Name:_____ Period: _____

Algorithm Performance Analysis

As a beginning programmer, the concept of program efficiency seems irrelevant and unnecessary. Even as a novice programmer, the time spent examining algorithms and source code is often boring and tedious. However, advanced programmers, as their source code routinely exceeds the thousands of lines mark, begin to experience resource drains (loss of memory, slow interfaces, slow decision making, etc.) and become increasingly more interested and concerned with *algorithm efficiency*.

DEFINITION: Efficiency is the measure of an algorithm's (or program's) impact on a computer's resources. An algorithm or program is efficient if it maximizes the scope of accomplishing the task and minimizes the usage of the computer's resources (time and memory). An algorithm becomes more efficient as the efficiency rating drops: the lower the rating, the more efficient an algorithm.

Algorithm Design

The scope of a programming task includes addressing all possible events of that task, including all explicit (obvious) and boundary (special case) conditions. When designing an algorithm, it is not acceptable to address a narrow representation of the data. An efficient algorithm must properly process input data for all cases, including error messages and error handling for inappropriate data.

Computer Resources

Computers have two basic resources – *Space* and *Time*. This means that the efficiency of a program can be measured by examining the impact the program has in one of two ways:

1. How much computer memory does the program require?
2. How long will the program take to execute?

Running Time

As computer systems advance and the memory available to programmers grows, *space* is not a great concern to us right now. *Time*, however, is a ***very*** relevant consideration. By using a computer's built-in clock a programmer can determine to the nearest milli- or microsecond how long it takes for a program to execute. However, actual ***running time*** is a poor means for comparison because it is not universal. It will vary considerably depending on factors such as:

- how well the algorithm was implemented in a language;
- the speed of the computer running the program;
- the compiler used to translate the program into executable form;
- the computer's operating system;
- other computer system loads.

As a result, actual running time measurements usually say more about the computer than about the algorithm. How, then, does a programmer compare algorithms?

Proportional Measures

An algorithm's efficiency can be determined by approximating the *number of steps* required to complete the task.

> **DEFINITION:** Proportional Measures. The running time of an algorithm is *proportional* to the **number of steps** it takes to complete the algorithm. The number of steps taken to carry out the set of instructions is ***N***.

This is a pretty reasonable notion. This means that, regardless of particular implementation, compilers, or computers, an algorithm that takes N steps (an efficiency rating of N) to complete its task will have a running time that is two times that of an algorithm that only requires ½ N (an efficiency rating of ½ N).

➔ ½ N is **two times** faster than N.

However, it would not be practical to determine, or manageable to calculate, the exact number of steps an algorithm would require. It would be sufficient to look at the characteristics of an algorithm and approximate what the proportional running time will be.

This approximation will usually be related to

1. the amount of input the program expects, or
2. the amount of output it will produce, or
3. the 'size' of a value on which the algorithm's computations will be based.

Let's look at an example:

Linear Running Time

```
public void addSeries(Scanner scan)
{
   int n = 0, sum = 0;
   out.print("Enter N:");
   n = scan.nextInt();
   for(int counter = 0; counter<n; counter++)
          sum+= counter;
   out.println("The grand total is " + sum);
}
```

It is easy to count the statements that will be executed in addSeries. There are two print statements, a single scan and initializing assignment, and a loop that will execute N assignments. The total is N + 4. However, as the value of N gets large (I mean ***really*** large), the + 4 becomes insignificant.

For all practical purposes, if N doubles, then the running time of the program doubles. If N triples, then the running time of the program triples. The efficiency of this program is directly related to N, as N grows the efficiency grows linearly with N.

> **DEFINITION:** Linear (efficiency) Running Time. Since the running time of a program based on the summing algorithm will change by about the same amount that N changes, the algorithm is said to be *linear* (it is directly proportional to N). Meaning that as N grows the efficiency rating (running time) grows at the same rate.

Let's look at two other examples.

Constant running time

```
public void gaussSumming(Scanner scan)
{
   out.print("Enter N: ");
   int n = scan.nextInt();
   int sum = (1-n)*(n/2);
   out.println("The grand total is " + sum);
}
```

Method ***gaussSumming*** has 4 statements. Even if the value of N changes, the number of statements will always remain 4. The efficiency will remain a constant four. If we add a heading, an explanation, or a menu, the efficiency will remain a constant (the number of statements). Note that in this program the run time is constant and there is no iteration.

```
public void staticSumming(Scanner scan)
{
   out.print("Enter N: ");
   int n = scan.nextInt();
   int sum = 0;
   for(int x = 6; x<=14; x++)
      sum+= n;
   out.println("The grand total is "+sum);
}
```

Method ***staticSumming*** contains a loop but the upper and lower bounds of the iteration are fixed. The iteration has a fixed number of repetitions that can be calculated. The loop is **not** affected by the variable N, thus the efficiency will remain constant. We can state that the ***gaussSumming*** and ***staticSumming*** algorithms require a constant number of statements, regardless of the value of N.

