Algorithm Analysis

1. Introduction

Usually, a given problem can be solved with different algorithms which differ in efficiency. These differences may be negligible for a small number of data items, but may grow proportionally with the amount of data. To compare the efficiency of algorithms, a measure of the degree of difficulty of an algorithm, called **computational complexity** was developed by Juris Hartmanis and Richard E. Stearns.

Computational complexity provides us with an idea of the effort needed to apply an algorithm to solve a given problem or how costly it is. This cost can be measured in several ways and the particular context determines its meaning.

If a particular program is slow, is it badly implemented or is it solving a hard problem? Questions like these ask us to consider the difficulty of a problem, or the relative efficiency of two or more approaches to solving a problem.

This section introduces the motivation, basic notation, and fundamental techniques of algorithm analysis. It focuses on a methodology known as **asymptotic algorithm analysis**, or simply **asymptotic analysis**. **Asymptotic analysis** attempts to estimate the resource consumption of an algorithm. It allows us to compare the relative costs of two or more algorithms for solving the same problem. Asymptotic analysis also gives algorithm designers a tool for estimating whether a proposed solution is likely to meet the resource constraints for a problem before they implement an actual program.
After reading this material, you should understand

- the concept of a growth rate, the rate at which the cost of an algorithm grows as the size of its input grows;
- the concept of upper and lower bounds for a growth rate, and how to estimate
- these bounds for a simple program, algorithm, or problem; and
- the difference between the cost of an algorithm (or program) and the cost of a problem.

How do you compare two algorithms for solving some problem in terms of efficiency? One way is to implement both algorithms as computer programs and then run them on a suitable range of inputs, measuring how much of the resources in question each program uses. This approach is often unsatisfactory for four reasons.

First, there is the effort involved in programming and testing two algorithms when at best you want to keep only one. Second, when empirically comparing two algorithms there is always the chance that one of the programs was “better written” than the other, and that the relative qualities of the underlying algorithms are not truly represented by their implementations. This is especially likely to occur when the programmer has a bias regarding the algorithms. Third, the choice of empirical test cases might unfairly favour one algorithm. Fourth, you could find that even the better of the two algorithms does not fall within your resource budget. In that case you must begin the entire process again with yet another program implementing a new algorithm. But, how would you know if any algorithm can meet the resource budget? Perhaps the problem is simply too difficult for any implementation to be within budget.

These problems can often be avoided by using asymptotic analysis. Asymptotic analysis measures the efficiency of an algorithm, or its implementation as a program, as the input size becomes large. It is actually an estimating technique and does not tell us anything about the relative merits of two programs where one is always "slightly faster" than the other. However, asymptotic analysis has proved useful to computer scientists who must determine if a particular algorithm is worth considering for implementation.

The critical resource for a program is most often its running time. However, you cannot pay attention to running time alone. You must also be concerned with other factors such as the space required to run the program (both main memory and disk space). Typically you will analyze the time required for an algorithm (or the instantiation of an algorithm in the form of a program), and the space required for a data structure.

Many factors affect the running time of a program. Some relate to the environment in which the program is compiled and run. Such factors include the speed of the computer's CPU, bus, and peripheral hardware. Competition with other users for the computer's resources can make a program slow to a crawl. The programming language and the quality of code generated by a particular compiler can have a significant effect. The "coding efficiency" of the programmer who converts the algorithm to a program can have a tremendous impact as well.

If you need to get a program working within time and space constraints on a particular computer, all of these factors can be relevant. Yet, none of these factors address the differences between two algorithms or data structures. To be fair, programs derived from two algorithms for solving the same problem should both be compiled with the same compiler and run on the same computer under the same conditions. As much as possible, the same amount of care should be taken in the programming effort devoted to each program to make the implementations "equally efficient." In this sense, all of the factors mentioned above should cancel out of the comparison because they apply to both algorithms equally.

If you truly wish to understand the running time of an algorithm, there are other factors that are more appropriate to consider than machine speed, programming language, compiler, and so forth. Ideally we would measure the running time of the algorithm under standard benchmark conditions. However, we have no way to calculate the running time reliably other than to run an implementation of the algorithm on some computer. The only alternative is to use some other measure as a surrogate for running time.

Of primary consideration when estimating an algorithm's performance is the number of **basic operations** required by the algorithm to process an input of a certain **size**. The terms "basic operations" and "size" are both rather vague and depend on the algorithm being analyzed. Size is often the number of inputs processed. For example, when comparing sorting algorithms, the size of the problem is typically measured by the number of records to be sorted. A basic operation must have the property that its time to complete does not depend on the particular values of its operands. Adding or comparing two integer variables are examples of basic operations in most programming languages. Summing the contents of an array containing *n* integers is not, because the cost depends on the value of *n* (i.e., the size of the input).

