Design and Analysis of Algorithm

Introduction

There are many algorithms that can solve a given problem. They will have different characteristics that will determine how efficiently each will operate. When we analyze an algorithm, we first have to show that the algorithm does properly solve the problem because if it doesn't, its efficiency is not important. Next, we look at how efficiently it solves the problem. This chapter sets the groundwork for the analysis and comparison of more complex algorithms.

Analyzing an algorithm determines the amount of "time" that algorithm takes to execute. This is not really a number of seconds or any other clock measurement but rather an approximation of the number of operations that an algorithm performs. The number of operations is related to the execution time, so we will sometimes use the word *time* to describe an algorithm's computational complexity. The actual number of seconds it takes an algorithm to execute on a computer is not useful in our analysis because we are concerned with the relative efficiency of algorithms that solve a particular problem. You should also see that the actual execution time is not a good measure of algorithm efficiency because an algorithm does not get "better" just because we move it to a faster computer or "worse" because we move it to a slower one. The actual number of operations done for some specific size of input data sets is not very interesting nor does it tells us very much. Instead, our analysis will determine an equation that relates the number of operations that a particular algorithm does to the size of the input. We can then compare two algorithms by comparing the rate at which their equations grow. The growth rate is critical because there are instances where algorithm A may take fewer operations than algorithm B when the input size is small, but many more when the input size gets large.

In a very general sense, algorithms can be classified as either repetitive or recursive. Repetitive algorithms have loops and conditional statements as their basis, and so their analysis will entail determining the work done in the loop and how many times the loop executes. Recursive algorithms solve a large problem by breaking it into pieces and then applying the algorithm to each of the pieces. These are sometimes called divide and conquer algorithms and provide a great deal of power in solving problems. The process of solving a large problem by breaking it up into smaller pieces can produce an algorithm that is small, straightforward, and simple to understand. Analyzing a recursive algorithm will entail determining the amount of work done to produce the smaller pieces and then putting their individual solutions together to get the solution to the whole problem. Combining this information with the number of the smaller pieces and their sizes, we can produce a recurrence relation for the algorithm. This recurrence relation can then be converted into a closed form that can be compared with other equations.

We begin this chapter by describing what analysis is and why we do it. We then look at what operations will be considered and what categories of analysis we will do. Because mathematics is critical to our analysis, the next few sections explore the important mathematical concepts and properties used to analyze iterative and recursive algorithms.

WHAT IS ANALYSIS?

The analysis of an algorithm provides background information that gives us a general idea of how long an algorithm will take for a given problem set. For each algorithm considered, we will come up with an estimate of how long it will take to solve a problem that has a set of *N* input values. So, for example, we might determine how many comparisons a sorting algorithm does to put a list of *N* values into ascending order, or we might determine how many arithmetic operations it takes to multiply two matrices of size $N \times N$.

There are a number of algorithms that will solve a problem. Studying the analysis of algorithms gives us the tools to choose between algorithms. For example, consider the following two algorithms to find the largest of four values:

Algorithm1

```
largest = a
if b > largest then
    largest = b
end if
```

```
    if c > largest then
        largest = c
    end if
    if d > largest then
        largest = d
    end if
```

Algorithm 2

```
    return largest
        if a > b then
        if a > c then
        if a > d then
        return a
    else
        return d
    end if
    else
        if c > d then
        return c
        else
        return d
    end if
    end if
    else
        if b > c then
        if b > d then
        return b
        else
        return d
        end if
    else
        if c > d then
        return c
        else
        return d
    end if
    end if
    end if
```

If you examine these two algorithms, you will see that each one will do exactly three comparisons to find the answer. Even though the first is easier for us to read and understand, they are both of the same level of complexity for a computer to execute. In terms of time, these two algorithms are the same, but in terms of space, the first needs more because of the temporary variable called largest. This extra space is not significant if we are comparing numbers or characters, but it may be with other types of data. In many modern programming languages, we can define comparison operators for large and complex objects or records. For those cases, the amount of space needed for the temporary variable could be quite significant. When we are interested in the efficiency of algorithms, we will primarily be concerned with time issues, but when space may be an issue, it will also be discussed.

The purpose of determining these values is to then use them to compare how efficiently two different algorithms solve a problem. For this reason, we will never compare a sorting algorithm with a matrix multiplication algorithm, but rather we will compare two different sorting algorithms to each other.

The purpose of analysis of algorithms is not to give a formula that will tell us exactly how many seconds or computer cycles a particular algorithm will take. This is not useful information because we would then need to talk about the type of computer, whether it has one or many users at a time, what processor it has, how fast its clock is, whether it has a complex or reduced instruction set processor chip, and how well the compiler optimizes the executable code.

All of those will have an impact on how fast a program for an algorithm will run. To talk about analysis in those terms would mean that by moving a program to a faster computer, the algorithm would become better because it now completes its job faster. That's not true, so, we do our analysis without regard to any specific computer. In the case of a small or simple routine it might be possible to count the exact number of operations performed as a function of N. Most of the time, however, this will not be useful. Therefore, the difference between an algorithm that does $N + 5$ operations and one that does $N + 250$ operations becomes meaningless as N gets very large. As an introduction to analysis of algorithms, however, we will count the exact number of operations for this first section.

