A growth rate function is a mathematical function used to describe the performance of an algorithm as the number of data items it must process increases. Some growth rate functions can be expressed as polynomials, while others are exponential or logarithmic in nature. The linear function is easiest to understand. If an algorithm requires 1 millisecond to process 10 data items, it should require twice as much time (2 ms) to process twice as many (20) items. When considering space requirements, if an algorithm needs 100 bytes of memory to store 10 items, it should need 200 bytes to store 20 items. Table 1.1 lists the orders of magnitude of the most commonly encountered growth rate functions. The functions are listed from top to bottom in increasing order of complexity. Complexity can refer to increased running time, increased space consumption, or both.

Some important things to note:

1. The table shows only the orders of magnitude of the growth rate functions. Although the functions listed are valid growth rate functions, an infinite number of other functions could be inserted between each entry in the table. For example, an algorithm with a growth rate of $N^{2.5}$ would be slower than an algorithm with a N^2 growth rate, but faster than an algorithm with a N^3 growth rate. A growth rate of log log N is more efficient than a growth rate of log N, but less efficient than a constant growth rate (*c*). As we will see later, when comparing the efficiency of two algorithms, sometimes only the orders of magnitude of the functions are considered, while in other cases the entire functions are considered.

2. Each function can also have an infinite number of variants that still fall within the same family, simply by multiplying the function by a constant, *c*. So, algorithms with growth rates of N, 2N, 0.5N and 10^6N would all be considered linear algorithms.

3. Growth rates can also be expressed by more complicated mathematical functions such as $6N^3 + 45N^2 + 13N + 200$.

4. The logarithmic functions refer to base 2, not base 10, logarithms.

Function	**Common Name**	**Examples**
c	constant	any single, simple statement such as int i = 1;
log N	logarithmic	repeated halving, doubling algorithms (binary search)
$\log_2$ N	logarithmic	
N	linear	a single loop through an entire data set of size N
N log N	N log N	quicksort, divide & conquer algorithms using recursion
N^2	quadratic	linear loop nested within a linear loop (bubble sort, insertion sort)
N^3	cubic	3 nested linear loops (matrix multiplication)
N^k		k nested linear loops (calculating combinations)
2^N	exponential	subset generation, inefficient use of recursion, certain brute force algorithms
N!	factorial	permutation generation

Table 1.1. Commonly encountered orders of growth rate functions.

1.4 Big-Oh, Big-Omega, and Big-Theta

Big-Oh, Big-Omega, and Big-Theta are notations used for expressing the running time or space requirements based on growth rate functions. The important concepts are these:

1. Big-Oh will tell you how badly an algorithm will perform in the worst case, or upper-bound scenario. Regardless of how many data items are being processed, the performance can't get any worse than this. Usually this is what we are most interested in, since poor performance is something we want to avoid. The worst case may happen rarely, but when it does it can cause problems.

2. Big-Omega will tell you how well an algorithm will perform in the best case, or lower-bound scenario. Performance can't get any better than this, no matter how many items you have. This analysis usually isn't of as much interest, since we can never guarantee how frequently the best case scenario will occur.

3. Big-Theta gives you a very precise estimate of how an algorithm will perform. Big-Theta is typically very difficult to calculate, so it isn't as widely used. Big-Theta does not describe average performance. In fact, neither does Big-Oh or Big-Omega. Average performance is often expressed using Big-Oh, but it must be calculated statistically, based on the likelihood of each possible input configuration, or it must be proven mathematically.

1.5 Comparing Growth Rates

When performing a theoretical analysis, we always assume N will be very large. Why? In practice, virtually all algorithms will function acceptably for small values of N, but only efficient algorithms will give acceptable performances for large values of N.

When we talk about the value of N being "large", keep in mind that in practice, "large" is arbitrary and depends very heavily on the environment in which the algorithm is run and the properties of the data being processed. For example, a $O(N^3)$ algorithm may take 1 minute to run on an older computer with a relatively slow CPU, but may run in less than a second on a state-of-the-art machine. A general purpose, $O(N^2)$ sorting algorithm may take less time to sort a list of N integers than it would to sort a list of N character strings.

Given two algorithms that perform the same function, and their Big-Oh running times, you can easily determine which algorithm should be faster for a given input size by simply substituting the input size in place of N. However, you cannot perform a meaningful comparison between two algorithms if the algorithms perform different functions. For example, a $O(N^2)$ algorithm that sorts integers cannot be meaningfully compared to a $O(N^2)$ algorithm that renders graphic images. Also, even though you may be able to conclude mathematically which algorithm should be faster, there is no guarantee this conclusion will hold in actual tests since other factors affect an algorithm's observed running time on a given computer. Some of these factors include the hardware and software configurations of the test computer, and the actual input data (different running times may be observed for different sets of data).

Big-Oh, Big-Omega and Big-Theta analyses do not hold up when N is small. The best way to visualize this is to plot the various growth rate functions and look at how the different curves overlap, as shown in the graphs below. The graphs show comparisons for 8 growth rate functions at four different ranges for N. In these graphs no units are specified for the running times because the important thing to note here is how the curves overlap each other at different values of N.

In a theoretical analysis we are not concerned with concrete numbers for either running time or space consumption; rather, we are concerned with the rate at which the running time or space consumption increases as the value for N increases (i.e. we are interested in the slopes of the functions, not their values at a particular point). The less efficient the algorithm, the steeper the slope.

This is not to say that the actual running time is not important—it is important, it just doesn't matter for comparing algorithms to determine which algorithm is most efficient. Remember, the same algorithm can

have different running times on different computers. Note in Figure 1.4 how dramatically the running time is decreased for O(N) and O(log N), compared to the slower algorithms. It is this difference in running time that makes finding efficient algorithms so important.