DEFINITION: Constant (efficiency) Running Time. An algorithm like `gaussSumming` or `StaticSumming` is said to have a *constant* (efficiency) running time, since it does not vary with the value of N. In another words, the efficiency is *constant* with the number of statements.

One more example.

N^2 (or quadratic) running time

```
public void sumSubSeries(Scanner scan)
{
   out.print("Enter N: ");
   int n = scan.nextInt();
   int total = 0;
   for(int i=0; i<n; i++){
      int sum = 0;
      for(int j=0; j<i; j++)
         sum+= j;
      total+=sum;
   }
  out.println("The sum of the subtotals is " + total);
```

This algorithm is a little harder to analyze. A quick inspection reveals the algorithm's main feature requires a loop within a loop (nested loops). The outer loop will iterate N times. However, the loop includes more than one statement – there are two assignments, as well as an inner loop. The "COST" of the outer loop, then, will be:

➔ (2*N)* the number of statements in the inner loop

Let's look at the inner loop. The first time it is entered only one statement will be executed. The second entry will execute two statements, then three, and so on. On the last pass through the inner loop N statements will be

executed. **ON AVERAGE** (because we are looking at the overall efficiency, not the single case specific efficiency), about N/2 statements will be executed in each iteration. The final efficiency of the algorithm can be calculated:

- ➔ (2*N) * (N/2)
- ➔ N^2 (--Simplified--)

> **DEFINITION:** Quadratic (efficiency) Running Time. An algorithm like `sumSubSeries` is said to have a *quadratic* (efficiency) running time, since the nested loop causes the efficiency to grow proportional (N*N) as N grows larger. Meaning, that as N grows, the efficiency rating (running time) grows at a **very fast exponential** rate.

Asymptotic Notation (aka "Big O Notation" or "Order of Magnitude")

The word *asymptotic* means the study of functions of a parameter N, as N becomes larger and larger without bounds. The goal is to develop a notation that will accurately reflect the way in which the running time increases with the size of N, but that will ignore the superfluous details which have little effect on the total efficiency. This is similar to the concept of a limit in mathematics. To accomplish this, the focus must be placed on the one or two basic operations within the algorithm, without too much concern for all the housekeeping operations.

> **DEFINITION:** Big O, or 'order of', notation is used to express the running time (efficiency) of an algorithm in terms of N. For the purpose of big O notation, constant factors (constant multiples of, or additions to, N), or smaller terms involving N, are ignored. Again, we are interested in really big values of N and therefore small values of N and constants (c) are meaningless.

Thus, the efficiency of an algorithm will be expressed by finding a particular factor of N – such as the primary number of characters that have to be inspected, the size of a file that must be sorted, etc. – that has primary influence over an algorithm's running time (insignificant statements are ignored).

TERMINOLOGY: Saying that an algorithm is O(N) means that it will take 'on the order of' N steps to complete the algorithm. Similarly, an $O(N^2)$ algorithm will require about N^2 statements. We say 'about' and 'on the order of' because the number of steps may be multiplied by a constant, or have a factor added. Big O is an analysis of an algorithm for **really large values of N**, without worrying about the details of every statement.

Big O values you must know:

O(1) **Constant time**. The running time (efficiency) of the algorithm is not effected by the data, as in the `gaussSumming` algorithm. Example: Any algorithm that works by evaluating a formula.

$O(\log_2 N)$ **Logarithmic time**. Running time (efficiency) increases less than proportionally and very slowly with N, since Log2 N only doubles when N is squared. For instance, log2 1000 is 10 while log2 1,000,000 is just 20 (increases very slowly). Example: Binary search – the 'split the remainder' algorithm you follow when you look up a number in the telephone book N names long.

O(N) **Linear time**. Running time (efficiency) is directly dependent on N; if N doubles, then the running time doubles as well. Example: Searching through an unordered list of length N, starting at one end.

