Introduction and Big-Oh

Topics

I. Why Study Data Structures?

Before you begin to study data structures, the first question to ask yourself is why bother studying this topic? After all, most modern programming languages come with all the data structures a programmer will ever use. Java comes with a nearly complete set of data structures in its java.util.Collections classes, and what is not in Java's Collections classes can normally be found in other standard libraries, such as the one maintained by the Apache project (www.apache.org) or Source Forge (www.sourceforge.net). So it would seem that there is no need for programmers ever to implement their own data structures.

This argument misses a number of important points. The most obvious is that not all the possible data structures that can be implemented have, in fact, been implemented. Many problems that need to be solved have some unique aspects that allow slight modifications of a data structure so that it is not generic, but the modified data structure will have significant advantages (in performance or space) over its more generic counterpart. In addition, as will be shown through many examples in these modules, there are times that the standard data structures can be combined to create solutions that have great advantages over the use of more-generic data structures. It is only in understanding how these data structures are implemented that a programmer can understand these trade-offs and make the appropriate modifications to produce better programs.

The second fallacy of the argument is that often, the study of data structures is not limited to how to implement the data structure, but includes discussion of ways the data structure can be used in an algorithm to solve a problem. Furthermore, the solutions to one set of problems (such as building an expression tree, as we will discuss in module 2) can often be used as a template for a solution of another problem (for example, building a Huffman tree). Only by studying some common algorithms that use these data structures will you as students learn how to apply these data structures to algorithms in your own unique problems.

The third reason to study data structures is that doing so provides an easy way to judge just how efficient a program implementation is. For example, there was once a program that was run daily to process 10,000 items. This program took 18 hours to run each day. The program was fairly short and easy to evaluate. A quick evaluation of the program showed that amount of time the program took to run was proportional to the square of the number of inputs, or N^2 where N is the number of input items: 10,000 in this case. On a close examination it was found that a simple change to the algorithm could result in a run time proportional to only ($N \log_2 N$). This implied that the original algorithm would take 100,000,000 time units for an input of 10,000 items, but with the change, it would take only about 140,000 time units. Thus, this simple change would result in the program taking one-seventieth of the time; the program would go from running in 18 hours to running in about 15 minutes. The change was made, and the reduction in time from 18 hours to a few minutes was the observed result.

As this program represented a significant bottleneck on daily work load for the overall system, this was a big improvement to the ability to process work through the whole system. The change itself was minor, and the likely reason it had not been made earlier was that the programmer did not understand the implications of a decision he or she had made about how to process the data. Unfortunately, this is not an isolated incident, and most good programmers can give many examples of solving such problems. Understanding the data structures and the implications of their use can often result in huge savings of time and/or space in solving a problem.

The purpose of this course, then, is to study data structures as a means of understanding how to implement better programs. Our discussions will not be limited to how the data structures are implemented, though that is an important part of the material covered. We will also discuss examples of how to use these data structures, and how to evaluate the efficiency of a program using these data structures. The goal of this course is, in essence, to help you become better programmers.

II. Review of Algebra and Big-Oh Notation

This module will begin by reviewing Big-Oh notation, which you studied in CMSC 230 or an equivalent course. We begin with an explanation of Big-Oh and why it is useful. We will then apply the meaning of Big-Oh as we discuss the types of problems that can be solved on a computer. Our first focus will be on problems that cannot be solved algorithmically and thus cannot be programmed to be solved on a computer. Then we'll move onto problems that have a Big-Oh run-time growth so high that they require an unreasonable amount of time to solve on a computer, very often for even reasonable-sized inputs. Finally, we will examine a subset of these problems, those that have a Big-Oh growth rate that is slow enough that they can be solved in a reasonable time for even large numbers of inputs.

This section will also cover some of the algebraic relationships that are needed to perform Big-Oh calculations, and the implications of these relationships.

A. Understanding Big-Oh

Big-Oh, also called *order of*, is an estimate of the increase in CPU execution time, memory, clock time (the amount of time user has to wait), or other resource used by an algorithm as the size of the input to the algorithm grows. Big-Oh uses a simple function to characterize the relationship between the number of input data elements (the size of the input to the algorithm) and the resource utilization of an algorithm. Thus $O(f(N))$, which is read as "Big-Oh of $f(N)$," says that the expected growth rate for the time or space requirements of an algorithm for large values of N is roughly the function $f(N)$.

The growth rate can be for the best case, the average case, or the worst case, since in some algorithms these can be very different. For example, in the sorting covered in CMSC 230, the selection sort had an average case and worst case of $O(N^2)$, but a best case (when it operated on already sorted data) of $O(N)$. In predicting the behavior of a program, what is of most interest is how large the resource utilization will eventually become. So in these modules, Big-Oh will always be the worst-case growth rate, unless otherwise specified.

As an example, on an algorithm with a Big-Oh of $O(N)$, for large values of N, we would expect that the growth rate in time and/or space will be linearly proportional to N. The following table gives example growth rates as a problem grows for $O(f(N))$, where $f(N) = \log_{10} N$, N, $N \log_{10}N$, N^2, N^3, 2^N, and $N!$. Note how slowly logarithmic values grow, and how quickly 2^N and $N!$ grow.

	N=10	N=50	N=100	N=1,000	N=10,000	N=100,000	N=1,000,000
$O(\log_{10}N)$	1	1.7	2	3	4	5	6
$O(N)$	10	50	100	1,000	10,000	100,000	1,000,000
$O(N \log_{10}N)$	10	85	200	3,000	40,000	500,000	6,000,000
$O(N^2)$	100	2500	10,000	1,000,000	100,000,000	10^{10}	10^{12}
$O(N^3)$	1000	125,000	10^6	10^9	10^{12}	10^{15}	10^{18}
$O(2^N)$	1024	$1.12 * 10^{15}$	10^{30}	$> 10^{99}$	$>> 10^{99}$	$>>> 10^{99}$	$>>>> 10^{99}$
$O(N!)$	3 million	$3 * 10^{64}$	$> 10^{99}$	$>> 10^{99}$	$>>> 10^{99}$	$>>>> 10^{99}$	$>>>>> 10^{99}$

When performing Big-Oh analysis, only large values of N are generally considered. Thus, Big-Oh is a bound on the run-time growth of a program, and it is only a rough measure of that growth. For example, consider a program which has some startup time when it starts to run, 50 time units, plus two loops, one that is $O(N)$ and another that is $O(N^2)$. In this program, the linear run-time loop takes 15 units of time for each data item processed, and the quadratic run-time loop takes 1 unit of time for each data item. This program would have a run time represented by the following equation:

Run time = $50 + 15N + (N^2)$

A graph of this function would appear as follows:

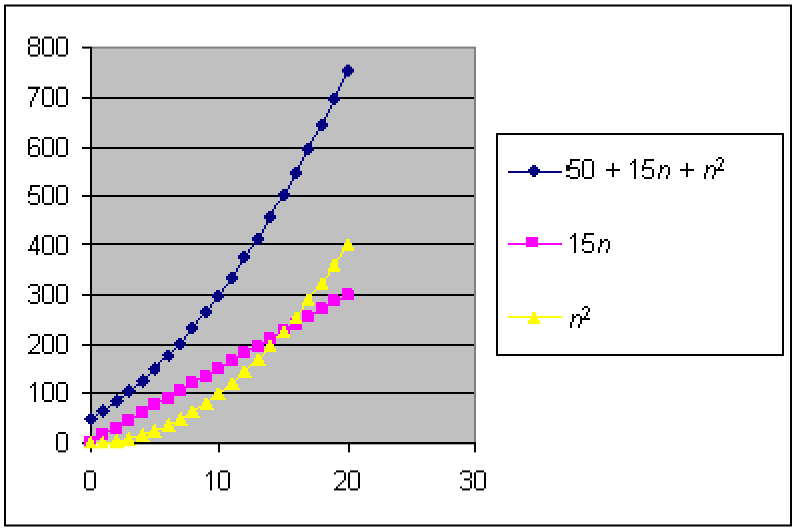

Figure 1.II.1 – Graph of run-time growth of hypothetical program

While the Big-Oh for this program is O(N^2), the actual run time of the program does not correspond to this growth rate for small values of N. There is an initial amount of startup time (in this case 50), and after that the growth rate corresponds for some time to the linear portion of the equation, as it is much larger than the polynomial and tends to dominate for small values of N. Eventually, however, the O(N^2) term becomes large and begins to dominate. This illustrates the point that Big-Oh is not a function that characterizes the run time of the program, but an estimate of its growth as the data size increases.

