ANALYSIS OF ALGORITHM

After designing an algorithm, it has to be checked and its correctness needs to be predicted; this is done by analyzing the algorithm. The algorithm can be analyzed by tracing all step-by-step instructions, reading the algorithm for logical correctness, and testing it on some data using mathematical techniques to prove it correct. Another type of analysis is to analyze the simplicity of the algorithm. That is, design the algorithm in a simple way so that it becomes easier to be implemented. However, the simplest and most straightforward way of solving a problem may not be sometimes the best one. Moreover there may be more than one algorithm to solve a problem. The choice of a particular algorithm depends on following performance analysis and measurements :

1. Space complexity
2. Time complexity

SPACE COMPLEXITY

Analysis of space complexity of an algorithm or program is the amount of memory it needs to run to completion.

Some of the reasons for studying space complexity are:

1. If the program is to run on multi user system, it may be required to specify the amount of memory to be allocated to the program.
2. We may be interested to know in advance that whether sufficient memory is available to run the program.
3. There may be several possible solutions with different space requirements.
4. Can be used to estimate the size of the largest problem that a program can solve.

The space needed by a program consists of following components.

- *Instruction space* : Space needed to store the executable version of the program and it is fixed.
- *Data space* : Space needed to store all constants, variable values and has further two components :
  - (*a*) Space needed by constants and simple variables. This space is fixed.
  - (*b*) Space needed by fixed sized structural variables, such as arrays and structures.
  - (*c*) Dynamically allocated space. This space usually varies.
- *Environment stack space:* This space is needed to store the information to resume the suspended (partially completed) functions. Each time a function is invoked the following data is saved on the environment stack :
  - (*a*) Return address : *i.e.*, from where it has to resume after completion of the called function.
  - (*b*) Values of all lead variables and the values of formal parameters in the function being invoked .

The amount of space needed by recursive function is called the recursion stack space. For each recursive function, this space depends on the space needed by the local variables and the formal parameter. In addition, this space depends on the maximum depth of the recursion *i.e.*, maximum number of nested recursive calls.

TIME COMPLEXITY

The time complexity of an algorithm or a program is the amount of time it needs to run to completion. The exact time will depend on the implementation of the algorithm, programming language, optimizing the capabilities of the compiler used, the CPU speed, other hardware characteristics/specifications and so on. To measure the time complexity accurately, we have to count all sorts of operations performed in an algorithm. If we know the time for each one of the primitive operations performed in a given computer, we can easily compute the time taken by an algorithm to complete its execution. This time will vary from machine to machine. By analyzing an algorithm, it is hard to come out with an exact time required. To find out exact time complexity, we need to know the exact instructions executed by the hardware and the time required for the instruction. The time complexity also depends on the amount of data inputted to an algorithm. But we can calculate the order of magnitude for the time required.

That is, our intention is to estimate the execution time of an algorithm irrespective of the computer machine on which it will be used. Here, the more sophisticated method is to identify the key operations and count such operations performed till the program completes its execution. A key operation in our algorithm is an operation that takes maximum time among all possible operations in the algorithm. Such an abstract, theoretical approach is not only useful for discussing and comparing algorithms, but also it is useful to improve solutions to practical problems. The time complexity can now be expressed as function of number of key operations performed. Before we go ahead with our discussions, it is important to understand the rate growth analysis of an algorithm, as shown in Fig. 1.3.

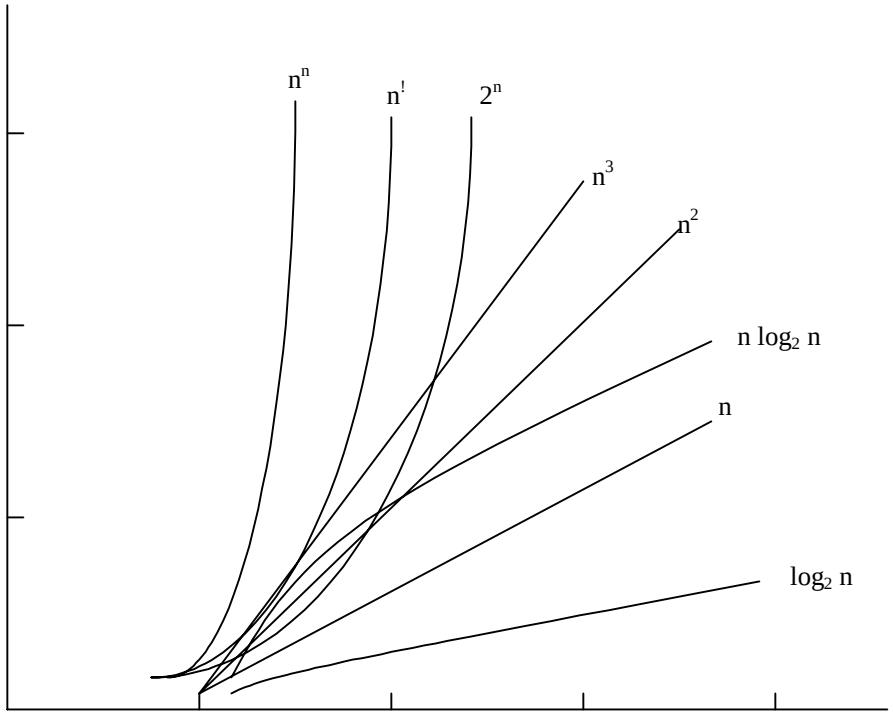

Fig. 1.3

The function that involves '*n*' as an exponent, *i.e.*, 2^n, n^n, n ! are called exponential functions, which is too slow except for small size input function where growth is less than or equal to n^c,(where '*c*' is a constant) *i.e.*; n^3, n^2, $n \log_2 n$, n, $\log_2 n$ are said to be polynomial. Algorithms with polynomial time can solve reasonable sized problems if the constant in the exponent is small.