Another reason we do not try to count every operation that is performed by an algorithm is that we could fine-tune an algorithm extensively but not really make much of a difference in its overall performance. For instance, let's say that we have an algorithm that counts the number of different characters in a file. An algorithm for that might look like the following:

```
for all 256 characters do
    assign zero to the counter
end for
    while there are more characters in the file do
    get the next character
    increment the counter for this character by one
end while
```

When we look at this algorithm, we see that there are 256 passes for the initialization loop. If there are *N* characters in the input file, there are *N* passes for the second loop. So the question becomes What do we count? In a for loop, we have the initialization of the loop variable and then for each pass of the loop, a check that the loop variable is within the bounds, the execution of the loop, and the increment of the loop variable. This means that the initialization loop does a set of 257 assignments (1 for the loop variable and 256 for the counters), 256 increments of the loop variable, and 257 checks that this variable is within the loop bounds (the extra one is when the loop stops). For the second loop, we will need to do a check of the condition $N + 1$ times (the + 1 is for the last check when the file is empty), and we will increment *N* counters. The total number of operations is

Increments $N + 256$

Assignments 257

Conditional checks $N + 258$

So, if we have 500 characters in the file, the algorithm will do a total of 1771 operations, of which 770 are associated with the initialization (43%).

Now consider what happens as the value of *N* gets large. If we have a file with 50,000 characters, the algorithm will do a total of 100,771 operations, of which there are still only 770 associated with the initialization (less than 1% of the total work). The number of initialization operations has not changed, but they become a much smaller percentage of the total as *N* increases. Let’s look at this another way. Computer organization information shows that copying large blocks of data is as quick as an assignment. We could initialize the first 16 counters to zero and then copy this block 15 times to fill in the rest of the counters. This would mean a reduction in the initialization pass down to 33 conditional checks, 33 assignments, and 31 increments. This reduces the initialization operations to 97 from 770, a saving of 87%. When we consider this relative to the work of processing the file of 50,000 characters, we have saved less than 0.7% (100,098 vs. 100,771). Notice we could save even more time if we did all of these initializations without loops, because only 31 pure assignments would be needed, but this would only save an additional 0.07%. It’s not worth the effort.

We see that the importance of the initialization is small relative to the overall execution of this algorithm. In analysis terms, the cost of the initialization becomes meaningless as the number of input values increases.

The earliest work in analysis of algorithms determined the computability of an algorithm on a Turing machine. The analysis would count the number of times that the transition function needed to be applied to solve the problem. An analysis of the space needs of an algorithm would count how many cells of a Turing machine tape would be needed to solve the problem. This sort of analysis is a valid determination of the relative speed of two algorithms, but it is also time consuming and difficult. To do this sort of analysis, you would first need to determine the process used by the transition functions of the Turing machine that carries out the algorithm. Then you would need to determine how long it executes—a very tedious process.

An equally valid way to analyze an algorithm, and the one we will use, is to consider the algorithm as it is written in a higher-level language. This language can be Pascal, C, C++, Java, or a general pseudocode. The specifics don’t really matter as long as the language can express the major control structures common to algorithms. This means that any language that has a looping mechanism, like a for or while, and a selection mechanism, like an if, case, or switch, will serve our needs. Because we will be concerned with just one algorithm at a time, we will rarely write more than a single function or code fragment, and so the power of many of the languages mentioned will not even come into play. For this reason, a generic pseudocode will be used in this book.

Some languages uses short-circuit evaluation when determining the value of a Boolean expression. This means that in the expression A and B, the term B will only be evaluated if A is true, because if A is false, the result will be false no matter what B is. Likewise, for A or B, B will not be evaluated if A is true. As we will see, counting a compound expression as one or two comparisons will not be significant. So, once we are past the basics in this chapter, we will not worry about short-circuited evaluations.

Input Classes

Input plays an important role in analyzing algorithms because it is the input that determines what the path of execution through an algorithm will be. For example, if we are interested in finding the largest value in a list of *N* numbers, we can use the following algorithm:

```
largest = list[1]
for i = 2 to N do
if (list[i] > largest) then
largest = list[i]
end if
end for
```

We can see that if the list is in decreasing order, there will only be one assignment done before the loop starts. If the list is in increasing order, however, there will be *N* assignments (one before the loop starts and *N*-1 inside the loop). Our analysis must consider more than one possible set of input, because if we only look at one set of input, it may be the set that is solved the fastest (or slowest). This will give us a false impression of the algorithm. Instead we consider all types of input sets.

When looking at the input, we will try to break up all the different input sets into classes based on how the algorithm behaves on each set. This helps to reduce the number of possibilities that we will need to consider.

For example, if we use our largest-element algorithm with a list of 10 distinct numbers, there are 10!, or 3,628,800, different ways that those numbers could be arranged. We saw that if the largest is first, there is only one assignment done, so we can take the 362,880 input sets that have the largest value first and put them into one class. If the largest value is second, the algorithm will do exactly two assignments. There are another 362,880 inputs sets with the largest value second, and they can all be put into another class. When looking at this algorithm, we can see that there will be between one and *N* assignments. We would, therefore, create *N* different classes for the input sets based on the number of assignments done. As you will see, we will not necessarily care about listing or describing all of the input sets in each class, but we will need to know how many classes there are and how much work is done for each.