Remember that Big-Oh applies only to large values of N. This allows the function used by the Big-Oh analysis to be greatly simplified. For example, consider a program that has the following $T(N)$, where $T(N)$ is the execution time of the program.

$T(N) = 50 + 15N + 3N^2$

For large values of N, the term $3N^2$ will dominate all other terms in this equation. Thus, the constant term, 50, and the linear term, $15N$, are simply dropped. This leaves only the term $3N^2$, so the Big-Oh of this algorithm is O($3N^2$). There is, however, one more change that can be made to this equation. Since Big-Oh evaluates run-time growth—not absolute times—what we are interested in is how much more time the algorithm takes on an input data set of size n_2 than on an input data set of size n_1. This can be written as follows:

$$\frac{3n_1^{\ 2}}{3n_2^{\ 2}} = \frac{n_1^{\ 2}}{n_2^{\ 2}}$$

In this equation the constant 3 is canceled out. Constants are always dropped in Big-Oh analysis, since they will appear in the numerator and denominator and thus cancel each other out, and the Big-Oh of the equation $T(N) = 50 + 15N + 3N^2$ simplifies to O(N^2).

The following example applies Big-Oh to a program. Consider the following method that sums the values of an array:

```
public static int sumArray(int array[]) {
    int total = 0;
    for (int i = 0; i < array.length; i++) {
        total = total + array[i];
    }
    return total;
}
```

In this method, the size of the problem to be solved grows as the size of the input array grows. For large values of *N*, the growth of the run time for the method is expected to grow (roughly) as a linear function of the number of elements in the array, or O(*N*). Thus, the growth in the size of the problem to be solved is proportional to the growth in the size of the input data set, which is the array. It is often, but not always, the case that the size of the problem is some input data set size, such as a file or array, to be processed.

The problem size is not always the size of an input data set. For example, consider the following method that calculates a Fibonacci number:

```
public static int calcFibLinear(int number) {
     int fibonacci = 0;
     if (number < 0) return 0;
     if (number == 0) return 0;
     if (number == 1) return 1;

     int oldest = 0;
     int last = 1;
     for (int i = 2; i <= number; i++) {
         fibonacci = oldest + last;
         oldest = last;
         last = fibonacci;
     }
     return fibonacci;
 }
```

In this case, the growth of the run time of this problem grows (roughly) as a linear function of the size of the Fibonacci number to be calculated, even though the input consists of only one number.

Before proceeding, it is important to consider a number of points about the use of Big-Oh. First Big-Oh is defined only for large values of *N*. If you have a problem in which you know *N* will always be small and thus the run time for all practical cases of the program will be small, Big-Oh analysis is not relevant.

Second, Big-Oh is never an absolute measure of the run time of a program. Many things go into calculating the run time of a program, most of which are highly dependent on the particular computer it is run on. These factors will vary widely from computer to computer or even between two runs of a particular program on the same computer. Sometimes large data sets will require operations, such as paging to a disk, that can cause singularities in the run time and cause the algorithm to suddenly require an order-of-magnitude growth in a resource such as clock time. Thus, in order to get some idea of how long a particular program will take on a given computer, it is often necessary to benchmark the performance of a program by actually running it on the given hardware. Once the program has been benchmarked on a particular piece of hardware, Big-Oh evaluation will provide an idea of how long the program will take when the size of the program input increases.

Third, Big-Oh is concerned only with the worst case for an algorithm. If an algorithm has an average growth rate that is much better than its Big-Oh growth rate, the Big-Oh might not be important.

Finally, the fact that an algorithm has a reasonable growth rate does not mean that the algorithm is reasonable. If an online transaction system that is O(N) takes 2 seconds for 10 data items, it is probably unreasonable to expect a user to wait 2000 seconds (about 33 minutes) when there are 10,000 data items. Even though an O(N) is generally considered a good growth rate, in some instances, it is not acceptable.

It might seem that if Big-Oh cannot be used to give the actual run time of a program, it is not useful. Although Big-Oh does not give an absolute run-time performance, it is still very important. Many projects have been doomed to failure by programmers who benchmark a program with a small set of test data, only to find that the performance is inadequate when the program is scaled up for production.

B. What Can a Computer Calculate?

If the typical person on the street is asked what problems a computer can solve, a common answer would be that a computer can calculate solutions to any problem. This is a common misconception. The truth is that computers are extremely limited in the types of problems which they can solve. Of the three major types of problems—undecidable, intractable, and tractable—only the last can be solved for reasonable-sized data sets on what is typically thought of as a computer.

Intractable problems can be solved on a computer as long as the input size is limited. For example, the recursive Fibonacci program presented later in the module can be solved for inputs up to 40 in a reasonable time. The problem is that for an input size of 50 the problem takes hours to solve on a typical current PC, and for an input size of 60 the calculation will take days. Often, adding one or two values to an intractable problem changes the calculation time from seconds to days.

If the input size of the problem is well defined and does not grow, a tractable problem that runs in a reasonable time can be run on a computer. Normal programs, however, change the input size of the problem, and intractable problems generally grow very fast for even small input data sets and are thus considered not solvable on a computer.

So what are undecidable, intractable, and tractable problems? An undecidable problem is one for which there is no algorithmic solution. The classic example of this is Turing's Halting Problem. While the details of this problem are beyond this course, what Turing did was, in essence, show that a problem exists for which no algorithm can produce a solution to the problem. While this may seem arcane, it turns out that many interesting problems correspond to this problem and are thus undecidable. This type of problem, while important, is not covered in this course.

Intractable problems are those that have an algorithm solution (in other words, a program can be written to solve them), but the run time for the solution grows so quickly that as the size of input set grows it is impossible to create a computer that can solve the problem in a reasonable amount of time. Moore's Law, which posits that computers double in computing power every 18 months, does not significantly impact the fact that intractable problems are too large to calculate on a computer. Intractable problems will be covered in the next section of this module.

Finally, there is the class of problems known as tractable. A tractable problem is one that can be solved in a reasonable time on computer; as you might expect, this is the type of problem we will be dealing with in this course. Tractable problems are at the heart of a programming course, since they are problems which can be solved programmatically. The other types of problems, particularly intractable problems, are discussed mainly in order to help you recognize when a problem is intractable, so you can then attempt to find another solution.

C. Exponential Growth (Intractable Problems)

The reason some problems are intractable is that their growth rates are so high that the growth of execution time is so rapid that the problem takes an unreasonably long time to run on a computer for even small increases in input size. The following Java program, which recursively calculates a Fibonacci number, illustrates this point. To run the program, on the command line you would enter the ordinal value of the Fibonacci number to be calculated; for example, "Java Fibonacci 7" would return the seventh number in the Fibonacci series. The program will calculate and print the value, as well as the time (in milliseconds) required for the calculation.

```
/**
 * Purpose: This program calculates and times the calculation for
            a Fibonacci number.
 */
public class Fibonacci {
  public static int fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
  }

  public static void main(String[] args) {
    int input = 0;
    if (args.length != 1) {
        System.out.println("You must enter a number");
        System.out.println("For example: java Fibonacci 7");
        System.exit(1);
    }
    try {
      input = Integer.parseInt(args[0]);
    } catch(Exception e) {
      System.out.println ("Input value must be an integer");
      System.exit(1);
    }

    long startTime = System.currentTimeMillis();
    System.out.println("The Fibonacci number for " + input +
                       " is " + fibonacci(input));
    long endTime = System.currentTimeMillis();
    System.out.println("Total time used = " +
                      (endTime - startTime) +
                       " milliseconds");
    }
}
```

Run this program for values such as 1, 2, 5, 10, 20, 30, 35, 40, 45, and 48. (48 was the largest value that would run in a reasonable time on a typical circa 2005 computer running Windows and using the Java JDK 1.5-4). When the program is run for small values, it takes practically no time to run. However, as the Fibonacci number to be calculated increases, the amount of time needed grows very rapidly. The following graph shows a typical growth rate for running this program.