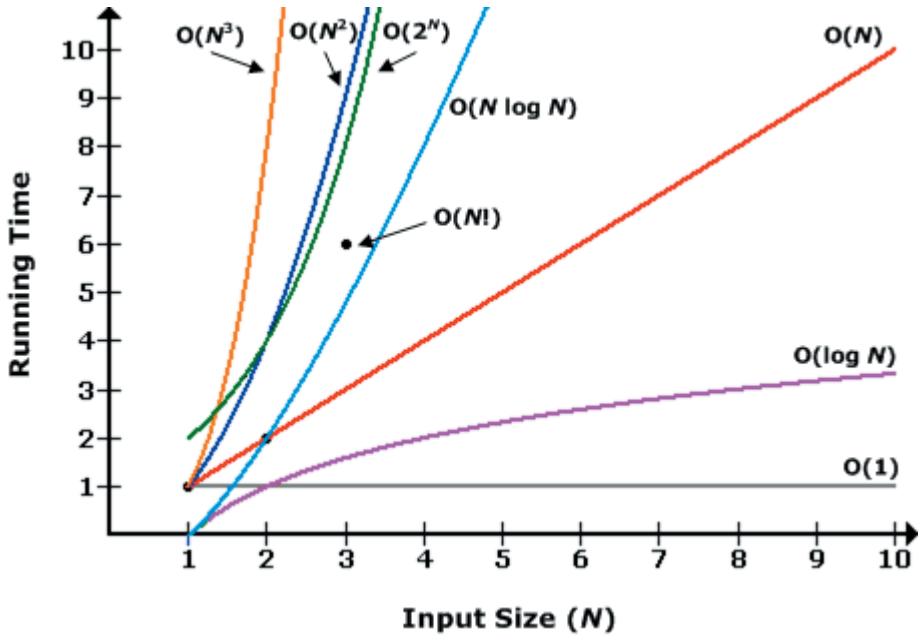

Figure 1.1. Comparative running times for various time functions for 1 <= N <= 10. Note how the different curves overlap when N is very small. O(N!) is shown as a point series rather than a solid line since it is not a continuous function.

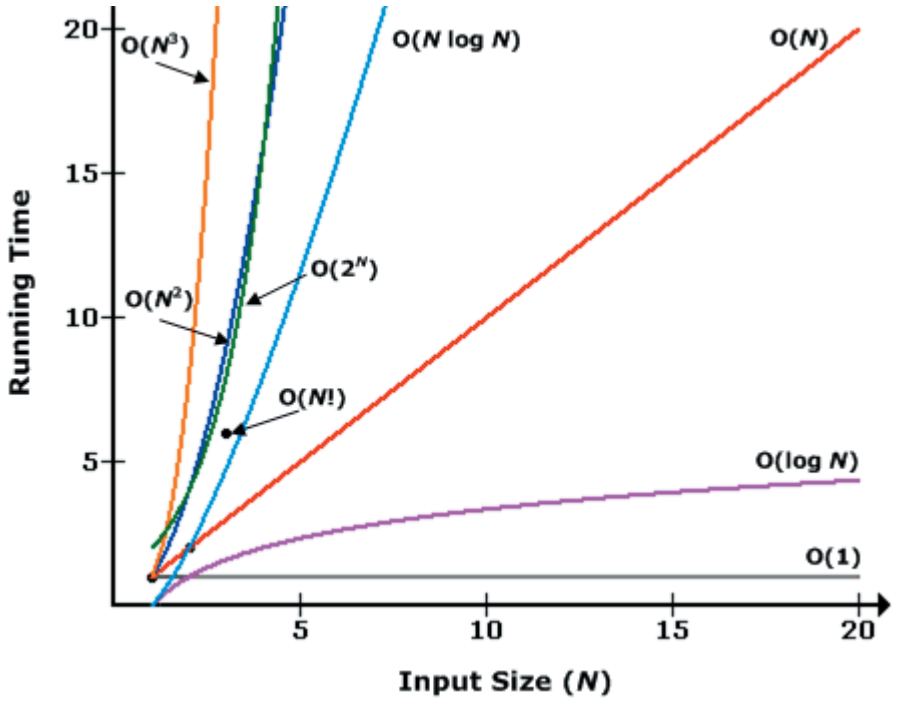

Figure 1.2. Comparative running times for various time functions for 1 <= N <= 20. O(N!) is shown as a point series rather than a solid line since it is not a continuous function.

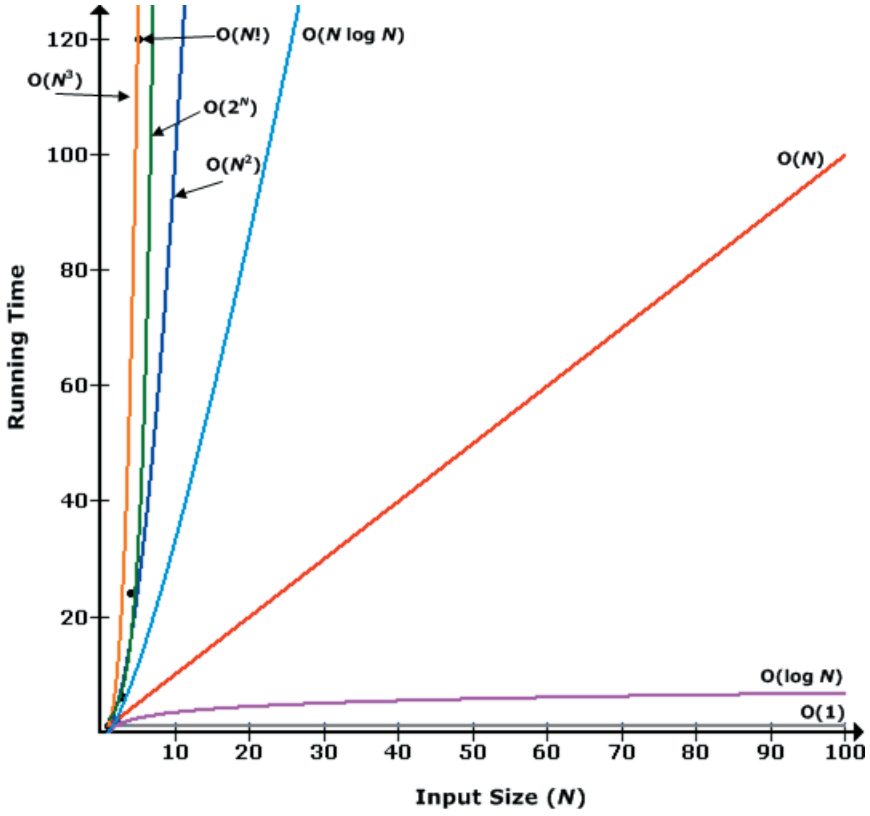

Figure 1.3. Comparative running times for various time functions for 1 <= N <= 100. O(N!) is shown as a point series rather than a solid line since it is not a continuous function.