The number of possible inputs can get very large as *N* increases. For instance, if we are interested in a list of 10 distinct numbers, there are 3,628,800 different orderings of these 10 numbers. It would be impossible to look at all of these different possibilities. We instead break these possible lists into classes based on what the algorithm is going to do. For the above algorithm, the breakdown could be based on where the largest value is stored and would result in 10 different classes. For a different algorithm, for example, one that finds the largest and smallest values, our breakdown could be based on where the largest and smallest are stored and would result in 90 different classes. Once we have identified the classes, we can look at how an algorithm would behave on one input from each of the classes. If the classes are properly chosen, all input sets in the class will have the same number of operations, and all of the classes are likely to have different results.

Space Complexity

Most of what we will be discussing is going to be how efficient various algorithms are in terms of time, but some forms of analysis could be done based on how much space an algorithm needs to complete its task. This space complexity analysis was critical in the early days of computing when storage space on a computer (both internal and external) was limited. When considering space complexity, algorithms are divided into those that need extra space to do their work and those that work in place. It was not unusual for programmers to choose an algorithm that was slower just because it worked in place, because there was not enough extra memory for a faster algorithm.

Computer memory was at a premium, so another form of space analysis would examine all of the data being stored to see if there were more efficient ways to store it. For example, suppose we are storing a real number that has only one place of precision after the decimal point and ranges between 10 and +10. If we store this as a real number, most computers will use between 4 and 8 bytes of memory, but if we first multiply the value by 10, we can then store this as an integer between

100 and +100. This needs only 1 byte, a savings of 3 to 7 bytes. A program that stores 1000 of these values can save 3000 to 7000 bytes.
When you consider that computers as recently as the early 1980s might have only had 65,536 bytes of memory, these savings are significant. It is this need to save space on these computers along with the longevity of working computer programs that lead to all of the Y2K bug problems. When you have a program that works with a lot of dates, you use half the space for the year by storing it as 99 instead of 1999. Also, people writing programs in the 1980s and earlier never really expected their programs to still be in use in 2000.

Looking at software that is on the market today, it is easy to see that space analysis is not being done. Programs, even simple ones, regularly quote space needs in a number of megabytes. Software companies seem to feel that making their software space efficient is not a consideration because customers who don’t have enough computer memory can just go out and buy another 32 megabytes (or more) of memory to run the program or a bigger hard disk to store it. This attitude drives computers into obsolescence long before they really are obsolete.

A recent change to this is the popularity of personal digital assistants (PDAs). These small handheld devices typically have between 2 and 8 megabytes for both their data and software. In this case, developing small programs that store data compactly is not only important, it is critical.

What to count and Consider

Deciding what to count involves two steps. The first is choosing the significant operation or operations, and the second is deciding which of those operations are integral to the algorithm and which are overhead or bookkeeping. There are two classes of operations that are typically chosen for the significant operation: comparison or arithmetic. The comparison operators are all considered equivalent and are counted in algorithms such as searching and sorting. In these algorithms, the important task being done is the comparison of two values to determine, when searching, if the value is the one we are looking for or, when sorting, if the values are out of order. Comparison operations include equal, not equal, less than, greater than, less than or equal, and greater than or equal.

We will count arithmetic operators in two groups: additive and multiplicative. Additive operators (usually called *additions* for short) include addition, subtraction, increment, and decrement. Multiplicative operators (usually called *multiplications* for short) include multiplication, division, and modulus.

These two groups are counted separately because multiplications are considered to take longer than additions. In fact, some algorithms are viewed more favorably if they reduce the number of multiplications even if that means a similar increase in the number of additions. In algorithms beyond the scope of this book, logarithms and geometric functions that are used in algorithms would be another group even more time consuming than multiplications because those are frequently calculated by a computer through a power series.

A special case is integer multiplication or division by a power of 2. This operation can be reduced to a shift operation, which is considered as fast as an addition. There will, however, be very few cases when this will be significant, because multiplication or division by 2 is commonly found in divide and conquer algorithms that frequently have comparison as their significant operation.

Cases to Consider

Choosing what input to consider when analyzing an algorithm can have a significant impact on how an algorithm will perform. If the input list is already sorted, some sorting algorithms will perform very well, but other sorting algorithms may perform very poorly. The opposite may be true if the list is randomly arranged instead of sorted. Because of this, we will not consider just one input set when we analyze an algorithm. In fact, we will actually look for those input sets that allow an algorithm to perform the most quickly and the most slowly. We will also consider an overall average performance of the algorithm as well.

Best Case

As its name indicates, the best case for an algorithm is the input that requires the algorithm to take the shortest time. This input is the combination of values that causes the algorithm to do the least amount of work. If we are looking at a searching algorithm, the best case would be if the value we are searching for (commonly called the target or key) was the value stored in the first location that the search algorithm would check. This would then require only one comparison no matter how complex the algorithm is. Notice that for searching through a list of values, no matter how large, the best case will result in a constant time of 1. Because the best case for an algorithm will usually be a very small and frequently constant value, we will not do a best-case analysis very frequently.

Worst Case

Worst case is an important analysis because it gives us an idea of the most time an algorithm will ever take. Worst-case analysis requires that we identify the input values that cause an algorithm to do the most work. For searching algorithms, the worst case is one where the value is in the last place we check or is not in the list. This could involve comparing the key to each list value for a total of N comparisons. The worst case gives us an upper bound on how slowly parts of our programs may work based on our algorithm choices.

Average Case

Average-case analysis is the toughest to do because there are a lot of details involved. The basic process begins by determining the number of different groups into which all possible input sets can be divided. The second step is to determine the probability that the input will come from each of these groups.