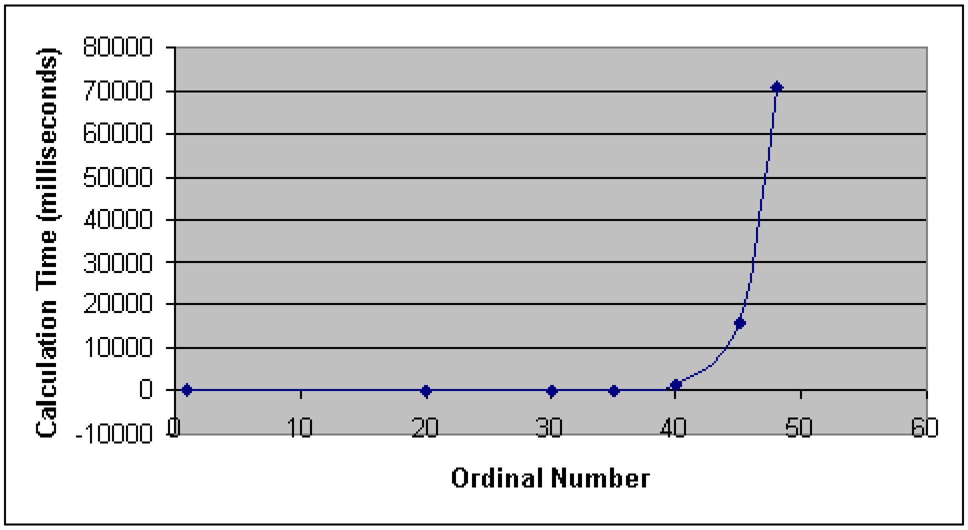

Figure 1.II.2 – Graph of run-time growth of Fibonacci program

From this graph, it is obvious that as the input value to the Fibonacci method increases, the run time grows very rapidly until it simply takes too long to run on a computer. Calculating the sixtieth Fibonacci number could take days or longer on a typical computer.

The growth rate for the recursive Fibonacci method illustrates the truth of a saying, widely attributed to Albert Einstein, that the most powerful force in the universe is compound interest. It shows why problems with exponential run-time growth are intractable and cannot be solved on a computer. As the size of the problem grows, the size of the solution grows so fast that it will overwhelm any computer. To see why this is so, consider how the recursive solution to the Fibonacci problem is generated. First look at the calling tree for calculating the fourth Fibonacci number

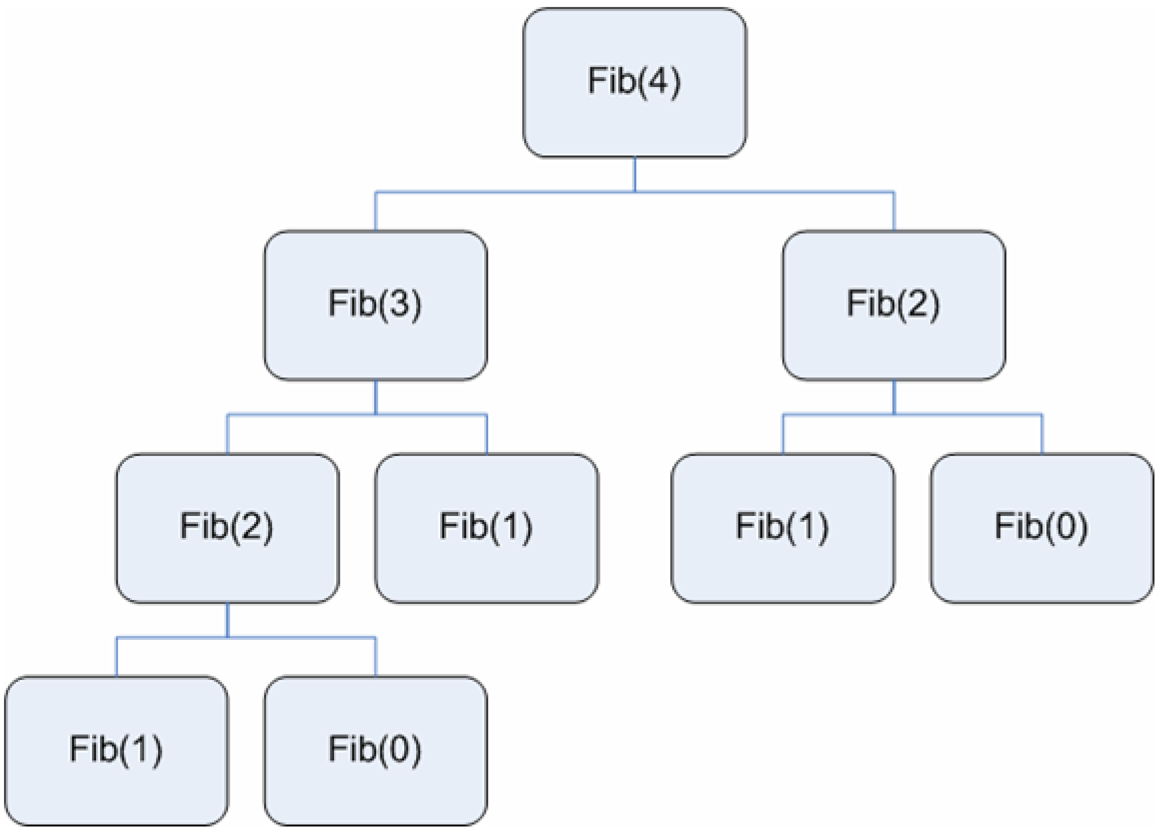

Figure 1.II.3 – Calling tree to calculate Fibonacci number 4

Contrast the Fibonacci tree for 4 above with the one for 5:

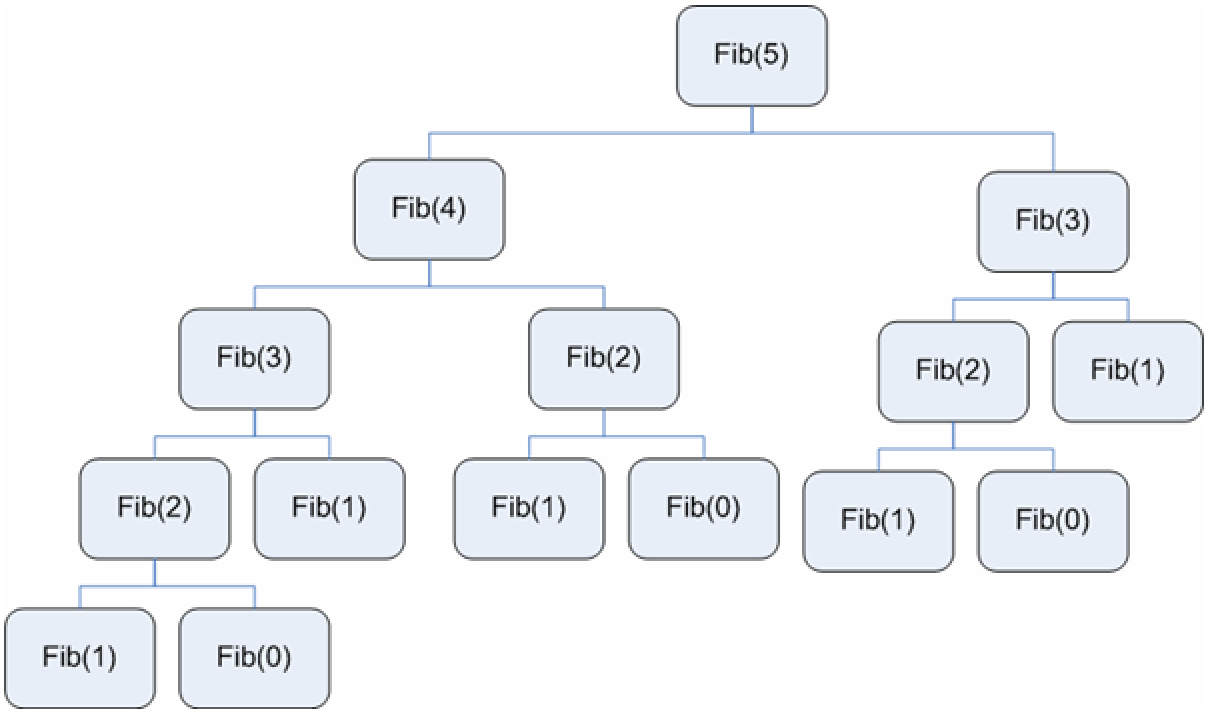

Figure 1.II.4 – Calling tree to calculate Fibonacci number 5

The tree to calculate the fifth Fibonacci number is almost twice as large as the one for the fourth. It is actually about 5/3 bigger, and the growth rate of this algorithm is thus $O\left(\left(\frac{5}{3}\right)^n\right)$. This means that a computer that is ten times as fast as the current one could calculate about three more Fibonacci numbers in the same time using this algorithm (it could calculate Fibonacci numbers to 51 instead of only to 48). Solutions such as this Fibonacci program that grow at an exponential rate in their requirements of either time or space are generally considered unsolvable on a computer since the solutions quickly grow to a point that they will either require days or completely overwhelm the computational resources of the computer.

In general this is true of all problems that can be mapped to a tree: as the size of the input data set grows, the solution grows in exponential time, and so the problem grows too fast to be effectively solved on a computer.