Example 1: *Consider a simple algorithm to solve the problem of finding the largest value in an array of n integers. The algorithm looks at each integer in turn, saving the position of the largest value seen so far. This algorithm is called the* ***largest-value sequential search*** *and is illustrated by the following function:*

```
// Return position of largest value in "A" of size "n"
int largest(int A[], int n) {
int currlarge = 0; // Holds largest element position
for (int i=1; i<n; i++) // For each array element
if (A[currlarge] < A[i]) // if A[i] is larger
currlarge = i; // remember its position
return currlarge; // Return largest position
}
```

Here, the size of the problem is ***n***, the number of integers stored in **A**. The basic operation is to compare an integer's value to that of the largest value seen so far. It is reasonable to assume that it takes a fixed amount of time to do one such comparison, regardless of the value of the two integers or their positions in the array.

Because the most important factor affecting running time is normally the size of the input, for a given input size ***n*** we often express the time ***T*** to run the algorithm as a function of ***n***, written as ***T(n)***. We will always assume ***T(n)*** is a non-negative value. Let us call ***c*** the amount of time required to compare two integers in function `largest`. We do not care right now what the precise value of ***c*** might be. Nor are we concerned with the time required to increment variable `i` because this must be done for each value in the array, or the time for the actual assignment when a larger value is found, or the little bit of extra time taken to initialize `currlarge`. We just want a reasonable approximation for the time taken to execute the algorithm. The total time to run largest is therefore approximately ***cn***, because we must make ***n*** comparisons, with each comparison costing ***c*** time. We say that function `largest` (and the *largest-value sequential search* algorithm in general) has a running time expressed by the equation

$$T(n) = cn$$

This equation describes the growth rate for the running time of the largest value sequential search algorithm.

Example 2: *The running time of a statement that assigns the first value of an integer array to a variable is simply the time required to copy the value of the first array value. We can assume this assignment takes a constant amount of time regardless of the value. Let us call* c_1 *the amount of time necessary to copy an integer. No matter how large the array on a typical computer (given reasonable conditions for memory and array size), the time to copy the value from the first position of the array is always* c_1. *Thus, the equation for this algorithm is simply:*

$$T(n) = c_1$$

indicating that the size of the input ***n*** *has no effect on the running time. This is called a constant running time.*

Example 3*: Consider the following code:*

```
sum = 0;
for (i=1; i<=n; i++)
for (j=1; j<=n; j++)
sum++;
```

What is the running time for this code fragment?

Clearly it takes longer to run when ***n*** *is larger. The basic operation in this example is the increment operation for variable* `sum`. *We can assume that incrementing takes constant time; call this time* c_2. *(We can ignore the time required to initialize sum, and to increment the loop counters* ***i*** *and* ***j****. In practice, these costs can safely be bundled into time* c_2.*) The total number of increment operations is* n^2. *Thus, we say that the running time is*

$$T(n) = c_2 n^2$$

The **growth rate** for an algorithm is the rate at which the **cost** of the algorithm grows as the **size** of its **input** grows. Figure 1 shows a graph for six equations, each meant to describe the running time for a particular program or algorithm. A variety of growth rates representative of typical algorithms are shown. The two equations labelled ***10n*** and ***20n*** are graphed by straight lines. A growth rate of ***cn*** (for ***c*** any positive constant) is often referred to as a **linear growth rat**e or running time. This means that as the value of ***n*** grows, the running time of the algorithm grows in the same proportion. Doubling the value of ***n*** roughly doubles the running time. An algorithm whose running-time equation has a highest-order term containing a factor of n^2 is said to have a **quadratic growth** rate. In Figure 1, the line labelled $2n^2$ represents a quadratic growth rate. The line labelled ***2n*** represents an **exponential growth rate**. This name comes from the fact that ***n*** appears in the exponent. The line labelled ***n!*** is also growing **exponentially**.