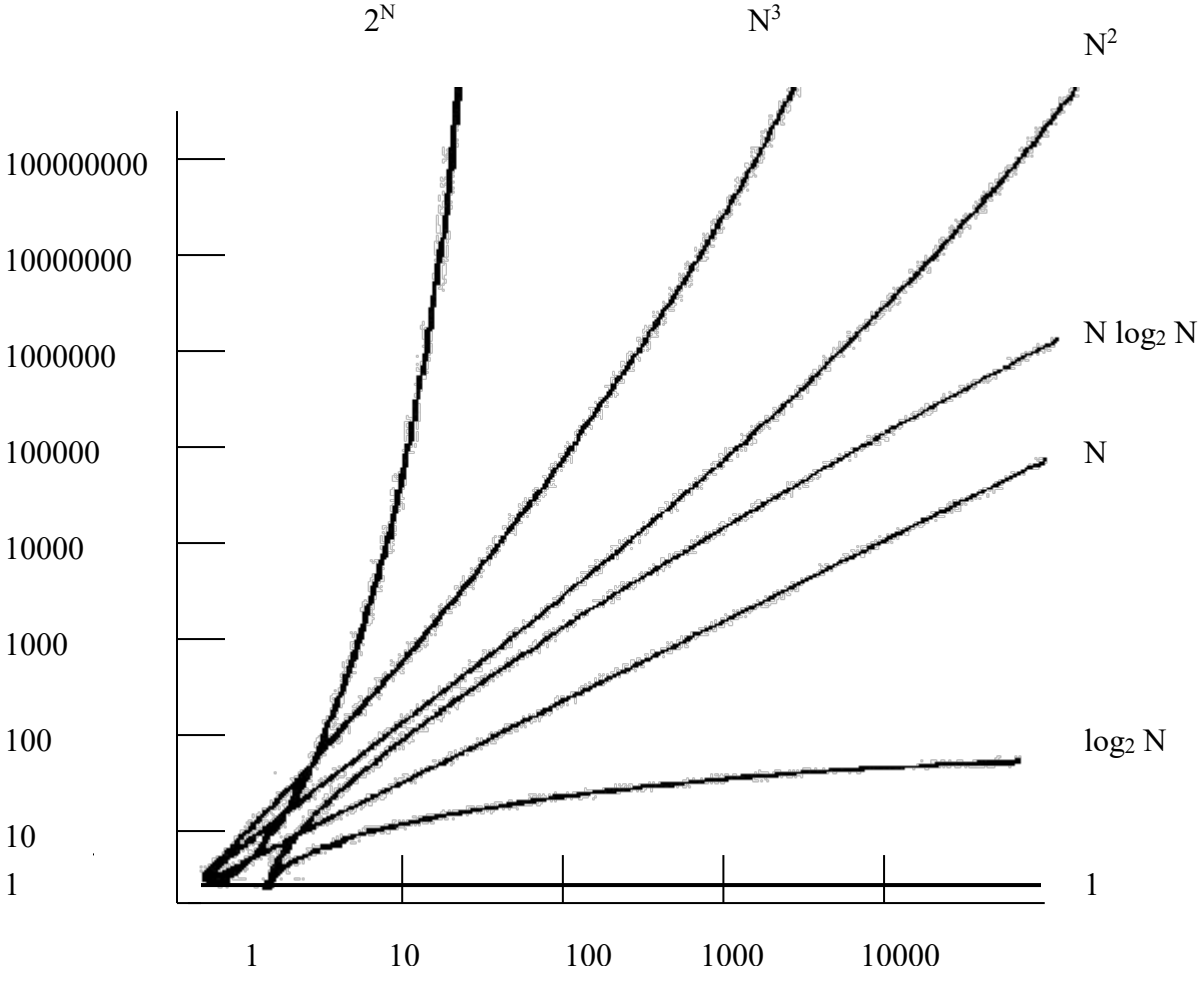
$O(N \log_2 N)$ *No special name*. Usually implies that a linear algorithm invokes a log2 N algorithm. Running time (efficiency) does not increase any faster than N alone, but much faster than proportionally to N. Example: Storing N numbers in a phone book in the first place.

$O(N^2)$ **Quadratic time**. Running time increases with the square of N; when N triples, its running time (efficiency) goes up nine times. Example: Sorting a list of N elements by multiple passes, pulling out the largest one each time.

$O(N^3)$ **Cubic time**. Running time (efficiency) increases the fastest of all big O values. When N is 1,000, N3 is one billion. The time increases by eight whenever N doubles. Example: Matrix multiplication.