One other thing to note about the Fibonacci number problem is that in the algorithm shown, the calculation of the Fibonacci number for an input of 2 is calculated three times in the process of calculating the Fibonacci number for an input of 5. Thus, for a Fibonacci number it must be possible to find a simpler solution by calculating this value only once. This ability to calculate each number once was the basis for the linear algorithm to calculate a Fibonacci number given earlier in this module in the public static int calcFibLinear(int number) program. But while the solution to this particular problem can be reduced to a linear solution, the same is not true of all problems that have exponential growth rates.

In fact, many interesting problems exhibit a factorial growth rate that is even higher than simple exponential growth. These problems, such as the Bin Packing and Traveling Salesman problems are encountered frequently in computer science, and there is no known solution in less than $N!$ time for either of these problems.

Many programmers believe that problems that have exponential growth are relatively rare and thus are not a real concern. These problems are much more common than most programmers would believe, though, and if care is not taken, an unwary programmer could easily be faced with a problem of this type. Consider for example a simple Crypto-quote problem. In a Crypto-quote, an encoded string is given in which each letter in the encoded string represents a letter in the message. For example, consider the following encrypted string:

fmep m czwp jmb

In this string, the letter f = h, m = a, etc., and the string can be decrypted to reveal the message:

have a nice day

Assuming that only lowercase letters are translated, a programmer could decide to try to break this code by choosing letters at random and substituting them into the encrypted string. For the first letter, there would be 26 choices; for the second, 25 choices; and so on. Because there are nine letters, the total number of combinations would be $(26 * 25 * 24 * 23 * 22 * 21 * 20 * 19 * 18) \approx 1.1 \times 10^{12}$. It might be possible to decrypt this message using this brute force attempt, but what happens if more letters are added to the encrypted string, or if the uppercase letters are added to the possible letters that can be used in the original, unencoded message? Obviously, this problem would become too large to calculate very quickly. Attempting to solve this problem using brute force is not, in general, realistic.

i. Exponential Algebra

There are some useful formulas that you should be familiar with when working with exponents. These formulas are summarized here:

$p^k * p^m = p^{k + m}$

$p^k \div p^m = p^{k - m}$

$(p^k)^m = p^{k * m}$

$p^n + p^n = 2 * p^n \neq p^{2n}$

$2^n + 2^n = 2^{n+1}$

D. Constant Growth (Tractable Problems)

The rest of the problems in this section will deal with problems that are tractable, those with growth rates that are less than exponential and thus do not grow too fast to be calculated on a computer. Specifically, the growth rates that we will cover are constant, logarithmic, and polynomial (including linear growth rate).

The slowest growth rate is one for which the execution time to solve the problem does not grow at all with respect to the size of the input. In other words, the execution time of the function is constant and is written O(1). Most of the cases in which the execution time of the problem is constant are not very interesting.

For example, the following method, which converts a temperature from Fahrenheit to Celsius, has a growth rate of O(1) since the amount of time the calculation requires does not depend on the size of the input. Note that saying this method has a growth rate of O(1) is not the same as saying that it takes one unit of time, or that it will run in the same amount of time as another method that is O(1). It says simply that the amount of time this method takes is not dependant on the input.

```
    public static double fToC (double temp) {
        return ((temp - 32) * 5 / 9);
    }
```

Most constant time operations are not really that interesting when doing a run-time analysis because the size of the input does not vary. One operation that is interesting is array addressing. In array addressing, the access to an array element is constant regardless of the size of the array. This is true because each member of the array is a fixed sized, so the address needed to access each member of the array can be calculated using the following formula:

```
ElementAddress = ArrayBase + (ElementSize * ElementNumber)
```

This fact will have implications later in many algorithms, in particular hash tables, binary heaps, and disjoint sets.

E. Logarithmic Growth (Tractable Problems)

Logarithmic growth is the mirror image of exponential growth. This is shown by the following relationship:

$\log(A^B) = B \log A$

This means that logarithms grow very slowly. In fact, logarithms grow so slowly that the growth rate can often be ignored for practical purposes in implementing a program. It is often true that O(log N) algorithms are more practical, and run in less time, than O(1) algorithms. The reason for this is that algorithms that have a growth rate of O(1) will often have a significantly larger constant execution times will be required of a logarithmic algorithm in any case that will be encountered during the life of the program. This does not mean that O(1) is larger than O(log N)—it just means that when considering the implementation details of the algorithm, logarithmic growth is often slow enough that it can, for practical purposes, be ignored.

Too understand how slow this growth rate is, consider the following program, which uses a binary search algorithm on a sorted array to find 1,000 random integers in an array. (If you are unfamiliar with or have forgotten the details of the binary search algorithm, please refer to CMSC 230 module 2).

```
import java.util.Random;

public class TimeBinarySearch {

  public static int binarySearch (int[] array, int value) {
     int start = 0;
     int end = array.length - 1;
     int found = -1;

     // check if the value is at the start of the array
     if (array[start] == value){
       found = 0;
       return found;
     }

     // check if the value is at the end of the array
     if (array[end] == value) {
       found = array.length;
       return found;
     }

     // start binary search
     int middle = array.length / 2;
     while (true) {
       // Value is found, so return its position
       if (array[middle] == value) {
         found = middle;
         break;
       }

       // Value is not in array, so return -1
       if (start == end) {
         break;
       }

       // Try next value.
       if (array[middle] > value)
         end = middle - 1;
       else
         start = middle + 1;
       middle = (start + end) / 2;
     }

     return found;
  }

  public static void main(String[] args) {
    final int NUMBER_OF_TRIES = 1000;
    int size = 0;

    // Process input parameter
    if (args.length != 1) {
      System.out.println("You must enter array size to search");
      System.exit(1);
    }
    try {
      size = Integer.parseInt(args[0]);
      if (size < 1000) {
        System.out.println("array size must be > 1000");
        System.exit(1);
      }
    } catch(Exception e) {
      System.out.println("array size must be integer");
      System.exit(1);
    }

    int array[] = new int[size];

    for (int i = 0; i < size; i++) {
      array[i] = i;
    }

    // Calculate the time to find NUMBER_OF_TRIES
    // items in the array using a binary search.
    long startTime = System.currentTimeMillis();
    Random random = new Random(startTime);
    int value = 0;
    for (int i = 0; i < NUMBER_OF_TRIES; i++) {
      value = random.nextInt(size-1);
      System.out.println(binarySearch(array, value));

    }
    long endTime = System.currentTimeMillis();

    System.out.println("time used for " + NUMBER_OF_TRIES +
                       " lookups is " + (endTime - startTime));
  }
}
```

This program was run on arrays with 10,000, 100,000, 1,000,000, and 10,000,000 items in the array. After running each array size five times, the average time required for each data set was computed, and the results are summarized in the following graph:

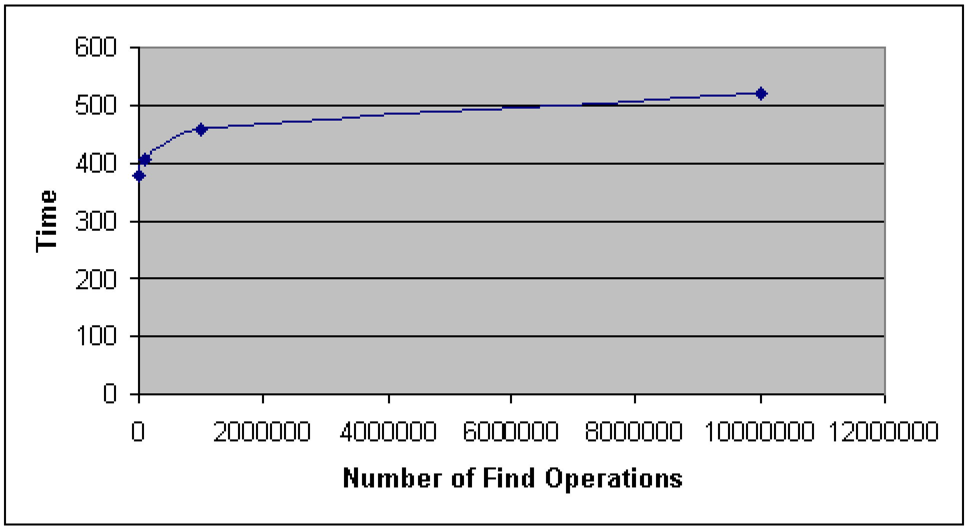

Figure 1.II.5 – Growth rate for a binary search: O(log *N*)

This graph represents the time needed to find the 1,000 items in the array. It shows that although the number data items in the array has increased substantially (by a factor of 1,000), the amount of time needed to find the 1,000 records grows very slowly; it is not much more than a constant as the size of the problem increases. This is not to say that the solution size does not grow with the size of the input, only that it grows very slowly—so slowly, in fact, that often it is safe to ignore it.

i. Logarithmic Algebra

Logarithmic solutions are interesting not only because they normally represent good solutions to a problem, but also because logarithmic relationships have interesting implications on run-time growth. This section gives some logarithm relationships and their implications.