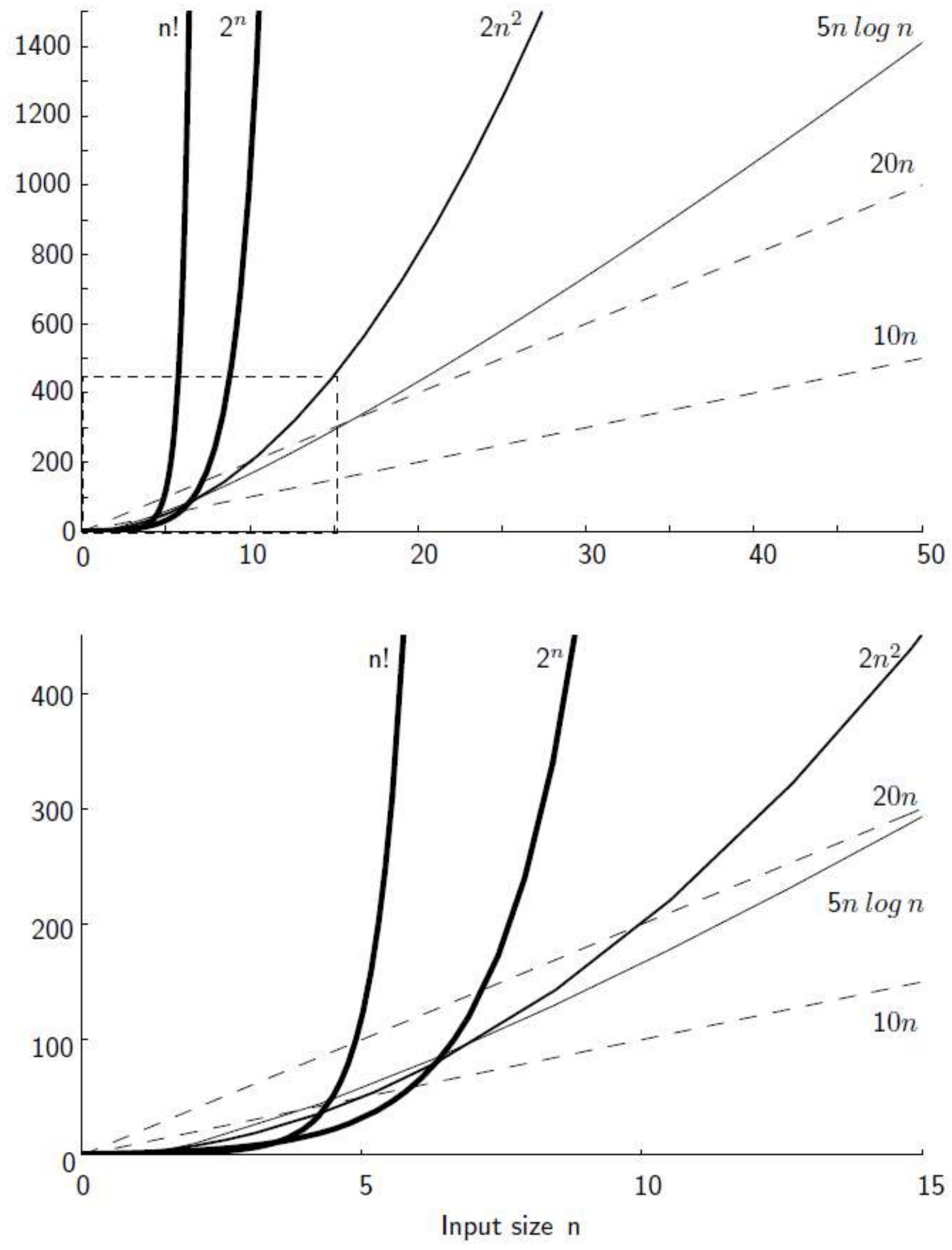


Figure 1: Two views of a graph illustrating the growth rates for six equations. The bottom view shows in detail the lower-left portion of the top view. The horizontal axis represents input size. The vertical axis can represent time, space, or any other measure of cost.

As you can see from Figure 1, the difference between an algorithm whose running time has cost $T(n) = 10n$ and another with cost $T(n) = 2n^2$ becomes tremendous as n grows. For $n > 5$, the algorithm with running time $T(n) = 2n^2$ is already much slower. This is despite the fact that $10n$ has a greater constant factor than $2n^2$. Comparing the two curves marked $20n$ and $2n^2$ shows that changing the constant factor for one of the equations only shifts the point at which the two curves cross. For $n > 10$, the algorithm with cost $T(n) = 2n^2$ is slower than the algorithm with cost $T(n) = 20n$. This graph also shows that the equation $T(n) = 5nlogn$ grows somewhat more quickly than both $T(n) = 10n$ and $T(n) = 20n$, but not nearly so quickly as the equation $T(n) = 2n^2$. For constants $a; b > 1$, n^a grows faster than either $log^b n$ or $logn^b$. Finally, algorithms with cost $T(n) = 2n$ or $T(n) = n!$ are prohibitively expensive for even modest values of n. Note that for constants $a; b \geq 1$, a^n grows faster than $\mathbf{n^b}$.

n	log log n	log n	n	n log n	n^2	n^3	2^n
16	2	4	2^4	$2 \cdot 2^4 = 2^5$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8 \cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10 \cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16 \cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	2^{64K}
1M	≈ 4.3	20	2^{20}	$20 \cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	2^{1M}
1G	≈ 4.9	30	2^{30}	$30 \cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	2^{1G}

Figure 2: Costs for growth rates representative of most computer algorithms.

We can get some further insight into relative growth rates for various algorithms from Figure 2. Most of the growth rates that appear in typical algorithms are shown, along with some representative input sizes. Once again, we see that the growth rate has a tremendous effect on the resources consumed by an algorithm.

2. Best, Worst, and Average Cases

Consider the problem of finding the factorial of **n**. For this problem, there is only one input of a given "size" (that is, there is only a single instance of size n for each value of n). Now consider our largest-value sequential search algorithm of Example 1, which always examines every array value. This algorithm works on many inputs of a given size n. That is, there are many possible arrays of any given size. However, no matter what array the algorithm looks at, its cost will always be the same in that it always looks at every element in the array one time.