The third step is to determine how long the algorithm will run for each of these groups. All of the input in each group should take the same amount of time, and if they do not, the group must be split into two separate groups. When all of this has been done, the average case time is given by the following formula:

$$\boldsymbol{A(n)} = \sum_{i=1}^{m} \boldsymbol{pi} \ * \ \boldsymbol{ti}$$

where *n* is the size of the input, *m* is the number of groups, *pi* is the probability that the input will be from group *i*, and *ti* is the time that the algorithm takes for input from group *i*.

In some cases, we will consider that each of the input groups has equal probabilities. In other words, if there are five input groups, the chance the input will be in group 1 is the same as the chance for group 2, and so on.

This would mean that for these five groups all probabilities would be 0.2. We could calculate the average case by the above formula, or we could note that the following simplified formula is equivalent in the case where all groups are equally probable:

$$\mathbf{A(n)} = \frac{1}{m}\sum_{i=1}^{m} ti$$

Mathematical Background

These are few mathematical concepts that will be used throughout this course.

The floor and ceiling of a number:

We say that the floor of *X* (written $\lfloor X \rfloor$) is the largest integer that is less than or equal to *X*. So, the $\lfloor 2.5 \rfloor$ would be 2 and $\lfloor 7.3 \rfloor$ would be 8. We say that the ceiling of *X* (written $\lceil X \rceil$) is the smallest integer that is greater than or equal to *X*. So, $\lceil 2.5 \rceil$ would be 3 and $\lceil 7.3 \rceil$ would be 7. Because we will be using just positive numbers, you can think of the floor as truncation and the ceiling as rounding up. For negative numbers, the effect is reversed.

The floor and ceiling will be used when we need to determine how many times something is done, and the value depends on some fraction of the items it is done to. For example, if we compare a set of *N* values in pairs, where the first value is compared to the second, the third to the fourth, and so on, the number of comparisons will be $\lfloor N / 2 \rfloor$. If *N* is 10, we will do five comparisons of pairs and $\lfloor 10 / 2 \rfloor = \lfloor 5 \rfloor = 5$. If *N* is 11, we will still do five comparisons of pairs and $\lfloor 11 / 2 \rfloor = \lfloor 5.5 \rfloor = 5$.

The factorial of the number *N*, written *N!*, is the product of all of the numbers between 1 and *N*. For example, 3! is 3 * 2 * 1, or 6, and 6! is 6 * 5 * 4 * 3 * 2 * 1, or 720. You can see that the factorial gets large very quickly. We will look at this more closely in Section 1.4.

Logarithms

Because logarithms will play an important role in our analysis, there are a few properties that must be discussed. The logarithm base *y* of a number *x* is the power of *y* that will produce the number *x*. So, the $\mathbf{\log_{10} 45}$ is about 1.653 because $\mathbf{10^{1.653}}$ is 45. The base of a logarithm can be any number, but we will typically use either base 10 or base 2 in our analysis. We will use **log** as short hand for $\mathbf{\log_{10}}$ and lg as shorthand for $\mathbf{\log_2}$.

Logarithms are a strictly increasing function. This means that given two numbers *X* and *Y*, if $\boldsymbol{X > Y}$**,** then $\mathbf{\log_B} \boldsymbol{X} > \mathbf{\log_B} \boldsymbol{Y}$ for all bases B. Logarithms are one-to-one functions. This means that if $\mathbf{\log_B} \boldsymbol{X} = \mathbf{\log_B} \boldsymbol{Y}$, the $\boldsymbol{X = Y}$. Other properties that are important for you to know are:

1. $\log_B 1 = 0$ 2. $\log_B B = 1$ 3. $\log_B X * Y = \log_B X + \log_B Y$

4. $\log_B X^Y = Y * \log_B X$ 5. $\log_A X = \frac{\log_B X}{\log_B A}$

These properties can be combined to help simplify a function. Equation 5 is a good fact to know for base conversion. Most calculators do $\mathbf{\log_{10}}$

and natural **logs**. For example, if we need to know **$\log_{42}$ 75**. Equation 5 would help you find the answer.

Probabilities

Because we will analyze algorithms relative to their input, we may at times need to consider the likelihood of a certain set of input. This means that we will need to work with the probability that the input will meet some condition. The probability that something will occur is given as a number in the range of 0 to 1, where 0 means it will never occur and 1 means it will always occur. If we know that there are exactly 10 different possible inputs, we can say that the probability of each of these is between 0 and 1 and that the total of all of the individual probabilities is 1, because one of these must happen. If there is an equal chance that any of these can occur, each will have a probability of 0.1 (one out of 10, or 1/10).

For most of our analyses, we will first determine how many possible situations there are and then assume that all are equally likely. If we determine that there are N possible situations, this lead to a probability of 1 / N for each of these situations.

Summations

We will be adding up sets of values as we analyze our algorithms. Let’s say we have an algorithm with a loop. We notice that when the loop variable is 5, we do 5 steps and when it is 20, we do 20 steps. We determine in general that when the loop variable is *M*, we do *M* steps. Overall, the loop variable will take on all values from 1 to *N*, so the total of the steps is the sum of the values from 1 through *N*. To easily express this, we use the equation $\boldsymbol{\sum_{i=0}^{n} i}$. The expression below the Σ represents the initial value for the summation variable, and the value above the Σ represents the ending value. You should see how this expression corresponds to the sum we are looking for.