Relation 1:

$\log(A^B) = B \log A$

This equation is important because it shows that taking the log of an exponential function results in a linear function (assuming A is a constant). Since the linear function is tractable, a logarithmic reduction of some intractable problem can result in a tractable solution. Also, as we will often see in this section, it serves to emphasize the fact that logarithms grow very slowly.

Relations 2 and 3:

$\log ab = \log a + \log b$ (where $a, b > 0$)

$\log \frac{a}{b} = \log a - \log b$ (where $a, b > 0$)

These are important relationships to understand, because sometimes they will allow a complex equation to be reduced to a simple summation. For example, consider the definition of a factorial:

$$\prod_{i=1}^{n} i = n * (n - 1) * (n - 2) \ldots$$

If the logarithm is taken of this, the result is:

$$\log\left(\prod_{i=1}^{n} i\right) = \log n + \log (n - 1) + \log (n - 2) \ldots = \sum_{i=1}^{n} \log i$$

This summation is often easier to deal with than the product, and is often used in run-time analysis. Therefore, products are often translated into summations by taking the logarithm in this manner.

Relation 4:

$$\log_B A = \frac{\log_C A}{\log_C B}$$

This relationship shows that the logarithm of any number is a constant in any base. This is useful in general, as it allows an easy way for a programmer to calculate a logarithm in any base when the libraries for a language provide methods for the logarithm in only certain bases (such as base ten). A logarithm in any base can be converted to another base by multiplying by the constant

(which is the denominator of the right hand side of the equation) in the formula

. Note that in this case *B* and *C* must be constants, as they are just the bases of the logarithms.

Furthermore, this relationship has a larger meaning in terms of run-time growth. Students will often ask what the base is for the log function if a program has a growth rate of O(log *N*). The answer is that it does not matter, and in fact it may be considered incorrect to specify a base for the logarithm in the answer. The reason is, once again, that Big-Oh is concerned with run-time growth rates, not absolute execution times. Therefore, any constants that might exist would appear in both the numerator and the denominator when talking about the relative execution time and would cancel one another out. This is the reason for the rule that constants are not considered in Big-Oh notation. Because the Big-Oh of the logs of different bases differ only by a constant, the base of the log is irrelevant to Big-Oh calculations.

Relations 5 and 6:

$\log^k N < O(N)$ for any constant k
$O(\log N^k) = O(k \log N) = O(\log N)$ (because constants are disregarded)

Both of these relationships might seem counterintuitive. In the second case, for example, $O(N^{100})$ is obviously much larger than $O(N)$. Once again, the point here is that the growth of logarithms is extremely slow and overwhelms any type of polynomial growth, and so any constant exponent in a polynomial equation should not be considered for Big-Oh calculations when a logarithm is taken of that polynomial. In fact, this is one of the reasons we will use later to support the statement that any function exhibiting polynomial growth (even $O(N^{100})$) is tractable.

Before continuing, it is important to emphasize again a central point of run-time growth analysis. Just because the run-time growth says the problem is tractable, or that an O(1) solution grows slower than an $O(\log N)$, the run-time growths should not be considered an absolute measure of how good a solution is. Run-time growth analysis should be used as one constraint on the design of a solution.

Not all tractable solutions to a problem are feasible, because an algorithm with an $O(N^2)$ solution that takes five minutes to arrive at an answer is not appropriate for an interactive program. Likewise, not all O(1) solutions are better than $O(\log N)$ solutions. An O(1) solution might be more complex than an $O(\log N)$ solution. It is even possible for an O(1) solution to take longer to execute in all practical applications of the program than an $O(\log N)$ solution. This doesn't mean that Big-Oh constraints are not useful; it simply means that programmers must consider the problem they are trying to solve and apply appropriate design techniques to solve that problem.

F. Polynomial Growth (Tractable Problems)

A polynomial growth rate is one in which the growth rate of the solution is represented by some polynomial based on the size of the input. Polynomial growth rates can be written as $O(N^k)$, where k is a constant. If $k = 1$, the growth rate is called **linear**, as the growth rate of the solution grows linearly with respect to the input. If $k = 2$, the growth rate is called **quadratic** , and if $k = 3$ the growth rate is called **cubic**.

An example of quadratic growth $O(N^2)$ is the simple bubble sort program given below. (Note: If you are unfamiliar with sorting, refer to CMSC 230 module 6.)

```
import java.util.Random;

public class BubbleSort {

  private static void sort(int[] array) {
    boolean swapOccured = false;
    for (int i = 0; i < (array.length-1); i++) {
      swapOccured = false;
      for (int k = 0; k < (array.length-1-i); k++) {
        if (array[k] > array[k+1]){
          swapOccured = true;
          int tmp;
          tmp = array[k];
          array[k] = array[k+1];
          array[k+1] = tmp;
        }
      }
      if (!swapOccured)
        break;
    }
  }

  public static void main(String[] args) {
    int[] array;
    int size = 0;

    // Process input parameter
    if (args.length != 1) {
      System.out.println("You must enter array size to search");
      System.exit(1);
    }
    try {
      size = Integer.parseInt(args[0]);
      if (size < 10) {
        System.out.println("array size must be > 10");
        System.exit(1);
      }

    } catch(Exception e) {
      System.out.println("array size must be integer");
      System.exit(1);
    }

    // Initialize the array with random data.
    Random random = new Random(System.currentTimeMillis());
    array = new int[size];
    for (int i = 0; i < array.length; i++) {
      array[i] = random.nextInt(size);
    }

    // Sort the array using a Bubble Sort and report the
    // time.
    long startTime = System.currentTimeMillis();
    sort(array);
    long endTime = System.currentTimeMillis();
    System.out.println("To sort an array of size " + size +
                       " required " + (endTime-startTime));
  }
}
```

This sort was run on 5,000, 10,000, 15,000, 20,000, and 25,000 items. The results are summarized in the graph below.

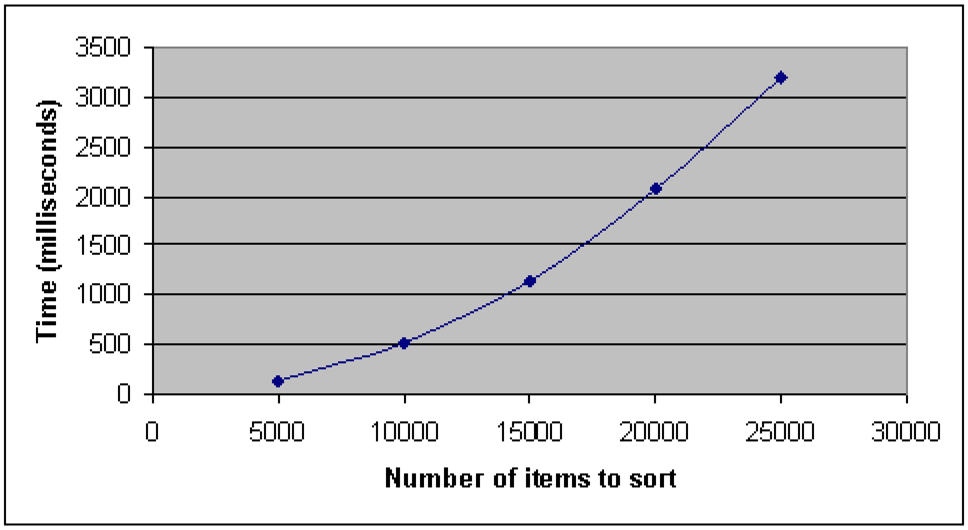

Figure 1.II.6 – Bubble sort run-time growth: O(log n^2)

As this graph shows, the growth rate of a bubble sort is fast, but nowhere near as severe as it would be with exponential growth. This is true of all polynomials. For any values of *k* and *a*, the value of $O(N^k) < O(a^N)$, where *a* and *k* are constants. Thus, it is said that for all *k*, a solution that results in a run-time growth of $O(N^k)$ is tractable. This means that even a solution that is $O(n^{100})$ is considered tractable. Remember that just because the problem is tractable does not mean the implementation is a practical solution. It simply means that the growth rate is such that the problem does not grow so fast that a computer might not be able to solve the problem.