For some algorithms, different inputs of a given size require different amounts of time. For example, consider the problem of searching an array containing n integers to find the one with a particular value **K** (assume that **K** appears exactly once in the array). The sequential search algorithm begins at the first position in the array and looks at each value in turn until **K** is found. Once **K** is found, the algorithm stops. This is different from the *largest-value sequential* search algorithm of Example 1, which always examines every array value. There is a wide range of possible running times for the sequential search algorithm. The first integer in the array could have value **K**, and so only one integer is examined. In this case the running time is short. This is the **best case** for this algorithm, because it is not possible for sequential search to look at less than one value. Alternatively, if the last position in the array contains **K**, then the running time is relatively long, because the algorithm must examine ***n*** values. This is the **worst case** for this algorithm, because sequential search never looks at more than ***n*** values. If we implement sequential search as a program and run it many times on many different arrays of size ***n***, or search for many different values of **K** within the same array, we expect the algorithm on average to go halfway through the array before finding the value we seek. On average, the algorithm examines about **n/2** values. We call this the **average case** for this algorithm.

When analyzing an algorithm, should we study the best, worst, or average case? Normally we are not interested in the best case, because this might happen only rarely and generally is too optimistic for a fair characterization of the algorithm's running time. In other words, analysis based on the best case is not likely to be representative of the behaviour of the algorithm. However, there are rare instances where a best-case analysis is useful — in particular, when the best case has high probability of occurring.

How about the worst case? The advantage to analyzing the worst case is that you know for certain that the algorithm must perform at least that well. This is especially important for real-time applications, such as for the computers that monitor an air traffic control system. Here, it would not be acceptable to use an algorithm that can handle n airplanes quickly enough most of the time, but which fails to perform quickly enough when all n airplanes are coming from the same direction.

For other applications — particularly when we wish to aggregate the cost of running the program many times on many different inputs — worst-case analysis might not be a representative measure of the algorithm's performance. Often we prefer to know the average-case running time. This means that we would like to know the typical behaviour of the algorithm on inputs of size n.Unfortunately, average-case analysis is not always possible. Average-case analysis first requires that we understand how the actual inputs to the program (and their costs) are distributed with respect to the set of all possible inputs to the program. For example, it was stated previously that the sequential search algorithm on average examines half of the array values. This is only true if the element with value K is equally likely to appear in any position in the array. If this assumption is not correct, then the algorithm does not necessarily examine half of the array values in the average case.

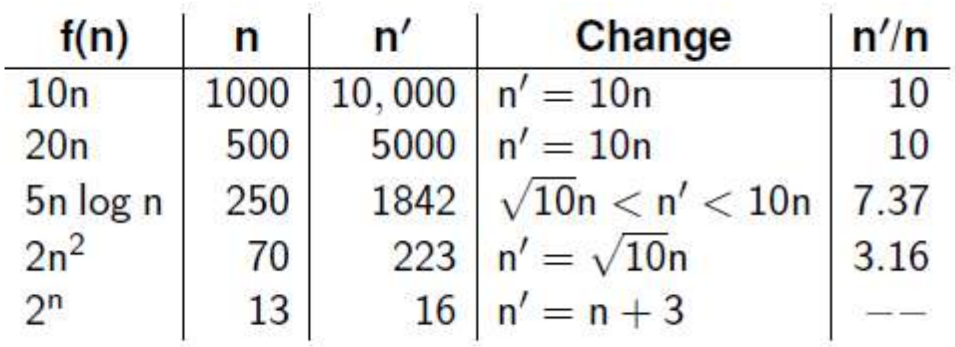
The characteristics of a data distribution have a significant effect on many search algorithms, such as those based on hashing and search trees. Incorrect assumptions about data distribution can have disastrous consequences on a program's space or time performance.

In summary, for real-time applications we are likely to prefer a worst-case analysis of an algorithm. Otherwise, we often desire an average-case analysis if we know enough about the distribution of our input to compute the average case. If not, then we must resort to worst-case analysis.

3. A Faster Computer, or a Faster Algorithm?

Imagine that you have a problem to solve, and you know of an algorithm whose running time is proportional to $\mathbf{n^2}$. Unfortunately, the resulting program takes ten times too long to run. If you replace your current computer with a new one that is ten times faster, will the n^2 algorithm become acceptable? If the problem size remains the same, then perhaps the faster computer will allow you to get your work done quickly enough even with an algorithm having a high growth rate. But a funny thing happens to most people who get a faster computer. They don’t run the same problem faster. They run a bigger problem! Assuming on your old computer you were content to sort 10,000 records because that could be done by the computer during your lunch break. On your new computer you might hope to sort 100,000 records in the same time. You won’t be back from lunch any sooner, so you are better off solving a larger problem. And because the new machine is ten times faster, you would like to sort ten times as many records.

If your algorithm’s growth rate is linear (i.e., if the equation that describes the running time on input size **n** is **T(n) = cn** for some constant c), then 100,000 records on the new machine will be sorted in the same time as 10,000 records on the old machine. If the algorithm’s growth rate is greater than **cn**, such as $\mathbf{c_1n^2}$, then you will not be able to do a problem ten times the size in the same amount of time on a machine that is ten times faster. How much larger a problem can be solved in a given amount of time by a faster computer? Assume that the new machine is ten times faster than the old. Say that the old machine could solve a problem of size n in an hour. What is the largest problem that the new machine can solve in one hour? Figure 3 shows how large a problem can be solved on the two machines for the five running-time functions from Figure 1.



f(n)	n	n′	Change	n′/n
10n	1000	10,000	$n' = 10n$	10
20n	500	5000	$n' = 10n$	10
5n log n	250	1842	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	−−