When we analyze an algorithm it depends on the input data, there are three cases :

1. Best case
2. Average case
3. Worst case

In the best case, the amount of time a program might be expected to take on best possible input data.

In the average case, the amount of time a program might be expected to take on typical (or average) input data.

In the worst case, the amount of time a program would take on the worst possible input configuration.

AMSTRONG COMPLEXITY

In many situations, data structures are subjected to a sequence of instructions rather than one set of instruction. In this sequence, one instruction may perform certain modifications that have an impact on other instructions in the sequence at the run time

itself. For example in a *for* loop there are 100 instructions in an *if* statement. If *if* condition is false then these 100 instructions will not be executed. If we apply the time complexity analysis in worst case, entire sequence is considered to compute the efficiency, which is an excessively large and unrealistic analysis of efficiency. But when we apply amortized complexity, the complexity is calculated when the instructions are executed (*i.e.*, when *if* condition is true)

Here the time required to perform a sequence of (related) operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a simple operation might be expensive. Amortized analysis guarantees the average performance of each operation in the worst case.

TIME-SPACE TRADE OFF

There may be more than one approach (or algorithm) to solve a problem. The best algorithm (or program) to solve a given problem is one that requires less space in memory and takes less time to complete its execution. But in practice, it is not always possible to achieve both of these objectives. One algorithm may require more space but less time to complete its execution while the other algorithm requires less time space but takes more time to complete its execution. Thus, we may have to sacrifice one at the cost of the other. If the space is our constraint, then we have to choose a program that requires less space at the cost of more execution time. On the other hand, if time is our constraint such as in real time system, we have to choose a program that takes less time to complete its execution at the cost of more space.

BIG "OH" NOTATION

Big Oh is a characteristic scheme that measures properties of algorithm complexity performance and/or memory requirements. The algorithm complexity can be determined by eliminating constant factors in the analysis of the algorithm. Clearly, the complexity function $f(n)$ of an algorithm increases as 'n' increases.

Let us find out the algorithm complexity by analyzing the sequential searching algorithm. In the sequential search algorithm we simply try to match the target value against each value in the memory. This process will continue until we find a match or finish scanning the whole elements in the array. If the array contains 'n' elements, the maximum possible number of comparisons with the target value will be 'n' *i.e.*, the worst case. That is the target value will be found at the nth position of the array.

$$f(n) = n$$

i.e., the worst case is when an algorithm requires a maximum number of iterations or steps to search and find out the target value in the array.

The best case is when the number of steps is less as possible. If the target value is found in a sequential search array of the first position (*i.e.*, we need to compare the target value with only one element from the array)—we have found the element by executing only one iteration (or by least possible statements)

$$f(n) = 1$$

Average case falls between these two extremes (*i.e.*, best and worst). If the target value is found at the $n/2$nd position, on an average we need to compare the target value with only half of the elements in the array, so

$$f(n) = n/2$$

The complexity function f(n) of an algorithm increases as '*n*' increases. The function $f(n) = O(n)$ can be read as "*f* of *n* is big Oh of *n*" or as "$f(n)$ is of the order of *n*". The total running time (or time complexity) includes the initializations and several other iterative statements through the loop.

The generalized form of the theorem is

$$f(n) = c_k n^k + c_{k-1} n^{k-1} + c_{k-2} n^{k-2} + \ldots\ldots + c_2 n^2 + c_1 n^1 + c_0 n^0$$

Where the constant $c_k > 0$

Then, $f(n) = O(n^k)$

Based on the time complexity representation of the big Oh notation, the algorithm can be categorized as :

1. Constant time O(1)
2. Logarithmic time Olog(*n*)
3. Linear time O(*n*)
4. Polynomial time $O(n^c)$
5. Exponential time $O(c^n)$

} Where $c > 1$

LIMITATION OF BIG "OH" NOTATION

Big Oh Notation has following two basic limitations :

1. It contains no effort to improve the programming methodology. Big Oh Notation does not discuss the way and means to improve the efficiency of the program, but it helps to analyze and calculate the efficiency (by finding time complexity) of the program.
2. It does not exhibit the potential of the constants. For example, one algorithm is taking $1000n^2$ time to execute and the other n^3 time. The first algorithm is $O(n^2)$, which implies that it will take less time than the other algorithm which is $O(n^3)$. However in actual execution the second algorithm will be faster for $n < 1000$.

We will analyze and design the problems in data structure. As we have discussed to develop a program of an algorithm, we should select an appropriate data structure for that algorithm.