What does this analysis mean in terms of actually observed running times? Suppose we have algorithms for each of the running times we've discussed. These algorithms must all perform the same function (e.g., calculating the greatest common divisor), they must all be run on the same data sets, and they must all be tested on the same machine, in order for the comparisons to be meaningful. Let's assume it takes 1 nanosecond (ns) to process a data set of size N = 1. We can estimate how many nanoseconds it will take for larger data sets by plugging the desired value of N into the Big-Oh formulae for the algorithms.

The results of such an analysis are tabulated in Figure 1.5. (A similar table is shown in Figure 4-4 in the textbook, but the values the textbook has for O(N!) are **not correct**). Note that for a data set of size N = 1 the running times for the O(log N) and O(N log N) algorithms are listed as zero, since that is the mathematically correct answer. However, it is impossible for an algorithm to process a data set in zero time. In fact, running times for data sets this small are generally meaningless. To give you some idea of how these running times translate into values more familiar to us, consider the following:

- 10^9 ns = 1 second
- 10^{12} ns = 1000 seconds ≈ 16.67 minutes
- 10^{15} ns = 106 seconds ≈ 16,667 minutes ≈ 277.8 hours ≈ 11.6 days
- 10^{18} ns ≈ 109 seconds ≈ 16,666,667 minutes ≈ 277,778 hours ≈ 11, 574 days ≈ 31.7 years

Thus, if you have a data set of say, 1 million items, you really can't afford to use anything less efficient than a quadratic algorithm, which will still take almost 17 minutes to run. Note that in Figure 1.5, some of the running times are listed as "N/A", for "Not Available". I used Java's BigInteger class to calculate the larger running times. The running times listed as "N/A" could not be calculated in a reasonable amount of time.

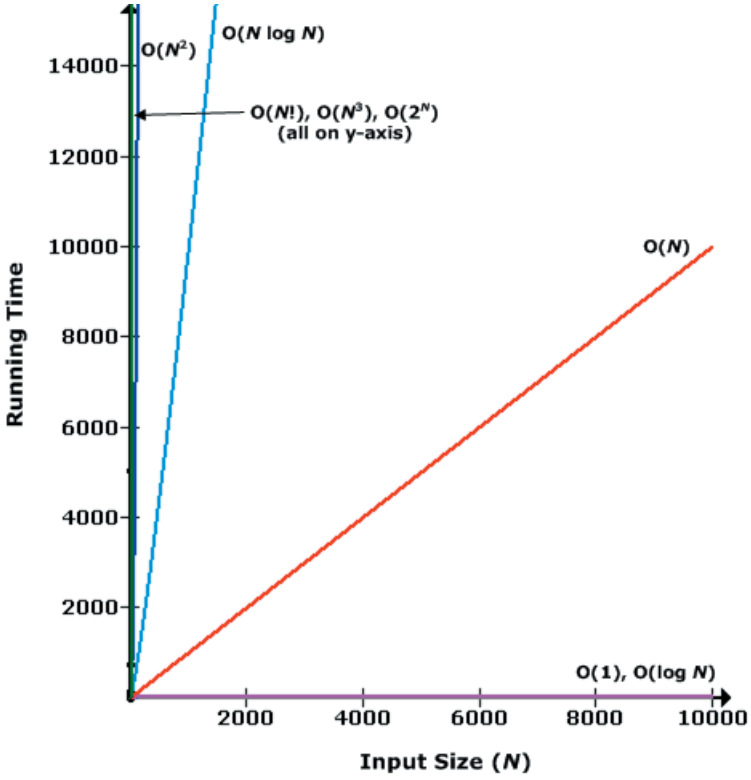

Figure 1.4. Comparative running times for various time functions for 1 <= N <= 10,000.

N	O(log N)	O(N)	O(N log N)	$O(N^2)$	$O(N^3)$	$O(2^N)$	O(N!)
1	0*	1	0*	1	1	2	1
10	3.32	10	33.2	10^2	10^3	1024	3.63×10^6
10^2	6.64	10^2	664	10^4	10^6	1.27×10^{30}	9.33×10^{157}
10^3	9.97	10^3	9966	10^6	10^9	1.07×10^{301}	4.02×10^{2567}
10^4	13.29	10^4	1.33×10^5	10^8	10^{12}	2.00×10^{3010}	2.85×10^{35659}
10^5	16.61	10^5	1.66×10^6	10^{10}	10^{15}	9.99×10^{30102}	2.82×10^{456573}
10^6	19.93	10^6	1.99×10^7	10^{12}	10^{18}	9.90×10^{301029}	N/A
10^7	23.25	10^7	2.33×10^8	10^{14}	10^{21}	N/A	N/A
10^8	26.58	10^8	2.66×10^9	10^{16}	10^{24}	N/A	N/A
10^9	29.90	10^9	2.99×10^{10}	10^{18}	10^{27}	N/A	N/A

Figure 1.5. Actual running times required to process data sets of different sizes of N using algorithms having different Big-Oh values. The times listed in the table are in nanoseconds (ns). *The value of 0 shown for the O(log N) and O(N log N) algorithms when N = 1 are mathematically correct, but are not meaningful in this analysis.