Figure 3: The increase in problem size that can be run in a fixed period of time on a computer that is ten times faster. The first column lists the right-hand sides for each of the five growth rate equations of Figure 1. For the purpose of this example, arbitrarily assume that the old machine can run 10,000 basic operations in one hour. The second column shows the maximum value for **n** that can be run in 10,000 basic operations on the old machine. The third column shows the value for $\mathbf{n'}$, the new maximum size for the problem that can be run in the same time on the new machine that is ten times faster. Variable $\mathbf{n'}$ is the greatest size for the problem that can run in 100,000 basic operations. The fourth column shows how the size of **n** changed to become $\mathbf{n'}$ on the new machine. The fifth column shows the increase in the problem size as the ratio of $\mathbf{n'}$ to **n**.

This table illustrates many important points. The first two equations are both linear; only the value of the constant factor has changed. In both cases, the machine that is ten times faster gives an increase in problem size by a factor of ten. In other words, while the value of the constant does affect the absolute size of the problem that can be solved in a fixed amount of time, it does not affect the improvement in problem size (as a proportion to the original size) gained by a faster computer. This relationship holds true regardless of the algorithm’s growth rate: Constant factors never affect the relative improvement gained by a faster computer.

An algorithm with time equation **$T(n) = 2n^2$** does not receive nearly as great an improvement from the faster machine as an algorithm with linear growth rate. Instead of an improvement by a factor of ten, the improvement is only the square root of that: $\sqrt{10} = 3.16$.

Thus, the algorithm with higher growth rate not only solves a smaller problem in a given time in the first place, it also receives less of a speedup from a faster computer. As computers get ever faster, the disparity in problem sizes becomes ever greater. The algorithm with growth rate **$T(n) = 5n \log n$** improves by a greater amount than the one with quadratic growth rate, but not by as great an amount as the algorithms with linear growth rates. Note that something special happens in the case of the algorithm whose running time grows exponentially. In Figure 1, the curve for the algorithm whose time is proportional to 2^n goes up very quickly. In Figure 3, the increase in problem size on the machine ten times as fast is shown to be about **$n + 3$** (to be precise, it is **$n + \log_2 10$**). The increase in problem size for an algorithm with exponential growth rate is by a constant addition, not by a multiplicative factor. Because the old value of **n** was 13, the new problem size is 16.

Assuming we buy a computer ten times faster , then the new computer (100 times faster than the original computer) will only run a problem of size 19. If you had a second program whose growth rate is **2^n** and for which the original computer could run a problem of size 1000 in an hour, than a machine ten times faster can run a problem only of size 1003 in an hour! Thus, an exponential growth rate is radically different than the other growth rates shown in Figure 3.

Instead of buying a faster computer, consider what happens if you replace an algorithm whose running time is proportional to **n^2** with a new algorithm whose running time is proportional to **$n \log n$**. In the graph of Figure 1, a fixed amount of time would appear as a horizontal line. If the line for the amount of time available to solve your problem is above the point at which the curves for the two growth rates in question meet, then the algorithm whose running time grows less quickly is faster.

An algorithm with running time **$T(n) = n^2$** requires $1024 \times 1024 = 1,048,576$ time steps for an input of size $n = 1024$. An algorithm with running time **$T(n) = n \log n$** requires $1024 \times 10 = 10,240$ time steps for an input of size $n = 1024$, which is an improvement of much more than a factor of ten when compared to the algorithm with running time **$T(n) = n^2$**. Because **$n^2 > 10n \log n$** whenever **$n > 58$**, if the typical problem size is larger than 58 for this example, then you would be much better off changing algorithms instead of buying a computer ten times faster. Furthermore, when you do buy a faster computer, an algorithm with a slower growth rate provides a greater benefit in terms of larger problem size that can run in a certain time on the new computer.

4. Asymptotic Analysis

Despite the larger constant for the curve labelled **10n** in Figure 1, the curve labelled **$2n^2$** crosses it at the relatively small value of **$n = 5$**. What if we double the value of the constant in front of the linear equation? As shown in the graph, the curve labelled **20n** is surpassed by the curve labelled **$2n^2$** once **$n = 10$**. The additional factor of two for the linear growth rate does not much matter; it only doubles the x-coordinate for the intersection point. In general, changes to a constant factor in either equation only shift *where* the two curves cross, not *whether* the two curves cross.

When you buy a faster computer or a faster compiler, the new problem size that can be run in a given amount of time for a given growth rate is larger by the same factor, regardless of the constant on the running-time equation. The time curves for two algorithms with different growth rates still cross, regardless of their running-time equation constants.

For these reasons, we usually ignore the constants when we want an estimate of the running time or other resource requirements of an algorithm. This simplifies the analysis and keeps us thinking about the most important aspect: the growth rate. This is called **asymptotic algorithm analysis**. To be precise, asymptotic analysis refers to the study of an algorithm as the input size "*gets big*" or reaches a limit (in the calculus sense).

