CHAPTER ONE

Analysis Of Algorithms

Justifying The Means

Not every time is the dictum "ends justify the means" correct, more so in Computer Science. Just because we got the right answer (end) does not mean that the method (means) that we employed to obtain it is correct. In fact the efficiency of obtaining the correct answer is largely dependent on the method employed to obtain it. And at times getting a correct solution late is as bad as getting a wrong solution.

The method of solving a problem is known as an algorithm. More precisely, an algorithm is a sequence of instructions that act on some input data to produce some output in a finite number of steps. An algorithm must have the following properties:

(a) Input – An algorithm must receive some input data supplied externally.

(b) Output – An algorithm must produce at least one output as the result.

(c) Finiteness – No matter what is the input, the algorithm must terminate after a finite number of steps. For example, a procedure which goes on performing a series of steps infinitely is not an algorithm.

(d) Definiteness – The steps to be performed in the algorithm must be clear and unambiguous.

(e) Effectiveness – One must be able to perform the steps in the algorithm without applying any intelligence. For example, the step—Select three numbers which form a Pythogorian triplet—is not effective.

All algorithms basically fall under two broad categories—Iterative (repetitive) algorithms and Recursive algorithms. Iterative algorithms typically use loops and conditional statements. As against this, the Recursive algorithms use a 'divide and conquer' strategy. As per this strategy the Recursive algorithm breaks down a large problem into small pieces and then applies the algorithm to each of these small pieces. This often makes the recursive algorithm small, straightforward, and simple to understand.

Why Analyze Algorithms

An algorithm must not only be able to solve the problem at hand, it must be able to do so in as efficient a manner as possible. There

might be different ways (algorithms) in which we can solve a given problem. The characteristics of each algorithm will determine how efficiently each will operate. Determining which algorithm is efficient than the other involves analysis of algorithms.

While analyzing an algorithm *time* required to execute it is determined. This time is not in terms of number of seconds or any such time unit. Instead it represents the number of operations that are carried out while executing the algorithm. Time units are not useful since while analyzing algorithms our main concern is the relative efficiency of the different algorithms. Do also note that an algorithm cannot be termed as better because it takes less time units or worse because it takes more time units to execute. A worse algorithm may take less time units to execute if we move it to a faster computer, or use a more efficient language. Hence while comparing two algorithms it is assumed that all other things like speed of the computer and the language used are same for both the algorithms.

Note that while analyzing the algorithm we would not be interested in the actual number of operations done for some specific size of input data. Instead, we would try to build an equation that relates the number of operations that a particular algorithm does to the size of the input. Once the equations for two algorithms are formed we can then compare the two algorithms by comparing that rate at which their equations grow. This growth rate is critical since there are situations where one algorithm needs fewer operations than the other when the input size is small, but many more when the input size gets large.

While analyzing iterative algorithms we need to determine how many times the loop is executed. To analyze a recursive algorithm one needs to determine amount of work done for three things—breaking down the large problem to smaller pieces, getting solution for each piece and combining the individual solutions to

get the solution to the whole problem. Combining this information and the number of the smaller pieces and their sizes we then need to create a recurrence relation for the algorithm. This recurrence relation can then be converted into a closed form that can be compared with other equations.

What Is Analysis

The analysis of an algorithm provides information that gives us a general idea of how long an algorithm will take for solving a problem. For comparing the performance of two algorithms we have to estimate the time taken to solve a problem using each algorithm for a set of N input values. For example, we might determine the number of comparisons a searching algorithm does to search a value in a list of N values, or we might determine the number of arithmetic operations it performs to add two matrices of size N*N.

As said earlier, a number of algorithms might be able to solve a problem successfully. Analysis of algorithms gives us the scientific reason to determine which algorithm should be chosen to solve the problem. For example, consider the following two algorithms to find the biggest of four values:

Algorithm One:

```
big = a
if ( b > big )
    big = b
endif
if ( c > big )
    big = c
endif
if ( d > big )
    big = d
endif
return big
```


Algorithm Two:

```
if ( a > b )
    if ( a > c )
        if ( a > d )
            return a
        else
            return d
        endif
    else
        if ( c > d )
            return c
        else
            return d
        endif
    endif
else
    if ( b > c )
        if ( b > d )
            return b
        else
            return d
        endif
    else
        if ( c > d )
            return c
        else
            return d
        endif
    endif
endif
```

On careful examination of the two algorithms, you can observe that each does exactly three comparisons to find the biggest number. 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 big. This extra space is not significant if we are comparing numbers or characters, but it may be with other types of data, like say record of an employee. A record may typically contain 15-20 fields and the criterion for determining the larger of the two records might be complex, like say, assigning weights to values in different fields. The purpose of determining the number of comparisons is to then use them to figure out which of the algorithms under consideration can solve the problem more efficiently.