1.6 Precision of Big-Oh Estimates

Big-Oh estimates, like decimal numbers, can be expressed to varying degrees of precision. Suppose we have an algorithm with a quadratic Big-Oh value of $O(3N^2 + 4N + 7)$. The most precise way to express the Big-Oh value for this algorithm would be to use the entire function. We may, however, only be interested in the general order of the algorithm's efficiency. In this case, we would eliminate all but the highest order term, as well as any coefficients of this term. For the above algorithm, we would reduce the Big-Oh value to $O(N^2)$. Both expressions are correct, they just differ in precision. Conversationally, it is easier to talk about a O(N2) algorithm than it is to talk about a $O(3N^2 + 4N + 7)$ algorithm. The choice of precision is usually only important when comparing two algorithms that perform the same task. If we have a second algorithm that accomplishes the same task as our quadratic algorithm, whose Big-Oh value is $O(N^3 + 5N + 2)$, we would most likely use the lower precision and state the second, O(N3) algorithm, will probably be less efficient than the first, $O(N^2)$ algorithm. On the other hand, if we had two quadratic algorithms, we would need to use more precise Big-Oh values in the comparison, since we can't determine which algorithm is more efficient if we express both their Big-Oh values as $O(N^2)$.

1.7 Determining the Big-Oh Value of an Algorithm's Source Code

We have talked about a growth rate function having the form of a mathematical function, but how do we actually go about determining what that mathematical function is? One way to look at this with high precision is to consider the algorithm's Big-Oh value as being represented by the number of times each statement in the algorithm is executed. The Big-Oh value is given as the sum of all the statement executions in the algorithm, expressed in terms of N. Basically, we are just counting the total number of statement executions. This will give us a reasonably precise estimate that serves quite well when comparing the complexities of two algorithms that perform the same task.

For Big-Oh, we would look at the worst-case scenario, so if we are using the total number of statement executions as our guide, we need to consider the case where every statement in the algorithm is executed the maximum number of times. Finding the worst case scenario can be ridiculously easy, or confoundingly hard, depending on the algorithm. Once we know what the worst case scenario is, we need to determine the total number of statement executions. Often, this can be done by simply looking at the code and doing some bookkeeping. However, nested loops, recursion, and compound statements can make determining an algorithm's Big-Oh value quite complicated. For algorithms that are very complex, we can instrument the code to count the number of statement executions for us. "Instrumenting" code refers to the adding of statements that perform bookkeeping tasks or taking measurements such as elapsed time, so that information about an algorithm can be discovered as the algorithm's code is executed.

Counting the total number of executions can be tedious, and for very complex code it can be very difficult. Compound programming statements, such as for loops, conditional statements containing more than one condition, and statements that use the tertiary operator (`? :`), must be expanded so that the number of executions of each component statement can be determined separately. Statements containing calls to other methods must have the method calls replaced with the Big-Oh values of the methods being called. If we just need to know the "ballpark" value for an algorithm's Big-Oh value, we can use some tricks to make the Big-Oh determination easier. These tricks will not give us a precise value, but they will generally give us the order of magnitude, or "ballpark", figure we're looking for. The most commonly employed tricks are:

1. Simple statements, such as assignment statements, are considered to be O(1). Note that this assumes all statements are equal in terms of running time; in reality, they are not. Some operations take slightly more time than other operations.

2. A single loop that iterates N times, or a variable number of times, is considered to be O(N).

3. A single loop that executes anywhere between 1 and N times, where the loop's termination is controlled by the value of some variable, is considered to be O(N).

4. If the loop always iterates a defined number of times, such as 10, it is considered to have a numeric Big-Oh value equal to the number of loop iterations (in this case it would be O(10).

5. For a statement inside a loop, multiply the Big-Oh value for the statement by the Big-Oh value for the loop. For a loop that is nested within another loop, multiply the Big-Oh values for each loop. So, a O(N) loop that is nested within a O(N) loop would have a Big-Oh value of $O(N^2)$.

6. If two loops are consecutive (i.e., one loop completes before the next one begins), add the Big-Oh values for the two loops instead of multiplying them. So, two consecutive O(N) loops would have a Big-Oh value of O(N) + O(N), which is O(2N), which simplifies to O(N), since constants are disregarded.

7. For statements that contain calls to other methods, the method calls must still be replaced with the Big-Oh values of the methods they call.

8. At the end, disregard all but the dominant term, and disregard any coefficient of that term as well as any constant values that may be associated with that term. The dominant term is the term that contributes the most to a function's value as the value of N increases. The dominant term also represents the statement(s) that are executed the greatest number of times. Basically, what you are doing at this step is boiling down all the terms to one of the values listed in Table 1.1, ordering the terms from slowest to fastest, and taking the slowest term.

1.8 Counting Statement Executions in Loops

Loops are commonly encountered in many programs, and can contribute significantly to an algorithm's running time. Counting the number of statement executions in a loop can be rather tricky, especially if the loop is nested. As I'm sure you remember from CS I or CS II, all well-formed loops have three components:

1. a loop variable initializer

2. a termination condition

3. a loop variable updater

Consider the typical for loop: `for (int i = 0; i < n; i++)`, here written as a compound statement where all three components are contained on the same line of code. How do you determine how many times this loop executes? Using one of the shortcuts mentioned in a previous section, we could say the loop is O(N), since the loop iterates N times. But if we really want to look at the total number of statement executions, we need to consider each of the individual statements separately. To help us visualize this we can rewrite the loop as follows:

```
int i = 0;

for ( ; ; )
{
      if (i < n)
      {
            // statements go here
      }
      else
      {
            break;
      }

      i++;
}
```

Rewritten this way, it is easy to see that the loop initializer (`int i = 0;`) is O(1). It is tempting to state that all the statements inside the loop will execute N times, but that wouldn't be quite accurate. As it turns

out, the termination condition actually executes N + 1 times, with the "extra" execution being the one that determines the value of the loop variable has gone past the termination condition. The loop variable update statement (i++;) does execute N times, however. The Nth update will be the one that pushes the value loop variable to the termination condition, at which point the loop will terminate, bypassing the loop variable updater statement. The break statement is not counted, since it isn't present in the original, compound form of the loop, and since in the compiled code, when the termination condition is reached execution simply proceeds to the next statement after the loop.