However, it has proved to be so useful to ignore all constant factors that asymptotic analysis is used for most algorithm comparisons. It is not always reasonable to ignore the constants. When comparing algorithms meant to run on small values of n, the constant can have a large effect. For example, if the problem is to sort a collection of exactly five records, then an algorithm designed for sorting thousands of records is probably not appropriate, even if its asymptotic analysis indicates good performance.

There are rare cases where the constants for two algorithms under comparison can differ by a factor of 1000 or more, making the one with lower growth rate impractical for most purposes due to its large constant.

5. Upper Bounds

Several terms are used to describe the running-time equation for an algorithm. These terms — and their associated symbols — indicate precisely what aspect of the algorithm's behaviour is being described. One is the **upper bound** for the growth of the algorithm's running time. It indicates the upper or highest growth rate that the algorithm can have.

To make any statement about the upper bound of an algorithm, we must be making it about some class of inputs of *size* ***n***. We measure this upper bound nearly always on the *best-case*, *average-case*, or *worst-case* inputs. Thus, we cannot say, "this algorithm has an upper bound to its growth rate of n^2." We must say something like, "this algorithm has an *upper bound* to its growth rate of n^2 in the *average case*."

Because the phrase "has an upper bound to its growth rate of ***f(n)***" is long and often used when discussing algorithms, we adopt a special notation, called ***big-Oh notation***. If the upper bound for an algorithm's growth rate (for, say, the worst case) is ***f(n)***, then we can write that this algorithm is "in the set ***O(f(n))***in the *worst case*" (or just "in ***O(f(n))***in the worst case"). For example, if n^2 grows as fast as **T(n)** (the running time of our algorithm) for the worst-case input, we would say the algorithm is "in O(n2) in the worst case." The following is a precise definition for an upper bound. **T(n)** represents the true running time of the algorithm. **f(n)** is some expression for the upper bound.

For $T(n)$ a non-negatively valued function, $T(n)$ is in set $O(f(n))$ if there exist two positive constants c and n_0 such that $T(n) \le cf(n)$ for all $n > n_0$.

The constant $\mathbf{n_0}$ is the smallest value of **n** for which the claim of an upper bound holds true. Usually $\mathbf{n_0}$ is small, such as 1, but does not need to be. You must also be able to pick some constant **c**, but it is irrelevant what the value for c actually is. In other words, the definition says that for all inputs of the type in question (such as the worst case for all inputs of size n) that are large enough (i.e., $n > n_0$), the algorithm always executes in less than ***cf(n)*** steps for some constant **c**.

Example 4: *Consider the sequential search algorithm for finding a specified value in an array of integers. If visiting and examining one value in the array requires* c_s *steps where* c_s *is a positive number, and if the value we search for has equal probability of appearing in any position in the array, then in the average case* $T(n) = c_s n/2$. *For all values of* $n > 1$, $c_s n/2 \le c_s n$. *Therefore, by the definition,* ***T(n)*** *is in* ***O(n)*** *for* $n_0 = 1$ *and* $c = c_s$.

Example 5: *For a particular algorithm,* $T(n) = c_1 n^2 + c_2 n$ *in the average case where* c_1 *and* c_2 *are positive numbers. Then,* $c_1 n^2 + c_2 n \le c_1 n^2 + c_2 n^2 \le (c_1 + c_2) n^2$ *for all* $n > 1$. *So,* $T(n) \le cn^2$ *for* $c = c_1 + c_2$, *and* $n_0 = 1$. *Therefore,* ***T(n)*** *is in* $O(n^2)$ *by the definition.*

Example 6: *Assigning the value from the first position of an array to a variable takes constant time regardless of the size of the array. Thus,* ***T(n) = c*** *(for the best, worst, and average cases). We could say in this case that* ***T(n) is in O(c)***. *However, it is traditional to say that an algorithm whose running time has a constant upper bound is in* ***O(1)***.

Just knowing that something is in **O(f(n))** says only how bad things can get. Perhaps, things are not nearly so bad. Because we know sequential search is in **O(n)** in the worst case, it is also true to say that sequential search is in $O(n^2)$. But sequential search is practical for large **n**, in a way that is not true for some other algorithms in $O(n^2)$. We always seek to define the running time of an algorithm with the tightest (*lowest*) possible upper bound. Thus, we prefer to say that sequential search is in **O(n)**.

6. Lower Bounds

Big-Oh notation describes an upper bound. In other words, big-Oh notation states a claim about the greatest amount of some resource (usually time) that is required by an algorithm for some class of inputs of size n (typically the worst such input, the average of all possible inputs, or the best such input).

Similar notation is used to describe the least amount of a resource that an algorithm needs for some class of input. Like big-Oh notation, this is a measure of the algorithm's growth rate. Like big-Oh notation, it works for any resource, but we most often measure the least amount of time required. And again, like big-Oh notation, we are measuring the resource required for some particular class of inputs: the worst-, average-, or best-case input of size ***n***. The lower bound for an algorithm (or a problem, as explained later) is denoted by the symbol $\mathbf{\Omega}$, pronounced "big-Omega" or just "Omega." The following definition for $\mathbf{\Omega}$ is symmetric with the definition of big-Oh.