There is another important consideration when analyzing the run-time growth of a program. The run-time growth tells you only how the problem will behave as the size of the input grows; it does not tell you anything about the amount of time an algorithm will take, and as such, it is not always a good way to judge how efficient an algorithm is. For example, consider the following program, which implements an insertion sort, employing another $O(N^2)$ algorithm:

```
import java.util.Random;

public class InsertionSort {

  private static void sort(int[] array) {
    for (int i = 0; i < array.length; i++) {
      for (int k = i; k > 0; k--){
        if (array[k]< array[k-1]) {
          int tmp;
          tmp = array[k];
          array[k] = array[k-1];
          array[k-1] = tmp;
        }
        else break;
      }

    }
  }
  public static void main(String[] args) {
    int[] array;
    int size = 0;

    // Process input parameter
    if (args.length != 1) {
      System.out.println("You must enter array size to search");
      System.exit(1);
    }
    try {
      size = Integer.parseInt(args[0]);
      if (size < 10) {
        System.out.println("array size must be > 10");
        System.exit(1);
      }

    } catch(Exception e) {
      System.out.println("array size must be integer");
      System.exit(1);
    }

    // Initialize the array with random data.
    Random random = new Random(System.currentTimeMillis());
    array = new int[size];
    for (int i = 0; i < array.length; i++) {
      array[i] = random.nextInt(size);
    }

    // Sort the array using a Insertion Sort and report the
    // time.
    long startTime = System.currentTimeMillis();
    sort(array);
    long endTime = System.currentTimeMillis();
    System.out.println("To sort an array of size " + size +
                       " required " + (endTime-startTime));
  }
}
```

The results of running the insertion sort are summarized in the graph below. For comparison purposes, the results from the earlier bubble sort are also shown:

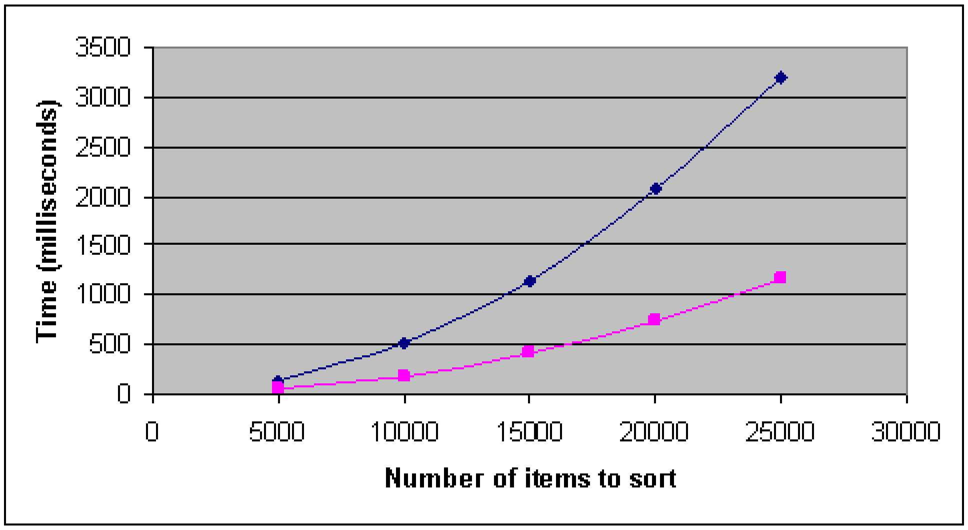

Figure 1.II.7 – Comparison of bubble sort and insertion sort

This graph shows that while bubble sort and insertion sort are both O(N^2), the insertion sort is always faster. The reason is that the latter uses fewer swap operations, which are relatively expensive. This example serves to illustrate the limits of Big-Oh analysis and supports the assertion that it should be used only to evaluate algorithms in terms of their run-time growth, not their overall effectiveness to solve a particular problem.

i. Polynomial Algebra

There are only two relationships for polynomial algebra that interest us for this course. They are:

$n^k * n^m = n^{(k + m)}$
$(n^k)^m = n^{(k*m)}$

These relationships show that any combinations of polynomials will result in another polynomial. This fact is important because it means that if a program is made up of parts that all have polynomial run times, the overall program will have a polynomial run time.

III. Summations and Proofs

It is impossible to understand algorithm analysis without understanding some of the math involved. Therefore, you need to understand how to solve summations and do proofs. This is a point that many students find extremely frightening and challenging. These topics are necessary, however, as will be shown later in this module.

This material, as with all the mathematics in this course, will be presented in an intuitive manner designed for relatively easy understanding . But it is still difficult, so study it carefully and review this material from your algebra, calculus, and discrete math books as necessary.

A. Summations

To solve the growth rate of a program, it is necessary to come up with a way to deal with structures such as loops. This is done easily using summations. A summation is much like a for loop. For example, consider the following simple summation:

$$\sum_{k=1}^{n} C = C + C + C + \ldots = C * n$$

In this summation, the variable *k* is varied from 1 to *n*, with each term inside the summation added to the total for each value of *k*. Inside this summation is the term *C*. *C* is not related to the variable *k* in any way, so for the purposes of this summation it is a constant. Because *C* is a constant, it is added to itself for the number of times that the summation is run. The summation runs from 1...*n*, or (*n* – 1 + 1) = *n* times, meaning *C* is added to itself *n* times, giving the final answer of *Cn*.

This result of summing a constant occurs often in analyzing the Big-Oh of a loop. For example, the following equation appears different from the previous one because of the use of the term *n* in the summation. However, remembering that *n* is simply a limit of the summation and not dependant on *k* shows how easy it is to solve this equation.

$$\sum_{k=1}^{n} n = n + n + n + \ldots = n * n$$

This becomes slightly more difficult when the term inside the summation is dependent on the variable that is being varied in the summation. For example, consider the following summation:

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

This is an extremely common summation in algorithm analysis. Fortunately, the solution for it is not too bad and is given in the following step diagram.

The solutions to summations in this class will almost always follow this method of solving a summation. The summation will be manipulated through addition or subtraction until you get a result that either you recognize as a standard solution or that can be solved relatively easily.

B. Proof by Induction

Proof by induction is done in two steps. First, a **base case** is shown to be true. Next, the theorem is assumed to be true for all cases up to and including some arbitrary value *n*, called the **inductive hypothesis**. The theorem is then applied to *n* to derive the *n* + 1 term, and it is shown that the theorem is true for *n* + 1. We have thus shown that the theorem is true for some base term, and we have now shown that it is true for each term from the base term to *n*, and for *n* and *n* + 1. Since *n* and *n* + 1 represent all cases (including the case where *n* is the base case), a chain is formed from the base case to every case where *n* is greater than the base case, and the theorem must be true.

Proof by induction is often done in computer science when the size of the problem to be proved is not known. It is therefore impossible to enumerate all possible solutions to the problem to show that it is correct. Proof by induction allows us to show that the behavior of the program will obey some constraints, no matter how large the solution becomes.

For example, consider the following program to calculate a summation for an array:

```
public class Summation {
    public static void main(String[] arg) {
        final int n = 10;
        int total = 0;
        for (int k = 1; k <= n; k++){
            total = total + k;
        }
        System.out.println("Total = " + total);
    }
}
```

It is known that the result of this summation is . We want to show that this is indeed the correct result for this program.

An inductive proof shows that the program is correct up to some point. For example, in this program, we would show that the result of the loop would be correct for an array of size 1. We then assume that the result would be correct up to another arbitrary point, such as an array of size *n*. Finally, we show that the program produces the correct result if one more step is taken, for example to an array of size *n* + 1.

This is somewhat easier to see when written mathematically as $\sum_{k=1}^{n} k$. The following step diagram walks through the steps of the inductive proof.

Inductive proofs are very important in areas such as recursive and concurrent programming, robotics, systems that must be reliable (such as the Mars Explorer) compilers, and theory of computation. When dealing with these systems, it is often very important to show that a program or algorithm is correct, and there really is no other way to do so except through use of an inductive proof. However, these proofs are beyond the scope of this module.

Inductive proofs are also useful when discussing run-time growth. For example, it is often necessary to know the maximum number of nodes in a binary tree (if you are unfamiliar with tree structures, refer to CMSC 230 module 5). In the following diagram, it is obvious that the maximum number of nodes in a tree with a depth *d* would occur when all nodes at the level *d* are present, called a complete binary tree. It is also apparent that the number of nodes at any level is exactly twice the number of nodes at the previous level, as every internal node in a tree has exactly two children, so the number of nodes at any level in the tree is 2^d.