This leaves us with the for statement, which is still part of the source code, although it appears to do nothing now that it is empty. It does, however, still contribute to the complexity of the algorithm. Remember that any loop must be capable of jumping back to the beginning of the loop to begin the next iteration. This jump instruction is not explicitly stated in the source code, but it does exist in the compiled code. Analysis of compiled code can be murky, since different compilers can generate different compiled code for the same source. Additionally, a compiler that does optimization may generate compiled code that does not exactly mirror the source code as it is written. As a result, the complexity of compiled code will not necessarily be the exactly same as the complexity for the source code from which it was generated, although the order of magnitude will be the same. We won't discuss the complexity of compiled code in this course, but for the sake of clarity we will consider the jump instruction that redirects execution from the end of the loop back to the beginning of the loop. In general, the number of times this jump instruction will be executed will be the same as the number of times the loop termination condition is executed minus the "extra" executions of the loop termination condition, since when the loop terminates the jump instruction is bypassed. For the code above, this would be N times. We can't reliably determine the number of jumps based on the loop variable update statement, since some loops may modify the loop variable in wyas other than a simple increment or decrement. For that matter, the presence of intentional break statements (but not those added only to force dissected loops to functino correctly) can bypass the jump instruction and reduce the number of times the jump instruction is executed by 1.

The total number of statement executions for the above loop, counting the for statement as a placeholder for the loop jump instruction, is: 1 + N + N + 1 + N, which simplifies to 3N + 2. The Big-Oh value of the loop, therefore, is O(3N + 2), or if we're simply interested in the order of magnitude, O(N).

1.9 Counting Statement Executions in Nested Loops

It's a trivial matter to count the number of statement executions that occur inside a single loop. For example, in the statement, `for (int i = 0; i < N; i++)`, each statement inside the loop will be executed N times, from `i = 0` to `i = N - 1`. Now consider the following nested loop structure:

```
for (int i = 0; i < N; i++)
{
	for (int j = 0; j < N; j++)
	{
		// statements here
	}
}
```

Both loops iterate from 0 to N - 1; i.e., each loop is O(N). If we apply the rule of multiplying the Big-Oh values of nested loops, we find that overall the nested loops should be O(N * N), or $O(N^2)$. If we placed inside the innermost loop a third loop that also iterated from 0 to N - 1, the Big-Oh value of the three nested loops would be O(N * N * N), or $O(N^3)$.

So far, so good. But what if we have the following nested loop structure:

```
for (int i = 0; i < N; i++)
{
	for (int j = i; j < N; j++)
	{
		// statements here
	}
}
```

We can see that the outer loop still iterates from 0 to N - 1, but the inner loop iterates a decreasing number of times as the value of i increases. The number of times the inner loop iterates follows the progression:

N, N - 1, N - 2, ..., 3, 2, 1.

If we reverse the order of the terms we get:

1, 2, 3, ..., N - 2, N - 1, N.

If we want to get the total number of times the statements in the inner loop are executed, we sum the values of the arithmetic progression; i.e., SUM(1, 2, 3, ..., N - 2, N - 1, N). As it turns out, there is a simple formula we can use to get this value without having to manually sum the values of the progression. The general formula is

$$S_n = a_1 + a_2 + a_3 + ... + a_n$$

$$= n(a_1 + a_n)/2$$

$$= n[2a_1 + d(n - 1)]/2$$

where d is the difference between consecutive numbers in the progression. In our case, the arithmetic progression is 1 + 2 + 3 + ..., so the above formula reduces to $n(n + 1)/2$. You may have seen a proof of this in a previous math course. If you haven't seen it, or need a refresher, do an internet search for "triangular number" for a good way to visualize the problem. The actual proof can be performed as a proof by induction. If we multiply out the formula we get: $(n^2 + n)/2$. The order of this formula is quadratic, meaning the second nested loop example has a Big-Oh value of $O(N^2)$, which is consistent with what we would expect from two nested loops.

Now, let's add another loop to get a more complicated nested loop structure:

```
for (int i = 0; i < N; i++)
{
	for (int j = i; j < N; j++)
	{
		for (int k = j; k < N; k++)
		{
			// statements here
		}
	}
}
```

How do we go about determining how many times the statements of the innermost loop are executed this time? For two loops, the sum of the arithmetic progression happens to be a triangular number. Adding a third loop amounts to adding a third dimension to the triangle, to form a tetrahedron, or pyramid. The formula for obtaining the sum of the arithmetic progression of a tetrahedral number is given by $n(n + 1)(n + 2)/6$. Multiplied out, the formula becomes $(n^3 + 3n^2 + 2n)/6$, which is cubic in order—again, consistent with what we would expect from three nested loops.

If we have more nested loops, determining the sum of the progression (generally referred to as a **figurate number**), is hard to visualize since it's difficult for us to visualize objects beyond three dimensions. However, another easy way to determine the sum is make use of the fact that each iteration of the entire nested loop structure is a tuple of the possible values for the loop variables. For example, the tuple <0, 3, 5> represents the state of the loops when `i = 0`, `j = 3`, and `k = 5`. Determining the total number of statement executions inside the innermost loop can therefore be determined by calculating the number of combinations of tuples of loop variable values that are possible. The formula for this is:

$$C(N - 1 + r, r)$$

where N is the problem size, and r is the number of nested loops. The number of combinations is solved as follows:

$C(N - 1 + r, r) = (N - 1 + r)! /[r! * ((N - 1 + r) - r)!]$

So, if we have the triple nested loop structure shown above, when N = 10, the total number of times the statements in the innermost loop will be executed will be:

(10 - 1 + 3)!/[3! * ((10 - 1 + 3) - 3)!] = 12!/[3! * 9!] = 220.

We can verify this result by checking against the formula for calculating the tetrahedral number:

$n(n + 1)(n + 2)/6$ = 10(10 + 1)(10 + 2)/6 = 220.

1.10 Example: The Maximum Contiguous Subsequence (MCS) Problem

Now that we know how to determine the number of statement executions when nested loops are involved, let's determine the Big-Oh value of an actual algorithm's code, as well as understand how an inefficient algorithm can be optimized. For this example, we will look at the Maximum Contiguous Subsequence (MCS) Problem: Given a sequence of integers, $a_1, a_2, a_3, \ldots, a_n$, determine the subsequence of contiguous integers, $a_i, a_{i+1}, a_{i+2}, \ldots, a_k$, that has the maximum sum of all possible subsequences. The following rules apply:

1. The integers may be positive, negative, or zero.
2. The integers in the sequence do not have to be in any particular order.
3. If all the integers in a subsequence are negative, the sum is zero.