$O(2^N)$ **Exponential time**. If N is 10, running time (efficiency) is about 1,000; but doubling N (to 20) increases its running time to 1,000,000! Algorithms with exponential running times are considered impractical. Example: Most brute-force code-breaking methods require exponential time algorithms.

Note: When evaluating the efficiency of an algorithm, Log2 N is often referred to simply as log N or ln N. Quite often the big O notation depends on the instructor. The importance of the big O value is the relative size of N or log N, its base is insignificant. The general practice of using base two follows its close connection to the binary language used by programmers.



Growth Rates for common Big O Functions

Big O comparison

Our interest in efficiency mainly involves a concern about solving a problem of large size. If the array to be sorted contains only 10 elements, whether or not the sort is efficient doesn't matter, since the problem is small and the number of comparisons is reasonably small. But as the size of the array grows, the number of comparisons grows even faster.

A constant computing time is referred to as O(1). There is nothing better than a big O efficiency of O(1). Since it is our goal to be efficient and minimize work, O(1) is better than O(N) (linear time), which in turn is better than $O(N^2)$ (quadratic time), only $O(N^3)$ (cubic time) is worse, and $O(2^N)$ (exponential) is awful. One other frequently seen computing time is $O(\log_2 N)$, which is better than O(N). It is interesting to note that $O(2^N)$ grows so quickly and is so bad that the computation time required for problems of this order may exceed the estimated life span of the universe!

Imprecision of Big O, yet acceptable.

It is important to realize that the constant c in the definition of big O affects the efficiency of the algorithms for small values of N; however, because we are interested in large values of N the loss in precision is acceptable. Thus, calculating big O begins by identifying the constant and removing it and any insignificant values from the function.

Example: $17n - 5$, where c = 17
, where - 5 is insignificant
is O(n)

Example: $35n^2 + 100$, where c = 35
, where + 100 is insignificant
is $O(n^2)$

Big O comparison of searches.

The basic method of a sequential search is exactly the same for both a sorted and an unsorted array. In fact, the target search item is compared to exactly the same key elements in the items of the list in either version. Hence if we wish to determine the efficiency of the sequential search algorithm, then knowing the number of comparisons of keys is the most accurate.

Short as sequential search is, when you start to count executable statements (specifically, the comparison of keys), it becomes obvious that the number of comparisons is dependent on how many times the loop will be iterated. If the search is successful on the first comparison, it is very efficient, yet if the search is unsuccessful (the key is never found), the maximum number of comparisons is made (the entire set of data), and the search is terribly inefficient. There is no way to know in advance whether or not the search will be successful, and if the search is successful, there is no way of knowing if the target will be found on the first comparison, the last comparison, or somewhere between.

To obtain useful information, you must do several analyses. The three different conditions under which algorithms are most commonly evaluated and compared to are the *best-case* scenario, the *worst-case* scenario, and the *average-case* scenario. If the search is unsuccessful, then the target will have been compared to all items in the list, for a total of N comparisons, where N is the length of the list.

In this *worst-case* scenario, the efficiency is linear, the running time increases proportionally as the length of the list increases, and is thus O(N).

The *best-case* scenario would occur if the search is successful on the very first try, only one item is examined. In this *best-case* scenario, the efficiency is constant and will always make only one search, and is thus O(1).

The *average-case* scenario is rarely as easy to quantify as the *best-case* scenario or the *worst-case* scenario. Although the *average-case* scenario requires a bit more mathematical or sophistication, it will always lie between the *best-case* scenario and the *worst-case* scenario.

In the *average-case* scenario for a successful sequential search, if the target is in position *k*, then it will have been compared with the first *k* keys, for a total of exactly *k* comparisons. Under the assumption that each successful search has an equally likely chance of occurring we can find the average number of key comparisons done in a successful sequential search. To find the average number of key comparisons, you add the number for all successful searches, and divide by N, the number of items in the list.

➔ $\frac{1+2+3+ \ldots + n}{n}$

➔ $\frac{½\ n(n+1)}{n}$ (using summation identity ➔ $1+2+3+ \ldots + n = ½\ n\ (n+1)$

➔ ½ (n+1) (simplifying)

➔ ½ n + ½ , where c = ½
, where + ½ is insignificant

➔ therefore: O(N).

Hence, the on average efficiency of a sequential search for a successful search is O(N). Because both the worst-case and average-case of a sequential search are O(N), and are less efficient than the best-case, we conclude that, on a list of length N, a sequential search has '<u>at worst</u>' a running time O(N).