What Analysis Doesn't Do

The analysis of algorithms does not give a formula that helps us determine how many seconds or computer cycles a particular algorithm will take to solve a problem. This is not useful information to choose the right algorithm because it involves lots of variables like:

- Type of computer
- Instruction set used by the Microprocessor
- What optimization compiler performs on the executable code, etc.

No doubt that all these factors have a direct bearing on how fast a program for an algorithm will run. However, if decide to take them into consideration we might end up with a situation where by moving a program to a faster computer, the algorithm would become better because it now completes its job faster. That paints a wrong picture. Hence the analysis of algorithms should be done regardless of the computer on which the program that implements the algorithm is going to get executed.

What To Count And Consider

An algorithm may consist of several operations and it may not be possible to count every one of them as a function of N, the number of inputs. The difference between an algorithm that does N+5 operations and one that does N+250 operations becomes meaningless as N gets very large.

For example, suppose we have an algorithm that counts the number of characters in a file. An algorithm for that might look like the following:

```
Count = 0
While there are more characters in the file do
    Increment Count by 1
    Get the next character
End while
Print Count
```

If there are 500 characters present in the file we need to initialize Count once, check the condition 500 + 1 times (the +1 is for the last check when the file is empty), and increment the counter 500 times. The total number of operations is:

Initializations – 1
Increments – 500
Conditional checks – 500 + 1
Printing – 1

As can be seen from the above numbers the number of increments and conditional checks are far too many as compared to number of initialization and printing operations. The number of initialization and printing operations would remain same for a file of any size and they become a much smaller percentage of the total as the file size increases. For a large file the number of initialization and printing operations would be insignificant as compared to the

number of increments and conditional checks. Thus, while doing analysis of this algorithm the cost of the initialization becomes meaningless as the number of input values increases.

It is very important to decide what to count while analyzing an algorithm. We must first identify which is the significant operation or operations in the algorithm. Once that is decided, we should determine which of these operations are integral to the algorithm and which merely contribute to the overheads. There are two classes of operations that are typically chosen for the significant operation—comparison or arithmetic.

For example, in Searching and Sorting algorithms the important task being done is the comparison of two values. While searching the comparison is done to check if the value is the one we are looking for, whereas in sorting the comparison is done to see whether values being compared are out of order. Comparison operations include equal, not equal, less than, greater than, less than or equal and greater than or equal.

The arithmetic operations fall under two groups—additive and multiplicative. Additive operators include addition, subtraction, increment, and decrement. Multiplicative operators include multiplication, division, and modulus. These two groups are counted separately because multiplication operations take longer time to execute than additions.

Cases To Consider During Analysis

Choosing the input to consider when analyzing an algorithm can have a significant impact on how an algorithm will perform. For example, 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. Hence multiple input sets must be

considered while analyzing an algorithm. These include the following:

(a) Best Case Input – This represents the input set that allows an algorithm to perform most quickly. With this input the algorithm takes shortest time to execute, as it causes the algorithms to do the least amount of work. For example, for a searching algorithm the best case would be if the value we are searching for is found in the first location that the search algorithm checks. As a result, this algorithm would need only one comparison irrespective of the complexity of the algorithm. No matter how large is the input, searching in a best case will result in a constant time of 1. Since the best case for an algorithm would usually be very small and frequently constant value, a best case analysis is often not done.

(b) Worst Case Input – This represents the input set that allows an algorithm to perform most slowly. 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 example, for a searching algorithm, 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.

(c) Average Case Input – This represents the input set that allows an algorithm to deliver an average performance. Doing Average-case analysis is a four-step process. These steps are as under:

(i) Determine the number of different groups into which all possible input sets can be divided.
(ii) Determine the probability that the input will come from each of these groups.

(iii) 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.

Calculate average case time using the formula:

$$A(n) = \sum_{i=1}^{m} p_i * t_i$$

where,

n = Size of input
m = Number of groups
p_i = Probability that the input will be from group i
t_i = Time that the algorithm takes for input from group i.

Rates Of Growth

While doing analysis of algorithms more than the exact number of operations performed by the algorithm, it is the rate of increase in operations as the size of the problem increases that is of more importance. This is often called 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.