For example, given the sequence, -2, -3, 4, -1, 4, 2, 3, the maximum contiguous subsequence is 4, -1, 4, 2, 3, which has a sum of 12.

In the sections that follow, we will look at three different algorithms that solve this problem, in increasing order of efficiency. Before analyzing the code for these algorithms, we must determine what the worst case scenario is for this problem. Some thought suggests that the worst case scenario should occur when the MCS is equal to the entire input sequence, since in that scenario all the values in the input sequence must be summed.

1.11 MCS Algorithm #1: A Brute-force, $O(N^3)$ Algorithm

A **brute-force** algorithm operates by analyzing every possible state of a problem in order to find a solution. Brute-force algorithms are considered **complete** because every possible combination is analyzed; no stone is left unturned. Brute-force algorithms are often relatively easy to write since they use the most straightforward approach, but they also tend to be woefully inefficient. For the MCS problem, a brute-force algorithm must examine all possible subsequences in the sequence of integers. The code for this algorithm is shown in Figure 1.6. In this code, the value of *N* is represented by `a.length`.

The outermost loop defines the starting position of the subsequence currently being considered. The middle loop defines the ending position of the subsequence. The starting position can be any integer in the sequence, hence the outermost loop goes from the first integer to the last integer. The ending position can be any position to the right of the starting position, or it can be the same as the starting position since we can have a subsequence that consists of only one integer. So it's easy to see how these two loops work. Once we have defined the starting and ending positions for a given subsequence, we need to compute the sum of the integers in that subsequence. The most straightforward way to do this is to use a third loop that goes from the starting position to the ending position.

```
public int mcsCubic (int[] a)
{
    int maxSum = 0;
    int seqStart = -1;
    int seqEnd = -1;

    for (int i = 0; i < a.length; i++)
    {
        for (int j = i; j < a.length; j++)
        {
            int thisSum = 0;

            for (int k = i; k <= j; k++)
            {
                thisSum += a[k];
            }

            if (thisSum > maxSum)
            {
                maxSum = thisSum;
                seqStart = i;
                seqEnd = j;
            }
        }
    }

return maxSum;
}
```

Figure 1.6. A brute-force algorithm for solving the MCS problem. Comments have been omitted for greater clarity of the code structure.

Although the algorithm works, it won't be very efficient when N is large. To estimate the Big-Oh value for this algorithm using the shortcut rules listed in section 1.7, we can see that the running time is going to be determined primarily from the three nested loops. The outermost loop executes *N* times in the worst case scenario, so its Big-Oh value is O(*N*). The middle loop is executed a variable number of times, depending on the value of i, so by Rule #4 we say the middle loop is also O(*N*). The innermost loop is also O(*N*) by the same reasoning. All the other statements are O(1). To get the running time for the nested loop structure, we multiply the Big-Oh values for the individual loops. This gives us O(*N*) * O(*N*) * O(*N*), or $O(N^3)$. Thus, according to our quick estimation method, the brute-force algorithm is cubic.

By comparison, let's determine the brute-force algorithm's running time by estimating the number of times each statement is executed. The results are shown in Figure 1.7, with the compound `for` loops dissected.

There are several important things to note in Figure 1.7. First, the break statements are not counted since they do not appear in the original algorithm; they are merely artifacts required to make the dissected loops function properly. Second, each for statement have been counted as the jump instruction from the end of its loop back to the beginning, as discussed in section 1.8 above. Third, the number of times the loop conditional statements are executed is not given precisely by the formulae used for summing the arithmetic progressions. This is because the loop condition is actually executed one additional time for each complete traversal of the loop. The extra statement execution is the one that actually causes the loop to be terminated. To see this for the outermost loop, let's assume N = 10. For each value of `i`, the condition is evaluated, so for the progression 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, the termination condition is evaluated 10, or N, times. But since 9 < 10, the loop will actually try to begin an 11th iteration, with `i = 10`, since `i` is incremented at the end of the loop. So, the termination condition must be evaluated one extra time in order for execution to break out of the loop. So, the outermost loop termination condition will be executed one extra time for a total of N + 1 times. The termination condition for the middle loop will be executed an extra time for each iteration of the outer loop, meaning the middle loop's termination condition will be executed an extra N times. By now, the pattern should be clear, so it should be easy to see why the innermost loop's termination condition will be executed an extra N(N + 1)/2 times.

Finally, the number of times the maximum sum encountered so far (`maxSum`) must be swapped with the current sum (`thisSum`) may not be intuitively obvious. If we know that the conditional statement (`if (thisSum > maxSum)`) will be executed N(N + 1)/2 times in the worst case, we might reason that the

	# of times executed (general)	# of times executed (N = 10)
`public int mcsCubic (int[] a)`		
`{`		
`int maxSum = 0;`	1	1
`int seqStart = -1;`	1	1
`int seqEnd = -1;`	1	1
`int i = 0;`	1	1
`for ( ; ; ) // as jump statement`	N	10
`{`		
`if (i < a.length)`	N + 1	11
`{`		
`int j = i;`	N	10
`for ( ; ; ) // as jump statement`	N(N + 1)/2	55
`{`		
`if (j < a.length)`	N(N + 1)/2 + N	65
`{`		
`int thisSum = 0;`	N(N + 1)/2	55
`int k = i;`	N(N + 1)/2	55
`for ( ; ; ) // as jump statement`	N(N + 1)(N + 2)/6	220
`{`		
`if (k <= j)`	N(N + 1)(N + 2)/6 + N(N + 1)/2	275
`{`		
`thisSum += a[k];`	N(N + 1)(N + 2)/6	220
`}`		
`else`		
`{`		
`break;`		
`}`		
`k++;`	N(N + 1)(N + 2)/6	220
`}`		
`if (thisSum > maxSum)`	N(N + 1)/2	55
`{`		
`maxSum = thisSum;`	N	10
`seqStart = i;`	N	10
`seqEnd = j;`	N	10
`}`		
`}`		
`else`		
`{`		
`break;`		
`}`		
`j++;`	N(N + 1)/2	55
`}`		
`}`		
`i++;`	N	10
`}`		
`return maxSum;`	1	1
`}`		
Big-Oh:	**$(4N^3 + 33N^2 + 77N + 36)/6$**	
Total # of statement executions:	**1351**	