*For **T(n)** a non-negatively valued function, T(n) is in set (g(n)) if there exist two positive constants c and n_0 such that $T(n) \geq cg(n)$ for all $n > n_0$*

Example 7: *Assume $T(n) = c_1n^2 + c_2n$ for c_1 and $c_2 > 0$. Then,*

$c_1n^2 + c_2n \leq c_1n^2$

for all $n > 1$. So, $T(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$. Therefore, T(n) is in $\Omega(n^2)$ by the definition.

---

7. Θ Notation

The definitions for **big-Oh** and **Ω** give us ways to describe the upper bound for an algorithm (if we can find an equation for the maximum cost of a particular class of inputs of size ***n***) and the lower bound for an algorithm (if we can find an equation for the minimum cost for a particular class of inputs of size **n**). When the upper and lower bounds are the same within a constant factor, we indicate this by using **Θ (big-Theta)** notation. An algorithm is said to be **Θ *(h(n))*** if it is ***in O(h(n))*** and it is in **Ω *(h(n))***. Note that we drop the word "in" for **Θ** notation, because there is a strict equality for two equations with the same **Θ**. In other words, if **f(n)** is **Θ *(g(n))***, then ***g(n)*** is **Θ *(f(n))***.

While some programmers will casually say that an algorithm is "order of" or "big-Oh" of some cost function, it is generally better to use **Θ** notation rather than **big-Oh** notation whenever we have sufficient knowledge about an algorithm to be sure that the upper and lower bounds indeed match. The, **Θ** notation will be used in preference to *big-Oh* notation whenever our state of knowledge makes that possible. Limitations on our ability to analyze certain algorithms may require use of big-Oh or Ω notations. In rare occasions when the discussion is explicitly about the upper or lower bound of a problem or algorithm, the corresponding notation will be used in preference to **Θ** notation.

8. Simplifying Rules

Once you determine the running-time equation for an algorithm, it really is a simple matter to derive the big-Oh, Ω, and Θ expressions from the equation. You do not need to resort to the formal definitions of asymptotic analysis. Instead, you can use the following rules to determine the simplest form.

1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n) + f_2(n)$ is in $O(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

The first rule says that if some function g(n) is an upper bound for your cost function, then any upper bound for g(n) is also an upper bound for your cost function. A similar property holds true for Ω notation: If g(n) is a lower bound for your cost function, then any lower bound for g(n) is also a lower bound for your cost function. Likewise for Θ notation.

The significance of rule (2) is that you can ignore any multiplicative constants in your equations when using big-Oh notation. This rule also holds true for Ω and Θ notations.

Rule (3) says that given two parts of a program run in sequence (whether two statements or two sections of code), you need consider only the more expensive part. This rule applies to Ω and Θ notations as well: For both, you need consider only the more expensive part.

Rule (4) is used to analyze simple loops in programs. If some action is repeated some number of times, and each repetition has the same cost, then the total cost is the cost of the action multiplied by the number of times that the action takes place. This rule applies to Ω and Θ notations as well.

Taking the first three rules collectively, you can ignore all constants and all lower-order terms to determine the asymptotic growth rate for any cost function. The advantages and dangers of ignoring constants were discussed near the beginning of this section. Ignoring lower-order terms is reasonable when performing an asymptotic analysis. The higher-order terms soon swamp the lower-order terms in their contribution to the total cost as ***n*** becomes larger. Thus, if **T(n)** = $3n^4+5n^2$, then T(n) is in O(n^4). The n^2 term contributes relatively little to the total cost. Throughout the rest of this course, these simplifying rules are used when discussing the cost for a program or algorithm.

9. Classifying Functions

Given functions *f(n)* and *g(n)* whose growth rates are expressed as algebraic equations, we might like to determine if one grows faster than the other. The best way to do this is to take the limit of the two functions as n grows towards infinity,

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

If the limit goes to ∞, then *f(n)* is in Ω (*g(n)*) because *f(n)* grows faster. If the limit goes to zero, then *f(n)* is in O(*g(n)*) because *g(n)* grows faster. If the limit goes to some constant other than zero, then f(n) = Θ(*g(n)*) because both grow at the same rate.

Example 8: *If f(n) = 2n log n and g(n) = n^2, is f(n) in O(g(n)), Ω (g(n)), or Θ (g(n))?*

Solution:

$$\frac{n^2}{2n\log n}=\frac{n}{2\log n},$$

$$\lim_{n\to\infty}\frac{n^2}{2n\log n}=\infty$$

since n grows faster than 2 log n. Thus, n^2 is in Ω (2n log n).

10. Calculating the Running Time for a Program

This section presents the analysis for several simple code fragments.

***Example 9:** We begin with an analysis of a simple assignment statement to an integer variable.*

a = b;

Because the assignment statement takes constant time, it is Θ(1).