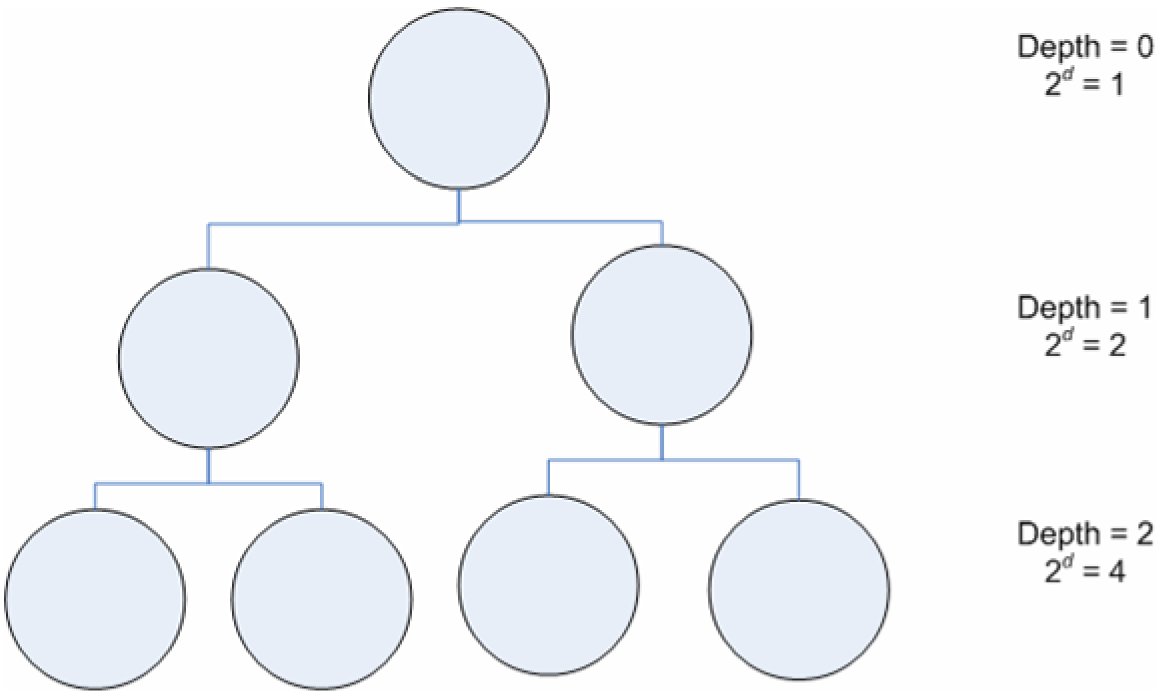

Figure 1.III.1 – Tree of depth $d = 2$

From this diagram, we know the maximum number of nodes in a tree with a depth d will be:

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

It remains now to show that the result of the summation is indeed $2^{(d+1)} - 1$. The following step diagram shows the steps involved it this proof.

Finally, induction can be used to show a relationship between two values. For example, it is important to know if the results of exponential growth (like 2^n) are ever greater than the results from factorial growth (n!). The following step diagram provides the proof that n! is greater than $2n$ for any value of $n > 3$.

Proof by induction is an important tool for any computer scientist, and as such it is important that all students be able to use it.

C. Proof by Counterexample

Proof by counterexample is the easiest possible proof. To do this, simply find a case where the statement is not true.

D. Proof by Contradiction

Proof by contradiction is done by first assuming that a theorem is false, then showing that this assumption contradicts some property that is know to be true, proving that the original assumption is false. A classic example is the question of whether there is a largest prime number. To show that there is no largest prime number, first the assumption is made that there is a largest prime number. If that is true, then all the prime numbers up to that number must be known. Therefore, there must exist a number N that is defined as follows:

$N = (P_1 * P_2 * P_3 * \ldots * P_n) + 1$

This number N is simply the product of all the prime numbers up to the largest prime number, plus 1. However, N is not divisible by any prime $P_1 \ldots P_n$, which means that N is also prime, and so a prime number N exists that is larger than P_n, so P_n cannot exist.

This type of proof is very useful in showing that a solution to a problem is an optimal solution. We will use a proof by contradiction to show the validity of algorithms such as Dijkstra's Shortest Path in module 5.

IV. Algorithm Analysis

In this section, we will look at how to analyze algorithms using Big-Oh notation. In the first part of this section, a formal definition of Big-Oh is presented. Big-Omega, Big-Theta, and Little-oh will also be discussed briefly in this section. Note that in section 1.V.A, the definition of Big-Oh differs significantly from the rest of the course modules, which is more closely associated with Big-Theta. Both definitions of Big-Oh are used in a number of texts, and although the definition in this section is technically more correct, the definition presented in section 1.II.A is more in keeping with how the term Big-Oh is used in industry and other less formal settings. In an effort to make a distinction, *Big-Oh* will be written as *Big-O* in section 1.V.A. The definition you will find in section 1.V.A is provided to underscore the fact that there is a difference between this formal definition and the more common usage of the term.

It is important to remember that the definition used in Section 1.V.A for Big-O will apply only to that section. Everywhere else in the course modules, including in the material from 1.V.B to the end of module 1, the definition given in 1.II.A will be used.

After covering the definitions in 1.V.A, the rest of the section shows the rule for applying Big-Oh analysis to programs. Addition and multiplication of Big-Oh equations will be covered, followed by a discussion of how to calculate Big-Oh as it applies to a program with for loops.

A. Definitions and What They Mean

This section will formally define Big-O, Big-Omega, Big-Theta, and Little-oh, and then explain the importance of these definitions in algorithm analysis. Note that Big-O analysis is covered in CMSC 230 module 2; if you have little current background in this concept, you'll want to review that material.

i. Big-O

The definition of Big-O is the following:

$T(N) = O(f(N))$ if there exist positive constants c and n such that $T(N) \leq cf(N)$ for all $N \geq n$.

This definition might seem complex, but it is really not bad once you understand it. It says that given a set of input data N, a program has a real run time that can be represented by $T(N)$. This program has an upper limit on its growth rate of $f(N)$ if you can define two constants c and n such that if you multiply f(N) by c, the function $cf(N)$ will eventually be larger than $T(N)$ for all values of the input set N that are larger than n.

For example, consider our hypothetical program from earlier in this module that had a run time defined as follows:

$T(N) = 50 + 15N + (N^2)$

The question is whether there is at least one set of values of c and n that will cause the function $c * N^2$ to be larger than $T(N)$ for all values of N larger than n. Note that the actual values of c and n are not important, and there can be infinite combinations of these two constants; all that matters is that at least one set of these constants exist. For example, $c = 5$ can be arbitrarily chosen, and as shown in the following graph, $5 * N^2 > T(N)$ for all values of $N > n = 6$. What this shows is that there is an upper bound on the growth of $T(N)$, and it is $f(N) = 5 * N^2$ for all values of $N > 6$, or $T(N) = O(N^2)$.

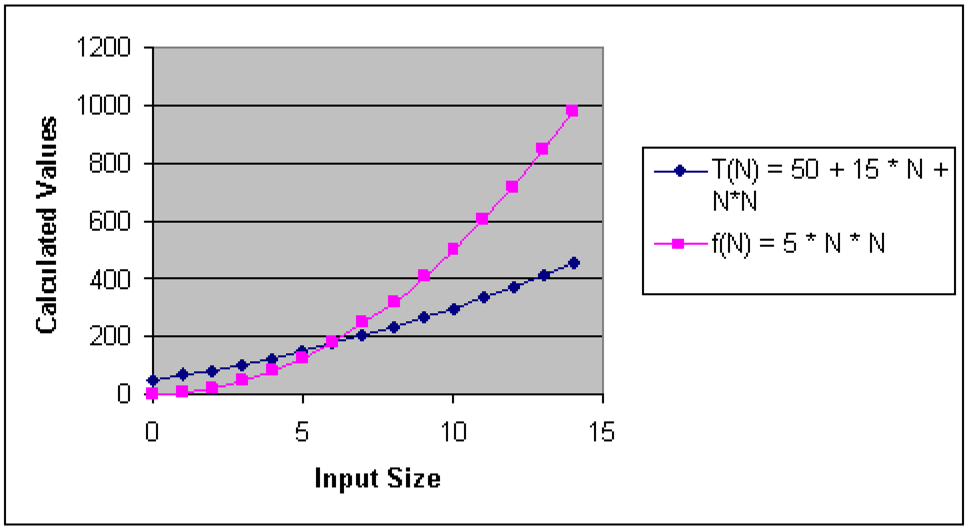

Figure 1.V.1 – Graph of $T(N)$ compared to that of $5N^2$

Note that there are other implications of this formula. Until now, when we talked about Big-Oh it was a **tight bound** on the growth rate of a program, meaning that there is a single $f(N)$—which bounds the run-time growth of the algorithm and consequently how fast the program will grow—that can be estimated given the Big-Oh of the program. This is not the case for the definition of Big-O given here.