When evaluating a binary search you must consider the restriction that the search requires all items in the list be completely in ascending order. If the list is not presently in order, then the process of putting the list in order must be studied when considering the efficiency. The efficiency of sorts will be considered shortly. For now let's assume that the list is in ascending order and evaluate just the binary search.

A binary search algorithm will use two indices, top and bottom, to enclose the part of the list in which we are looking for the target key. After each iteration, either the search terminates successfully or the size of the part of the list remaining to be searched is reduced by half. More formally, the invariant that must hold true before and after each iteration of the loop algorithm is:

➔ *the target key, provided it is present, will be found between the indices bottom and top, inclusive.*

DEFINITION: <u>Loop Invariant</u> is a statement that is true at the beginning of every iteration of the loop.

Therefore, after *k* comparisons the list remaining to be examined is at most of size [N/2k] (N is the number of items). For example, if N is the number of items in the original list, the list decreases in size in the following manner:

→ N/2, N/4, N/8, N/16, N/32 …

The maximum number of values that need to be examined in a binary search is *logarithmic*. The running time for the key **increases less than proportionately to N.**

➔ $\log_2 N$

This number represents the base-2 logarithm of the number of entries to be searched. For example, a given entry may be found in a file of a million records in approximately 19 attempts, on the average. A linear search, on the other hand, requires a number of attempts that averages half the number of items in the list. Therefore, a linear search of a list of a million items would have to sample approximately 500,000 items before finding the desired key, on the average.

Hence, in the *worst-case*, a binary search requires $O(\log_2 N)$ key comparisons to search a list.

Big O comparison of sorts.

The number of comparisons in a straight **selection** sort and **bubble** sort is $N(N - 1)/2$, or, in expanded form, $0.5N^2 - 0.5N$. In Big O notation, we consider the first term as the **dominant term** (the term which increases the processing time, thus making the sort less efficient). The $- 0.5N$ of the function becomes insignificant and the .5 constant can be ignored. This means that, for values of N, the computation time will be proportional to N^2, or in big O notation, $O(N^2)$.

The analysis of the **insertion** sort must consider two cases; *best-case* and *worst-case*. In the *best-case* the array would already be in order and only one comparison is made on each pass, therefore the best case sort is O(N). In the *worst-case* (the array is sorted in reverse order), the number of comparisons is the same as the straight selection sort and bubble sort, namely, $N(N - 1)/2$, or $O(N^2)$. *On the* average, the insertion sort is $O(N^2)$. It is better than the straight selection sort, however, if the original file approaches sorted order.

Since the straight selection, bubble, and insertion sorts are all on the same order of N2 for large values of N, *there is no difference between their performances*. Although, one sort or another may have advantages (if the sort is close to being in order or close to being out of order), the difference, *on average*, is not significant.

The analysis of **merge sort**, which is recursive, is a little more complex. The array is divided into two pieces, which are then examined. Each of these segments is then divided, making four pieces. At each level, the number of pieces doubles. The question becomes, at what level have we finished dividing the elements? If we split each segment into approximately one half each time, it will take up $\log_2 N$ splits, similar to a binary search. At each split, we make O(N) comparisons. Therefore, merge sort is $O(N \log_2 N)$ on the best case, average case, and worst case, which is quicker than $O(N^2)$.

A major disadvantage of merge sort is that it needs a temporary array that is as large as the original array to be sorted. This could be a problem if space is a factor.

References

Cooper, Doug & Clancy, Michael (1985). Oh! Pascal! (2nd ed.). New York, N.Y.: W. W. Norton & Company, Inc.

Dale, Nell & Lilly, Susan C. (1985). Pascal plus Data Structures. Lexington, Massachusetts: D.C. Heath and Company.

Gilbert, Harry M. & Larky, Arthur I. (1984). Practical Pascal. Cincinnati, Ohio: South-Western Publishing Co.

Horowitz, Ellis & Sahni, Sartaj (1984). Fundamentals of Data Structures in Pascal (4th ed.). Rockville, MD: Computer Science Press, Inc.

Horowitz, Ellis & Sahni, Sartaj (1984). Fundamentals of Data Structures in Pascal. Rockville, MD: Computer Science Press, Inc.

Kruse, Robert L. (1984). Data Structures and Program Design. Englewood Cliffs, New Jersey: Pretence – Hall Inc.

Mandell, Steven L. & Colleen J. (1985). Computer Science with Pascal for Advanced Placement Students. St. Paul, MN: West Publishing Co.

Miller, Philip L. & Lee W. (1987). Programming by Design. Pittsburgh, Pennsylvania: Carnegie Publishing, Inc.