Once we have expressed some solution in terms of this summation notation, we will want to simplify this so that we can make comparisons with other formulas. Deciding whether $\sum_{i=11}^{N} i^2 - i$ or $\sum_{i=0}^{N} i^2$- 2i is greater would be difficult to do by inspection, so we use the following set of standard summation formulas to determine the actual values these summations represent.

1. $\sum_{i=1}^{n} C * i$ = C* $\sum_{i=1}^{n} i$ with *C* a constant expression not dependent on $\boldsymbol{i}$.

2. $\sum_{i=C}^{n} i$ = $\sum_{i=0}^{n-c} i + C$

3. $\sum_{i=C}^{n} i$ = $\sum_{i=0}^{n} i - \sum_{i=0}^{C-1} i$

4. $\sum_{i=1}^{n}(A + B)$ = $\sum_{i=1}^{n} A + \sum_{i=1}^{n} B$

5. $\sum_{i=0}^{n}(n-i) = \sum_{i=0}^{n} i$

6. $\sum_{i=1}^{n} 1$ = n

7. $\sum_{i=1}^{n} C$ = C*n

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

9. $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{2n^3+3n^2+n}{6}$

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

11. $\sum_{i=1}^{n} A^i = \frac{A^{n+1}-1}{A-1}$

Rate of Growth

In analysis of algorithms, it is not important to know exactly how many operations an algorithm does. Our greater concern is the rate of increase in operations for an algorithm to solve a problem as the size of the problem increases. This is referred to as the rate of growth of the algorithm.

What happens with small sets of input data is not as interesting as what happens when the data set gets large, because we are interested in general behavior. We just look at the overall growth rate of algorithms, not at the details. For example, a function based on x^2 increases slowly at first, but as the problem size gets larger, it begins to grow at a rapid rate while, functions that are based on *x* both grow at a steady rate for the entire length of the graph. The function based on log *x* seems to not grow at all, but this is because it is actually growing at a very slow rate. The relative height of the functions is also different when we have small values versus large ones. Consider the value of the functions when *x* is 2. At that point, the function with the smallest value is *x*2 / 8 and the one with the largest value is *x* + 10. We can see, however, that as the value of *x* gets large, *x*2 / 8 becomes and stays the function with the largest value.

Putting all of these together means that as we analyze algorithms, we will be interested in which rate of growth class an algorithm falls into rather than trying to find out exactly how many of each operation are done by the algorithm. When we consider the relative "size" of a function, we will do so for large values of *x*, not small ones.

Classification of Growth

Because the rate of growth of an algorithm is important, and we have seen that the rate of growth is dominated by the largest term in an equation, we will discard the terms that grow more slowly. When we strip all of these things away, we are left with what we call the *order* of the function or related algorithm. We can then group algorithms together based on their order. We group them in three categories—those that grow at least as fast as some function, those that grow at the same rate, and those that grow no faster.

Big Omega

We use $\Omega(f)$, called *big omega*, to represent the class of functions that grow at least as fast as the function f. This means that for all values of n greater than some threshold $n0$, all of the functions in $\Omega(f)$ have values that are at least as large as f. You can view $\Omega(f)$ as setting a lower bound on a function, because all the functions in this class will grow as fast as f or even faster. Formally, this means that if $g(x) \in \Omega(f)$, $g(n) \geq cf(n)$ for all $n \geq n0$ (where c is a positive constant).

Because we are interested in efficiency, $\Omega(f)$ will not be of much interest to us because $\Omega(n2)$, for example, includes all functions that grow faster than $n2$ including $n3$ and $2n$.

Big Oh

At the other end of the spectrum, we have $O(f)$, called *big oh*, which represents the class of functions that grow no faster than f. This means that for all values of n greater than some threshold $n0$, all of the functions in $O(f)$ have values that are no greater than f. The class $O(f)$ has f as an upper bound, so none of the functions in this class grow faster than f. Formally this means that if $g(x) \in O(f)$, $g(n) \leq cf(n)$ for all $n \geq n0$ (where c is a positive constant).

This is the class that will be of the greatest interest to us. Considering two algorithms, we will want to know if the function categorizing the behavior of the first is in big oh of the second. If so, we know that the second algorithm does no better than the first in solving the problem.

Big Theta

We use θ(f), called *big theta*, to represent the class of functions that grow at the same rate as the function *f*. This means that for all values of *n* greater than some threshold *n*0, all of the functions in θ(*f*) have values that are about the same as *f*. Formally, this class of functions is defined as the place where big omega and big oh overlap, so $\theta(f) = \Omega(f) \cap O(f)$. When we consider algorithms, we will be interested in finding algorithms that might do better than the one we are considering. So, finding one that is in big theta (in other words, is of the same complexity) is not very interesting. We will not refer to this class very often.

Finding Big Oh

We can find if a function is in *O*(*f*), by using the following formal description above or by using the following alternative description:

$$\mathbf{g} \in \mathbf{f(n)} \text{ If } \lim_{n\to\infty} \frac{g(n)}{f(n)} = C \quad \text{for some } r \in R^{*}$$

This means that if the limit of *g*(*n*) / *f*(*n*) is some real number less than ,*g* is in *O*(*f*). With some functions, it might not be obvious that this is the case. We can then take the derivative of *f* and *g* and apply this same limit.