Figure 1.7. Estimation of Big-Oh value for the brute-force, cubic MCS algorithm by estimating the total number of statement executions. The compound for loops have been dissected into their separate statements.

worst case will also imply that the condition always evaluates to true, thus all the statements within the conditional statement will also be executed N(N + 1)/2 times. As it turns out, this is not possible for this particular algorithm. We will discuss why later, but for now let's look at the problem with respect only to the input sequence. If we know that the worst case scenario means the MCS is the entire input sequence, then to get the sum of the sequence we simply need to add the integers in the sequence. Since there are N integers in the sequence, there must be a maximum of N addition operations, and thus a maximum of N swaps between maxSum and
thisSum.

In summary, the complexity of the cubic MCS algorithm, as written here, is $O((4N^3 + 33N^2 + 77N + 36)/6)$, written as a single fraction over a common demoninator. The highest-order term is N^3, confirming the algorithm is indeed cubic. When the value of N becomes large enough, the other terms will only negligibly affect the value of the formula, so if we just want to talk about the order of the algorithm, we would say it is $O(N^3)$. However, if we want to know the precise formula, or compare this algorithm accurately with other MCS algorithms—in particular other cubic MCS algorithms—we would use all the terms.

1.12 MCS Algorithm #2: An Improved, $O(N^2)$ Algorithm

The cubic algorithm demonstrates how nested loops can result in an inefficient algorithm. Can we do anything to improve the algorithm's performance? If we look more closely at the brute-force algorithm we can see that the sole purpose of the innermost loop is to perform the actual summation of each subsequence. This loop is inefficient because the sums of some sub-subsequences are calculated multiple times. We can easily remove the innermost loop by maintaining a running sum in the preceding loop. Removal of one of the loops leaves us with only 2 nested loops, so the resulting algorithm should be $O(N^2)$ according to our shortcut estimation method. Figure 1.8 shows the improved algorithm, and Figure 1.9 shows a more accurate determination of its Big-Oh value based on the total number of statement executions. The results confirm that the algorithm is indeed quadratic, and that it requires less than a third of the statement executions of the cubic algorithm.

Note that this is still a brute-force algorithm, since it still analyzes all possible subsequences, but it is an order of magnitude more efficient than the first algorithm we discussed.

```
public int mcsQuadratic (int[] a)
{
    int maxSum = 0;
    int seqStart = -1;
    int seqEnd = -1;

    for (int i = 0; i < a.length; i++)
    {
        int thisSum = 0;

        for (int j = i; j < a.length; j++)
        {
            thisSum += a[j];

            if (thisSum > maxSum)
            {
                maxSum = thisSum;
                seqStart = i;
                seqEnd = j;
            }
        }
    }

    return maxSum;
}
```

Figure 1.8. An improved, brute-force, quadratic algorithm for solving the MCS problem that uses a running sum to reduce the number of loops needed to 2.

	# of times executed (general)	# of times executed (N = 10)
`public int mcsQuadratic (int[] a)`		
`{`		
`int maxSum = 0;`	1	1
`int seqStart = -1;`	1	1
`int seqEnd = -1;`	1	1
`int i = 0;`	1	1
`for ( ; ; ) // as jump statement`	N	10
`{`		
`if (i < a.length)`	N + 1	11
`{`		
`int thisSum = 0;`	N	10
`int j = i;`	N	10
`for ( ; ; ) // as jump statement`	N(N + 1)/2	55
`{`		
`if (j < a.length)`	N(N + 1)/2 + N	65
`{`		
`thisSum += a[j];`	N(N + 1)/2	55
`if (thisSum > maxSum)`	N(N + 1)/2	55
`{`		
`maxSum = thisSum;`	N	10
`seqStart = i;`	N	10
`seqEnd = j;`	N	10
`}`		
`}`		
`else`		
`{`		
`break;`		
`}`		
`j++;`	N(N + 1)/2	55
`}`		
`i++;`	N	10
`}`		
`else`		
`{`		
`break;`		
`}`		
`}`		
`return maxSum;`	1	1
`}`		
Big-Oh:	**$(5N^2 + 23N + 12)/2$**	
Sum of statement executions:	**371**	

Figure 1.9. Analysis of the quadratic MCS algorithm.

1.13 MCS Algorithm #3: An Even Better, O(N), Algorithm

We found that the quadratic algorithm is faster than the cubic algorithm, but even the quadratic algorithm can be cumbersome when N is very large. Can we remove another loop to get an algorithm that is linear? The answer is yes, but it won't be quite as easy as going from the cubic algorithm to the quadratic algorithm. To get linear performance we will need to avoid using the brute-force method, which means we need to be able to find the maximum contiguous subsequence without analyzing all possible sub-sequences. This might sound impossible, but it's not. There are mathematical theorems to explain how

this is possible, but here is a non-mathematical explanation:

If a negative sum is obtained when `j` is incremented and the next number is added to the current subsequence, all the integers from `i` to `j`, inclusive, can be eliminated from consideration:

- If the addition of `a[j]` produces a negative sum, the maximum contiguous subsequence sum will only be reduced by including `a[j]`, and thus any integers to the left of `j`.
- Since it's necessary to have a contiguous subsequence, if the starting point is before `a[j]`, and the ending point is after `a[j]`, `a[j]` must be included in the subsequence.
- So, the maximum contiguous subsequence must either begin and end before `a[j]`, or begin and end after `a[j]`; it cannot span or include `a[j]`.
- Therefore, the value of i can be advanced to position `j + 1`, eliminating all the remaining subsequences that include `a[j]`.