***Example 10:** Consider a simple **for** loop.*

```
sum = 0;

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

sum +=n;
```

*The first line is Θ(1). The **for loop** is repeated n times. The third line takes constant time so, by simplifying rule (4) of the previous section, the total cost for executing the two lines making up the for loop is Θ(n). By rule (3), the cost of the entire code fragment is also Θ(n).*

***Example 11:** We now analyze a code fragment with several for loops, some of which are nested.*

```
sum = 0;

for (i=1; i<=n; i++) // First for loop

for (j=1; j<=i; j++) // is a double loop

sum++;

for (k=0; k<n; k++) // Second for loop

A[k] = k;
```

This code fragment has three separate statements: the first assignment statement and the two for loops. Again the assignment statement takes constant time; call it c_1. The second for loop is just like the one in Example 10 and takes c_2n = (n) time.

*The first for loop is a double loop and requires a special technique. We work from the inside of the **loop** outward. The expression **sum++** requires constant time; call it c_3. Because the inner for loop is executed **i** times, by simplifying rule (4) it has cost c_3i. The outer **for loop** is executed n times, but each time the cost of the inner loop is different because it costs c_3i*

with i changing each time. You should see that for the first execution of the outer loop, i is 1. For the second execution of the outer loop, i is 2. Each time through the outer loop, i becomes one greater, until the last time through the loop when i = n. Thus, the total cost of the loop is c_3 times the sum of the integers 1 through n. We know that

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

which is $\Theta(n^2)$. By simplifying rule (3), $\Theta(c_1+c_2n+c_3n^2)$ is simply $\Theta(n^2)$.

Example 12 *Compare the asymptotic analysis for the following two code fragments:*

```
sum1 = 0;
for (i=1; i<=n; i++) // First double loop
for (j=1; j<=n; j++) // do n times
sum1++;
sum2 = 0;
for (i=1; i<=n; i++) // Second double loop
for (j=1; j<=i; j++) // do i times
sum2++;
```

*In the first double loop, the inner for loop always executes **n** times. Because the outer loop executes **n** times, it should be obvious that the statement **sum1++** is executed precisely **n^2** times. The second loop is similar to the one analyzed in the previous example, with cost $\sum_{j=1}^{n} j$. This is approximately $1/2n^2$. Thus, both double loops cost $\Theta(n^2)$, though the second requires about half the time of the first.*

Example 13: *Not all doubly nested for loops are $\Theta(n^2)$. The following pair of nested loops illustrates this fact.*

```
sum1 = 0;
for (k=1; k<=n; k*=2) // Do log n times
for (j=1; j<=n; j++) // Do n times
sum1++;
sum2 = 0;
for (k=1; k<=n; k*=2) // Do log n times
for (j=1; j<=k; j++) // Do k times
sum2++;
```

*When analyzing these two code fragments, we will assume that **n** is a power of two. The first code fragment has its outer for loop executed log n + 1 times because on each iteration k is multiplied by two until it reaches **n**. Because the inner loop always executes n times, the total cost for the first code fragment can be expressed as $\sum_{i=0}^{\log n} n$. Note that a variable substitution takes place here to create the summation, with $k = 2^i$. We know the solution for this summation is $\Theta(n \log n)$. In the second code fragment, the outer loop is also executed* *log n + 1 times. The inner loop has cost k, which doubles each time. The summation can be expressed as* $\sum_{i=0}^{\log n} 2^i$ *where n is assumed to be a power of two and again k = 2^i. This results in Θ(n).*

11. Some General Rules

RULE 1-FOR LOOPS:

The running time of a ***for loop*** is at most the running time of the statements inside the for loop (including tests) times the number of iterations. This also applies for ***while loops.***

RULE 2-NESTED FOR LOOPS:

Analyze these inside out. The total running time of a statement inside a group of nested for loops is the running time of the statement multiplied by the product of the sizes of all the ***for*** loops. This also applies for ***while loops.***

As an example, the following program fragment is O(n^2):

```
for( i=0; i<n; i++ )
for( j=0; j<n; j++ )
k++;
```

RULE 3-CONSECUTIVE STATEMENTS:

These just add (which means that the maximum is the one that counts see rule 3). As an example, the following program fragment, which has a cost of O(n) followed by O(n^2), is also O (n^2):

```
for( i=0; i<n; i++)
a[i] = 0;
for( i=0; i<n; i++ )
for( j=0; j<n; j++ )
a[i] += a[j] + i + j;
```

RULE 4-lF/ELSE:

For the fragment

```
if( cond )
S1
```

else

S2

the running time of an if/else statement is never more than the running time of the test plus the larger of the running times of S1 and S2. Clearly, this can be an over-estimate in some cases, but it is never an underestimate.

For ***switch*** statements, the worst-case cost is that of the most expensive branch. For subroutine calls, simply add the cost of executing the subroutine. There are rare situations in which the probability for executing the various branches of an ***if or switch*** statement are functions of the input size. For example, for input of size ***n***, the ***then*** clause of an ***if*** statement might be executed with probability 1/n. An example would be an ***if*** statement that executes the ***then*** clause only for the smallest of ***n*** values. To perform an average-case analysis for such programs, we cannot simply count the cost of the ***if*** statement as being the cost of the more expensive branch. In such situations, the technique of amortized analysis can come to the rescue. This is beyond the scope of this course and will not be discussed.

Determining the execution time of a recursive subroutine can be difficult. The running time for a recursive subroutine is typically best expressed by a recurrence relation.