Notation

Because $\theta(f)$, $\Omega(f)$, and $O(f)$ are sets, it is proper to say that a function *g* is an element of these sets. The analysis literature, however, accepts that a function *g* is equal to these sets as being equivalent to being a member of the set. So, when you see $g = O(f)$, this really means that $g \in O(f)$.

Algorithm Analysis

Let's say that we have a large, complex program that takes longer to run than we want. How can we identify parts of this program that with fine-tuning could improve the overall speed?

We could look at the program and find the subprograms (sometimes called subroutines, procedures, or functions) that have many calculations or loops and work on improving those. After a lot of effort to do this, we might not see an impact because the subprograms we worked on may not be used very much.

A better technique would be to first identify the subprograms that are used a lot and then improve those. One way to do this would be to create a set of global counters, one for each subprogram. They are initialized to zero at the start of the program. Every subprogram is then altered to increment one of those counters as its new first statement. Each time that subprogram is entered, it will now increase its counter by 1, so at the end of the program, our set of counters will tell us how many times each subprogram was called. We can now see which are called many times and which are called just a few.

Suppose we have a program where one simple subprogram is called 50,000 times and a bunch of complex subprograms are called just once each. We would have to reduce the complex subprograms by 50,000 operations to have the same effect as reducing the simple subprogram by just one operation. You should see that finding a simple improvement in one subprogram is much easier than finding 50,000 in a group of subprograms.

The technique of using counters can be applied at the subprogram level as well. In this case, we create a set of global counters, one for each of the significant points we want to know about. Suppose we wanted to know how many times the then and else parts of an “if “statements are executed. We could create two counters and increment the first inside the then part and increment the other inside the else part. At the end of the program, these two counters would tell us the information we are interested in. Other significant points we might want to test would be inside loops and just before conditional statements. More generally, we would place increment statements at any place where control can be transferred.

At the end of a program, these counters would tell us how many times each of the blocks of code in a subprogram was executed. We can then look at those parts where the most work is done for improvements.

This process is important, and many computers and software development systems have program profiling tools that will produce this information for you automatically.

Searching and Selection Algorithm

The act of searching for a piece of information in a list is one of the fundamental algorithms in computer science. In discussing searching, we assume that there is a list that contains records of information, which in practice is stored using an array in a program. The records, or items, are assumed to be in adjacent locations in the list, with no gaps or blank records in the middle. The list locations will be indexed from 1 to *N*, which represents the number of records in the list. Each record can be separated into fields, but we will only be interested in one of these fields, which we will call the key. Lists will be either unsorted or sorted based on their key value.

Records are in a random order in an unsorted list and are in order by increasing key value in a sorted list. When a list is unsorted, we only have one search option and that is to sequentially look through the list for the item we want. This is the simplest of the searching algorithms.

We will see that this algorithm is not very efficient but will successfully search in any list.
When a list of elements is sorted, the options for searching are expanded to include binary search. Binary search takes advantage of the ordered nature of the list to eliminate more than one element of the list with each comparison. This results in a more efficient search.

A problem related to searching for a particular value is the selection problem, where we are interested in finding the element that meets some criterion. It might be that we are looking for the fifth largest value, seventh smallest value, or the median value in the list. We will look at two techniques that can be used to solve this problem.

Sequential Searching

In searching algorithms, we are concerned with the process of looking through a list to find a particular element, called the target. Although not required, we usually consider the list to be unsorted when doing a sequential search, because there are other algorithms that perform better on sorted lists. Search algorithms are not just interested in whether the target is in the list but are usually part of a larger process that needs the data associated with that key. For example, the key value might be an employee number, a serial number, or other unique identifier. When the proper key is found, the program might change some of the data stored for that key or might simply output the record. In any case, the important task for a search algorithm is to identify the location of the key. For this reason, search algorithms return the index of where the record with the key is located. If the target value is not found, it is typical for the algorithm to return an index value that is outside the range of the list of elements.

For our purposes, we will assume that the elements of the list are located in positions 1 to *N* in the list. This allows us to return a value of zero if the target is not in the list. For the sake of simplicity, we will assume that the key values are unique for all of the elements in the list.

Sequential search looks at elements, one at a time, from the first in the list until a match for the target is found. It should be obvious that the further down the list a particular key value is, the longer it will take to find that key value. This is an important fact to remember when we begin to analyze sequential search.

The complete algorithm for sequential search is

```
SequentialSearch( list, target, N )
List =  the elements to be searched
Target =  the value being searched for
N =  the number of elements in the list
for i = 1 to N do
        if (target = list[i])
        return i
end if
```

```
    end for
    return 0
```

Worst case analysis

There are two worst cases for the sequential search algorithm. The first is if the target matches the last element in the list. The second is if the target is not in the list. For both of these cases, let’s look at how many comparisons are done.

We have said that all of the list keys will be unique, and so if the match is in the last location, that means that all of the other locations are different from the target. The algorithm will, therefore, compare the target with each of these values until it finds the match in the last location. This will take N comparisons, where N is the number of elements in the list.

We will have to compare the target to all of the elements in the list to determine that the target is not there. If we skip any of the elements, we will not know if the target is not present or is present in one of the locations we skipped. This means that we need to do N comparisons to see that none of the elements match the target. In both cases, whether the target is in the last location or not in the list, this algorithm takes N comparisons. You should see that this is the upper bound for any search algorithm, because to do more than N comparisons would mean that the algorithm compared at least one element with the target at least twice, which is unnecessary work, so the algorithm could be improved.