In short, we are simply eliminating from consideration any subsequence that we know cannot possibly be a contender for the maximum contiguous subsequence. The code for the linear algorithm is shown below. If you have a hard time understanding how this is done, you're not alone, it's a complicated algorithm that shows it is sometimes necessary to employ clever tricks and/or write complex code in order to improve an algorithm's efficiency. Figure 1.10 shows the code for the O(N) MCS algorithm. Figure 1.11 shows the Big-Oh analysis, which confirms the algorithm is both linear and more efficient than the quadratic algorithm. Note that in the worst case scenario, the value of `thisSum` will always be greater than `maxSum`, since every integer in the input sequence increases the sum; thus, the else clause that checks for a negative sum will never be executed.

```
public int mcsLinear (int[] a)
{
    int maxSum = 0;
    int thisSum = 0;
    int seqStart = -1;
    int seqEnd = -1;

    for (int i = 0, j = 0; j < a.length; j++)
    {
        thisSum += a[j];

        if (thisSum > maxSum)
        {
            maxSum = thisSum;
            seqStart = i;
            seqEnd = j;
        }
        else if (thisSum < 0)
        {
            i = j + 1;
            thisSum = 0;
        }
    }

    return maxSum;
}
```

Figure 1.10. A further improved, non-brute force, linear algorithm for solving the MCS problem that makes use of the running sum technique, and eliminates analysis of subsequences that cannot increase the maximum sum.

	# of times executed (general)	# of times executed (N = 10)
`public int mcsLinear (int[] a)`		
`{`		
`    int maxSum = 0;`	1	1
`    int thisSum = 0;`	1	1
`    int seqStart = -1;`	1	1
`    int seqEnd = -1;`	1	1
`    int i = 0;`	1	1
`    int j = 0;`	1	1
`    for ( ; ; ) // as jump statement`	N	10
`    {`		
`        if (j < a.length)`	N + 1	11
`        {`		
`            thisSum += a[j];`	N	10
`            if (thisSum > maxSum)`	N	10
`            {`		
`                maxSum = thisSum;`	N	10
`                seqStart = i;`	N	10
`                seqEnd = j;`	N	10
`            }`		
`            else if (thisSum < 0)`	0	0
`            {`		
`                i = j + 1;`	0	0
`                thisSum = 0;`	0	0
`            }`		
`        }`		
`        else`		
`        {`		
`            break;`	0	0
`        }`		
`        j++;`	N	10
`    }`		
`    return maxSum;`	1	1
`}`		
Big-Oh:	**8N + 8**	
Sum of statement executions:	**88**	

Figure 1.11. Analysis of the linear MCS algorithm.

1.14 Comparison of the MCS Algorithms

Although we saw significant reductions in the total number of statement executions as we went from the cubic MCS algorithm to the linear algorithm, a problem size of 10 is quite small. Remember that with Big-Oh we're not only interested in the worst case scenario, we're interested in the worst case scenario when N is large. Figure 1.12 shows a comparison of the performances of the cubic, quadratic, and linear MCS algorithms, in terms of total number of statement executions, when N = 10,000. If these values still don't convince you of the improvement in performance as the algorithm is optimized, try running the code for an input sequence of 10,000 positive integers. If your computer is fast enough that you don't tire of waiting for the cubic algorithm to run to completion, increase the value of N to 100,000 and try it again.

Algorithm	Big-Oh	Total # of Statement Executions (N = 10,000)
Cubic	$(4N^3 + 33N^2 + 77N + 36)/6$	4,003,300,770,036
Quadratic	$(5N^2 + 23N + 12)/2$	500,230,012
Linear	8N + 8	80,008

Figure 1.12. A comparison of the cubic, quadratic, and linear MCS algorithms, in terms of total number of statement executions, for N = 10,000.

1.15 Limitations of Big-Oh Analysis

Big-Oh analysis, while very useful, has some limitations. When performing Big-Oh analysis on an algorithm, keep the following factors in mind:

Problem Size

When N is small Big-Oh analysis breaks down, because for small values of N the term with the highest order of magnitude may not be dominant. As an example, consider the cubic MCS algorithm. When N = 10, the value of the quadratic term $12N^2$ is slightly greater than the value of the cubic term N^3 (1200 > 1000). At N = 12 the cubic term shares dominance with the quadratic term (1200 = 1200). At N = 1, the linear term is dominant over both the cubic and quadratic terms (19 > 1 and 19 > 12, respectively).

The above analysis leads to another question: is there a scenario where the cubic MCS algorithm is actually more efficient than either the quadratic or linear MCS algorithms? The answer in this case is no, although for the trivial problem where N = 1, the algorithms have very comparable running times. For some other algorithms, though, it is possible to have a cubic version of the algorithm perform better than a quadratic or linear version, if N is small enough.

Effect of Certain Types of Operations

Operations such as accessing a hard drive or network share can be very expensive in terms of running time, due to the time required for one hardware component to interface with another. These operations may not involve many statement executions, though. For example, reading a single byte of data from a hard drive can be accomplished in a single statement in code. This statement might be assigned a Big-Oh value of O(1), since it is a simple statement. However, reading a byte of data from a hard drive is much slower than reading a byte of data from main memory, another simple operation that might also be given a Big-Oh value of O(1). In this case, representing the read from a hard drive as a O(1) statement will underestimate the true cost of that operation. The discrepancy in cost can be accounted for by adjusting the values of statements performing these operations. For example, a simple statement that reads a single byte of data from main memory might be represented O(1), while a simple statement that reads a single byte of data from a network connection might be represented O(1000).

The Worst-Case Scenario May Be Very Rare

Big-Oh analysis describes how an algorithm will perform when the worst-case scenario is encountered. Usually the worst-case scenario can be expected to occur at a reasonable frequency, but if it can be shown the worst-case scenario will occur rarely or not at all, the actual performance of an algorithm will be better

than its Big-Oh value would indicate.

Worst-case scenarios are often associated with loops of size N, since such loops usually indicate examining and/or processing every item in a data set. However, if it can be shown that a loop of size N will rarely proceed past the first few items, that particular loop will lead to an overestimation of the algorithm's performance.