Table 1-1 shows rate of growth for some of the common classes of algorithms for a wide range of input sizes. You can observe that there isn't a significant difference in values when the input is small, but once the input value gets large, there are big differences. Hence while doing analysis of algorithm we must consider what happens when the size of the input is large, because small input sets can hide rather dramatic differences.

n	log n	n log n	n^2	n^3	2^n
1	0.0	0.0	1.0	1.0	2.0
2	1.0	2.0	4.0	8.0	4.0
5	2.3	11.6	25.0	125.0	32.0
10	3.3	33.2	100.0	1000.0	1024.0
15	3.9	58.6	225.0	3375.0	32768.0
20	4.3	86.4	400.0	8000.0	1048576.0
30	4.9	147.2	900.0	27000.0	1073741824.0
40	5.3	212.9	1600.0	64000.0	1099511627776.0
50	5.6	282.2	2500.0	125000.0	1125899906842620.0

Table 1-1. *Rate of increase in common algorithm classes.*

The data in Table 1-1 also illustrate that the faster growing functions increase at such a rate that they quickly dominate the slower-growing functions. Thus, if the algorithm's complexity is a combination of a two of these classes, we can safely ignore the slower growing terms. On discarding these terms we are left with what we call the order of the function or related algorithm. Based on their order algorithms can be grouped into three categories:

(a) Algorithms that grow at least as fast as some function

(b) Algorithms that grow at the same rate

(c) Algorithms that grow no faster

The categories (a), (b), (c) mentioned above are commonly known as Big Omega $\Omega(f)$, Big Oh $O(f)$ and Big Theta $\theta(f)$, respectively.

Of these, the Big Omega category of functions is not of much interest to us since for all values of n greater than some threshold

value n_0 all the functions in $\Omega(f)$ have values that are at least as large as f. That is, all functions in this category grow as fast as f or even faster.

While analyzing algorithms we are on the lookout for an algorithm that does better than the one that we are considering. Since big theta category represents a class of functions that grow at the same rate as the function f this category too is not of interest to us.

Big Oh $O(f)$ category represents the class of functions that grow no faster than f. This means that for all values of n greater than some threshold n_0, all the functions in $O(f)$ have values that are no greater than f. Thus the class $O(f)$ has f as an upper bound, so none of the functions in this class grow faster than f. This means that if $g(x) \varepsilon O(f)$, $g(n) < cf(n)$ for all $n > n_0$ (where c is a positive constant).

The Big Oh class of functions would be of interest to us. While 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.

Analysis of Sequential Search Algorithm

The Sequential Search Algorithm is the simplest searching algorithm. We will see that this algorithm is not very efficient but will successfully search in any list.

In any searching algorithm the aim is to look through a list to find a particular element. Although not required, while doing a sequential search the list is considered to be unsorted.

Sequential search looks at elements, one at a time, from the first in the list until a match is found. It returns the index of where the value being searched is found. If the value is not found the

algorithm returns an index value that is outside the range of the list of elements. Assuming that the elements of the list are located in positions 0 to N-1 in the list, we can return a value -1 if the element being searched is not in the list. The complete algorithm for sequential search in pseudo code form is given below. For the sake of simplicity, we have assumed that there is no repetition of values in the list.

```
Sequentialsearch (list, value, N)
List the elements to be searched
value    the value being searched for
N    the number of elements in the list

For i = 1 to N do
    If (value = list[i] )
        Return i
    End if
End for
Return 0
```

Let us now analyze this algorithm.

Worst Case Analysis

There are two worst cases for the sequential search algorithm:

- The value being searched matches the last element in the list
- The value being searched is not present in the list.

For both these cases we need to find out how many comparisons are done. Since we have assumed all the elements in the list are unique in both the cases N comparisons would be made.

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 value being searched will sometimes not be found.

If the value begin searched is present in the list, it can be present at any one of the N places. Since all these possibilities are equally likely, there is a probability of 1/N for each potential location.

The number of comparisons made if the value being searched is found at first, second, third location, etc. would be 1, 2, 3 and so on. This means that the number of comparisons made is the same as the location where the match occurs. This gives the following equation for this average case:

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

$$A(N) = \frac{1}{N} * \frac{N(N+1)}{2}$$

$$A(N) = \frac{N+1}{2}$$

Let us now consider the possibility that the value being searched is not present in the list. Now there are N+1 possibilities. For the case where the value being searched is not in the list there will be N comparisons. If we assume that all N+1 possibilities are equally likely, we get the following equation for this average case:

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

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

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

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

$$A(N) \approx \frac{N+2}{2}$$ (As N gets very large, $\frac{1}{N+1}$ becomes almost 0)

We can observe that by including the possibility of the target not being in the list only increases the average case by ½. When we consider this amount relative to the size of the list, which could be very large, this ½ is not significant.