There is a difference between the concept of an upper bound and a worst case. The upper bound is a concept based on the problem to be solved, and the worst case is based on the way a particular algorithm solves that problem. For this algorithm, the worst case is also the upper bound for the problem. We will later see another search algorithm that has a worst case that is less than this upper bound of *N*.

Average-Case Analysis

There are two average-case analyses that can be done for a search algorithm.

The first assumes that the search is always successful and the other assumes that the target will sometimes not be found.

If the target is in the list, there are *N* places where that target can be located.

It could be in the first, second, third, fourth, and so on, locations in the list. We will assume that all of these possibilities are equally likely, giving a probability of 1/*N* for each potential location.

Take a moment to answer the following questions before reading on:

- How many comparisons are done if the match is in the first location?
- What about the second location?

- What about the third?
- What about the last or *N*th location?

If you looked at the algorithm carefully, you should have determined that the answers to these questions are 1, 2, 3, and *N*, respectively. This means that for each of our *N* cases, the number of comparisons is the same as the location where the match occurs. This gives the following equation for this average case:

$$\text{A(N)} = \frac{1}{N}\sum_{i=1}^{N} i \;=\; \frac{1}{N} * \frac{N(N+1)}{N} = \frac{N+1}{2}$$

If we include the possibility that the target is not in the list, we will find that there are now *N* + 1 possibilities. As we have seen, the case where the target is not in the list will take *N* comparisons. If we assume that all *N* + 1 possibilities are equally likely, we wind up with the following:

$$\text{A(N)} = \frac{1}{N+1} * \left[\left(\sum_{i=1}^{N} i\right) * N\right] = \left(\frac{1}{N+1}\sum_{i=1}^{N} i\right) + \left(\frac{1}{N+1} * N\right)$$

$$= \left(\frac{1}{N+1} * \frac{N(N+1)}{2}\right) + \frac{N}{N+1} = \frac{N}{2} + \frac{N}{N+1} = \frac{N}{2} + 1 - \frac{1}{N+1} = \frac{N+2}{2}$$

As $N\ get\ very\ large$ $\frac{1}{N+1}$ becomes 0

We see that including the possibility of the target not being in the list only increases the average case by 1/2. When we consider this amount relative to the size of the list, which could be very large, this 1/2 is not significant.

Exercises

1. Sequential search can also be used for a sorted list. Write an algorithm called SortedSequentialSearch that will return the same results as the algorithm above but will run more quickly because it can stop with a failure the minute it finds that the target is smaller than the current list value. When you write your algorithm, use the Compare(x, y) function defined as: compare (x, y) = $\begin{cases} -1\ i\ x < y \\ 0\ if\ = y \\ 1\ i\ x > y \end{cases}$

   The Compare function should be counted as one comparison and can best be used in a switch. Do an analysis of the worst case, average case with the target found, and average with the target not found. (*Note:* This last analysis has many possibilities because of all of the additional early exits when the target is smaller than the current value.)

2. What is the average complexity of sequential search if there is a 0.25 chance that the target will not be found in the list and there is a 0.75 chance that when the target is in the list, it will be found in the first half of the list?

Selection algorithm

There are situations where we are interested in finding an element in a list that has a particular property, instead of a particular value. In other words, instead of finding a record with a particular key value, we may be interested in the record with the largest, smallest, or median value. More generally, we want to find the *K*th largest value in the list.

One way to accomplish this would be to sort the list in decreasing order, and then the *K*th largest value will be in position *K*. This is a lot more work than we need to do, because we don’t really care about the values that are smaller than the one we want. A related technique to this would be to find the largest value and then move it to the last location in the list. If we again look for the largest value in the list, ignoring the value we already found, we get the second largest value, which can be moved to the second last location in the list.

If we continue this process, we will find the *K*th largest value on the *K*th pass. This gives the algorithm

```
FindKthLargest( list, N, K )
list: the values to look through
N: the size of the list
K: the element to select
for i = 1 to K do
    largest = list[1]
    largestLocation = 1
for j = 2 to N-(i-1) do
    if list[j] > largest then
```

```
            largest = list[j]
             largestLocation = j
            end if
        end for
        Swap( list[N-(i-1)], list[largestLocation] )
        end for
        return largest
```

What is the complexity of this process? On each pass, we initialize largest to the first element of the list, and then we compare largest to every other element. On the first pass, we do *N* - 1 comparisons. On the second pass, we do *N* - 2 comparisons. On the *K*th pass, we do *N* - *K* comparisons. So, to find the *K*th largest element, we will do

$\sum_{i=1}^{k} N - i \ = N * k - \frac{K(K-1)}{2}$ comparisons, which is *O*(*K* * *N*).

Sorting Algorithms

Insertion sort

The basic idea of insertion sort is that if you have a list that is sorted and need to add a new element, the most efficient process is to put that new element into the correct position instead of adding it anywhere and then resorting the entire list. Insertion sort accomplishes its task by considering that the first element of any list is always a sorted list of size 1. We can create a two-element sorted list by correctly inserting the second element of the list into the one element list containing the first element. We can now insert the third element of the list into the two-element sorted list. This process is repeated until all of the elements have been put into the expanding sorted portion of the list.