The definition of Big-O given here implies that any function $g(N)$ that is an upper bound on $f(N)$ is $O(g(N))$. Simply put, since $f(N) = c_1 * N^2$ can be bound by a function $g(N) = c_2(N^3)$ for all values of $n > n$, any function that is $O(N^2)$ is also $O(N^3)$. So the program above is actually $O(N^2)$, $O(N^3)$, $O(N \log N)$, $O(2^N)$, and all higher-order functions. The definition of Big-O given here is a set of functions that provides an upper bound, not a single function as we have been saying with Big-Oh.

This implies that Big-O is a set relationship, such as $O(N^2) \subseteq O(N^3)$. So the hypothetical example program above can be said to be $T(N) = O(N^2)$ or $T(N) = O(N^3)$. In fact, $T(N)$ is bounded by any $O(g(N))$, where $O(N^2) \subseteq O(g(N))$.

In CMSC 230, these growth rates form a hierarchy:

$O(1) \subseteq O(\log N) \subseteq O(n^k(k<1)) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^k(k>1)) \subseteq O(2^n)$

ii. Big-Omega

The definition of Ω, or Big-Omega, is as follows:

$T(N) = \Omega(g(N))$ if there exist positive constants c and n such that $T(N) \geq cg(N)$ for all $N > n$.

This definition of Big-Omega is similar to that of Big-O, except that it establishes a lower bound on the solution, and thus the growth rate of $T(N) \geq g(N)$. If the definition of Big-O is understood as the upper bound, this definition is similar but represents instead the lower bound. The more difficult question is: what is the purpose of Big-Omega? The purpose of Big-O is more obvious, as it defines the upper limit on the growth of a program and thus can make some guarantees about a program's behavior as the size of the input grows. But Big-Omega's use is less obvious.

The usefulness of Big-Omega will be discussed in the next section, where Big-Theta and Little-oh are defined.

iii. Big-Theta and Little-oh

The definition of Θ, or Big-Theta, is as follows:

$T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$

Big-Theta is the intersection of the sets of functions that make up Big-Oh and Big-Omega. Big-Theta says that if the run-time growth of a program has an upper and a lower bound of $h(N)$, then the overall bound of the program must be $h(N)$; that is, $h(N)$ is an asymptotically tight bound on the program. This is very similar to the definition of Big-Oh that we had been using prior to this section.

Big-Theta is closely related to Little-oh, which is defined below:

$T(N) = o(p(N))$ if for all constants c there exists an n such that $T(N) < cp(N)$ when $N > n$.

This definition for Little-oh implies that if $T(N) = o(p(N))$ then $o(p(N))$ is not an asymptotically tight bound. For example, if $T(N) = 5 + 3N$, it is $O(N^2)$, in accordance with the definition of Big-O given in this section. However it is not $\Omega(N^2)$, because for all constants c there exists an n such that $T(N) < c * N^2$. Thus, $T(N) = 5 + 3N$ is $O(N)$, $\Omega(N)$, and $\Theta(N)$. $T(N)$ is $O(N^2)$, but not $\Omega(N^2)$, so $T(N)$ is $o(N^2)$.

The important definitions in this section were those of Big-O and Big-Theta, and the most important point to remember is that the definition of Big-Oh that we will be using in this course is closer to the definition of Big-Theta as it was defined in this section.

B. Operations on Big-Oh

The following two rules are presented, without derivation, for the addition and multiplication of Big-Oh.

i. Multiplication

For two run-time growth functions $T_1(N) = O\ (f_1(N))$ and $T_2(N) = O(f_2(N))$, $T_1(N) * T_2(N) = O(f_1(N) * f_2(N))$.

This relationship implies that if a program is composed of two parts that are dependent on each other, then the overall run-time growth of the program is the product of the two Big-Oh values for the two parts.

An example of this would be the following code fragment:

```
for (int k = 0; k < n; k++)
    for (int m = 0; m < n; m++);
         sum++
```

In this code fragment, the inner loop is executed once each time the outer loop is executed. Since the inner loop is $O(N)$, and the outer loop is $O(N)$, the overall program run time is $O(N^2)$.

Note that just because a loop is contained inside another loop does not mean that the algorithm is $O(N^2)$, as will be shown later in this section. This will become important later, particularly when calculating the Big-Oh of graph algorithms (such as Dijkstra's Shortest Path and Prim's Minimum Cost Spanning Tree) in module 5.

ii. Addition

For two run-time growth functions $T_1(N) = O(f_1(N))$ and $T_2(N) = O(f_2(N))$, $T_1(N) + T_2(N) = O(f_1(N) + O(f_2(N)) = \max\ (O(f_1(N)), O(f_2(N)))$.

This relationship implies that if a program is composed of two parts that are independent of each other, then the overall run-time growth of the program is the greater of the Big-Oh values for the two individual parts.

An example of this would be the following code fragment:

```
for (int k = 0; k < n; k++)
    for (int m = 0; m < n; m++);
         sum++

for (int k = 0; k < n; k++)
    sum++
```

In this code fragment the first loop is O(N^2), and the second loop is O(N). Because they are independent of each other, the overall Big-Oh is O (N^2) + O(N) = O(N^2).

C. Analyzing Algorithms and Programs

Run-time growth analysis is often done informally simply by figuring out the Big-Oh of loops and recursive structures in a program and using the multiplication and addition formulas to come up with an overall run-time growth rate for the algorithm or program. However, unless the programmer doing the analysis knows how to formally calculate the run-time growth of the program, he or she often will misunderstand the implications of the loops and other structures used in the algorithm or program.

Analyzing a program is basically counting the number of times a statement is executed. To do this, a mathematical formula must be created that can be solved to give a worst-case run time. To set up this formula, you must begin at the innermost blocks of code—those inside loops. The statements inside the loops should all be constant, or the evaluation needs to probe deeper until all the statements are constant time. This doesn't mean that all of the statements take the same amount of time, but that each statement takes a consistent amount of time every time the loop is executed; in other words, the statement does not change the amount of time it takes as the size of the input varies. Once these constant time statements have been characterized, the loop can be evaluated.

There are two caveats in evaluating statements in loops. First, if the code contains an *if* statement, the branches are O(1), but what is inside the branches might not be O(1). Because a branch might or might not be taken, the impact of an *if* statement on the algorithm is not always obvious. The general rule, then, with *if* statements is as follows: If the impact of the *if* statement on the algorithm can be determined, as in the following example, use the result.

```
    for (int k = 0; k < n; k++) {
        if (k == 5)
            for (int l = 0; l < n; l++)
                // Some O(1) operation
    }
```

In this example, it is obvious that the loop inside the *if* block will be executed only once, so even though this is a loop within a loop, the result is O(2*N) = O(N), not O(N^2). In many cases, however, it is not possible to determine when the statements inside the *if* block will be executed, so the analysis should be assume that the *if* branch is always taken, and the algorithm analysis should proceed from that assumption.

The rest of this section of this module will offer examples designed to help you understand the complexities involved in analyzing algorithms and programs.

To understand run-time growth, the first step here is to present a formal example of analyzing a program. The program to be analyzed in the following step diagram is the bubble sort program from section II.

D. Analyzing Loops

The result of the analysis of bubble sort is what was expected for a loop embedded inside of a loop. But that isn't always the case. For example, consider the following code fragments:

```
for (int k = 0; k < n; k++)
  for (int m = 0; m < 5; m++)
    sum++
```

This is an example of a loop inside of a loop. However, the inner loop is O(1), since it always runs five times regardless of the value of *n*. Therefore the Big-Oh of this code fragment is O(*n*).

```
for (int k = 1; k < n; k = k *2)
  sum++;
```

In the for loop in the above example, the value of *k* grows exponentially, so the Big-Oh of the loop is O(log *N*).

```
for (int k = 0; k < n; k++)
  for (int m = 1; m < n; m = m * 2)
    sum++
```

In the preceding code fragment, the outer loop grows at a rate of O(*N*), and the inner loop grows at a rate of (O(log *N*), so the growth rate is O(*N*) * O(log *N*), which is equal to O(*N* log *N*).

```
for (int k = 1; k < n; k = k *2)
  if (k == 1)
  for (int m = 1; m < n; m++)
    sum++
```

This last problem requires a little thought. The outer loop is O(log *N*), but the inner loop is O(*N*), so at first it appears to be the same as the previous problem, O(*N* log *N*). But in this algorithm, the if test says that the inner loop, which is O(*N*), is executed only when *k* = 1, which happens once. So this problem is O(log *N*) + O(*N*), simplified to O(*N*).