The following algorithm carries out this process:

```
InsertionSort( list, N )
list the elements to be put into order
N the number of elements in the list
for i = 2 to N do
newElement = list[ i ]
location = i - 1
while (location ≥ 1) and (list[ location ] > newElement) do
// move any larger elements out of the way
list[ location + 1] = list[ location ]
location = location - 1
end while
list[ location + 1 ] = newElement
end for
```

This algorithm copies the next value to be inserted into **newElement**. It then makes space for this new element by moving all elements that are larger one position over in the array. This is done by the while loop. The last copy of the loop moved the element in position location + 1 to position location + 2. This means that position location + 1 is available for the “new” element.

Worst-Case Analysis

When we look at the inner while loop, we see that the most work this loop will do is if the new element to be added is smaller than all of the elements already in the sorted part of the list. In this situation, the loop will stop when location becomes 0. So, the most work the entire algorithm will do is in the case where every new element is added to the front of the list. For this to happen, the list must be in decreasing order when we start. This is a worst-case input, but there are others.

Let’s look at how this input set will be handled. The first element to be inserted is the one in the second location of the list. This is compared to one other element at most. The second element inserted (that’s at location 3) will be compared to the two previous elements, and the third element inserted will be compared to the three previous elements. In general, the *i*th element inserted will be compared to *i* previous elements. This process is repeated for *N* 1 elements. This means that the worst-case complexity for insertion sort is given by

$$\text{A(N)} = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2} = \frac{N^2-N}{2}$$

$$\text{A(N)} = O(N^2)$$

Average-Case Analysis

Average-case analysis will be a two-step process. We first need to figure out the average number of comparisons needed to move one element into place. We can then determine the overall average number of operations by using the first step result for all of the other elements.

We begin by determining on average how many comparisons it takes to move the *i*th element into position. We said that adding the *i*th element to the sorted part of the list does at most *i* comparisons. It should be obvious that we do at least one comparison even if the element stays in its current position.

How many positions is it possible to move the *i*th element into? Let’s look at small cases and see if we can generalize from there. There are two possible positions for the first element to be added—either in location 1 or location 2.

There are three possible positions for the second element to be added—either in locations 1, 2, or 3. It appears that there are $i + 1$ possible positions for the *i*th element. We consider each of these equally likely.

How many comparisons does it take to get to each of these $i + 1$ possible positions? Again we look at small cases and generalize from there. If we are adding the fourth element, and it goes into location 5, the first comparison will fail. If it goes into location 4, the first comparison will succeed, but the second will fail. If it goes into location 3, the first and second comparisons will succeed, but the third will fail. If it goes into location 2, the first, second, and third comparisons will succeed, but the fourth will fail. If it goes into location 1, the first, second, third, and fourth comparisons will succeed, and there will be no further comparisons because location will have become zero. This seems to imply that for the *i* th element, it will do 1, 2, 3, . . ., *i* comparisons for locations *i* + 1, *i*, *i* 1, . . ., 2, and it will do *i* comparisons for location 1. The average number of comparisons to insert the *i*th element is given by the formula:

$$A_i = \frac{1}{i+1}\left[\left\{\sum_{p=1}^{i} p\right\} + i\right] = \frac{1}{i+1}\left[\frac{i(i+1)}{2} + i\right]$$

$$= \frac{i}{2} + \frac{i}{i+1} = \frac{i}{2} + 1- \frac{1}{i+1}$$

This is just the average amount of work to insert the *i*th element. This now needs to be summed up for each of the 1 through *N*- 1 elements that gets “added” to the list. The final average case result is given by the formula:

$$A(N) = \sum_{i=1}^{N-1} \frac{i}{2} + \sum_{i=1}^{N-1} 1 - \sum_{i=1}^{N-1} \frac{1}{i+1}$$

$$= \sum_{i=1}^{N-1} \frac{1}{i+1} = \sum_{i=2}^{N} \frac{1}{i} = \left(\sum_{i=1}^{N} \frac{1}{i}\right) - 1$$

$$=\frac{1}{2}\left(\frac{(N-1)N}{2}\right) + (N-1) - (lnN - 1)$$

$$= \frac{N^2-N}{4} + (N-1) - (\ln N - 1)$$

$$= \frac{N^2+3N-4}{4} - (\ln N - 1) \qquad \approx \frac{N^2}{4}$$

$$A(N) = O(N^2)$$

Exercises

1. Show the results of each pass of InsertionSort applied to the list [7, 3, 9, 4, 2, 5, 6, 1, 8].
2. Show the results of each pass of InsertionSort applied to the list [3, 5, 2, 9, 8, 1, 6, 4, 7].
3. Section 3.1.1 showed that a list in decreasing order leads to the worst case. This means that the list [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] would give the worst case of 45 comparisons. Find one other list of these 10 elements that will give the worst-case behavior. What more can you say generally about the
class of input that generates the worst case for this algorithm.
4. When you look closely at the InsertionSort algorithm, you will notice that the insertion of a new element basically does a sequential search for the new location. We saw that binary searches are much faster.

Consider a variation on insertion sort that does a binary search to find the correct position to insert this new element. You should notice that for unique keys the standard binary search would always return a failure. So for this problem, assume a revised binary search that returns the location at which this key belongs.

a. What is the worst-case analysis of this new insertion sort?

b. What is the average-case analysis of this new insertion sort?