

---

# UNIT 1 ANALYSIS OF ALGORITHMS

---

| Structure                             | Page Nos. |
|---------------------------------------|-----------|
| 1.0 Introduction                      | 7         |
| 1.1 Objectives                        | 7         |
| 1.2 Mathematical Background           | 8         |
| 1.3 Process of Analysis               | 12        |
| 1.4 Calculation of Storage Complexity | 18        |
| 1.5 Calculation of Time Complexity    | 19        |
| 1.6 Summary                           | 21        |
| 1.7 Solutions/Answers                 | 22        |
| 1.8 Further Readings                  | 22        |

---

## 1.0 INTRODUCTION

---

A common person's belief is that a computer can do anything. This is far from truth. In reality, computer can perform only certain predefined instructions. The formal representation of this model as a sequence of instructions is called an algorithm, and coded algorithm, in a specific computer language is called a program. Analysis of algorithms has been an area of research in computer science; evolution of very high speed computers has not diluted the need for the design of time-efficient algorithms.

Complexity theory in computer science is a part of theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are *time* (how many steps (time) does it take to solve a problem) and *space* (how much memory does it take to solve a problem). It may be noted that complexity theory differs from computability theory, which deals with whether a problem can be solved or not through algorithms, regardless of the resources required.

Analysis of Algorithms is a field of computer science whose overall goal is understand the complexity of algorithms. While an extremely large amount of research work is devoted to the worst-case evaluations, the focus in these pages is methods for average-case. One can easily grasp that the focus has shifted from computer to computer programming and then to creation of an algorithm. This is algorithm design, heart of problem solving.

---

## 1.1 OBJECTIVES

---

After going through this unit, you should be able to:

- understand the concept of algorithm;
- understand the mathematical foundation underlying the analysis of algorithm;
- to understand various asymptotic notations, such as Big O notation, theta notation and omega (big O,  $\Theta$ ,  $\Omega$ ) for analysis of algorithms;
- understand various notations for defining the complexity of algorithm;
- define the complexity of various well known algorithms, and
- learn the method to calculate time complexity of algorithm.

---

## 1.2 MATHEMATICAL BACKGROUND

---

To analyse an algorithm is to determine the amount of resources (such as time and storage) that are utilized by to execute. Most algorithms are designed to work with inputs of arbitrary length.

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

### Definition of Algorithm

Algorithm should have the following five characteristic features:

1. Input
2. Output
3. Definiteness
4. Effectiveness
5. Termination.

Therefore, an algorithm can be defined as a sequence of definite and effective instructions, which terminates with the production of correct output from the given input.

### Complexity classes

All decision problems fall into sets of comparable complexity, called complexity classes.

The complexity class P is the set of decision problems that can be solved by a deterministic machine in polynomial time. This class corresponds to set of problems which can be effectively solved in the worst cases. We will consider algorithms belonging to this class for analysis of time complexity. Not all algorithms in these classes make practical sense as many of them have higher complexity. These are discussed later.

The complexity class NP is a set of decision problems that can be solved by a non-deterministic machine in polynomial time. This class contains many problems like Boolean satisfiability problem, Hamiltonian path problem and the Vertex cover problem.

### What is Complexity?

Complexity refers to the rate at which the required storage or consumed time grows as a function of the problem size. The absolute growth depends on the machine used to execute the program, the compiler used to construct the program, and many other factors. We would like to have a way of describing the inherent complexity of a program (or piece of a program), independent of machine/compiler considerations. This means that we must not try to describe the absolute time or storage needed. We must instead concentrate on a “proportionality” approach, expressing the complexity in terms of its relationship to some known function. This type of analysis is known as **asymptotic analysis**. It may be noted that we are dealing with complexity of an algorithm not that of a problem. For example, the simple problem could have high order of time complexity and vice-versa.

## Asymptotic Analysis

Asymptotic analysis is based on the idea that as the problem size grows, the complexity can be described as a simple proportionality to some known function. This idea is incorporated in the “Big O”, “Omega” and “Theta” notation for asymptotic performance.

The notations like “Little Oh” are similar in spirit to “Big Oh” ; but are rarely used in computer science for asymptotic analysis.

## Tradeoff between space and time complexity

We may sometimes seek a tradeoff between space and time complexity. For example, we may have to choose a data structure that requires a lot of storage in order to reduce the computation time. Therefore, the programmer must make a judicious choice from an informed point of view. The programmer must have some verifiable basis based on which a data structure or algorithm can be selected. Complexity analysis provides such a basis.

We will learn about various techniques to bind the complexity function. In fact, our aim is not to count the exact number of steps of a program or the exact amount of time required for executing an algorithm. In theoretical analysis of algorithms, it is common to estimate their complexity in asymptotic sense, i.e., to estimate the complexity function for reasonably large length of input ‘n’. Big O notation, omega notation  $\Omega$  and theta notation  $\Theta$  are used for this purpose. In order to measure the performance of an algorithm underlying the computer program, our approach would be based on a concept called asymptotic measure of complexity of algorithm. There are notations like big O,  $\Theta$ ,  $\Omega$  for asymptotic measure of growth functions of algorithms. The most common being big-O notation. The asymptotic analysis of algorithms is often used because time taken to execute an algorithm varies with the input ‘n’ and other factors which may differ from computer to computer and from run to run. The essences of these asymptotic notations are to bind the growth function of time complexity with a function for sufficiently large input.

## The $\Theta$ -Notation (Tight Bound)

This notation bounds a function to within constant factors. We say  $f(n) = \Theta(g(n))$  if there exist positive constants  $n_0$ ,  $c_1$  and  $c_2$  such that to the right of  $n_0$  the value of  $f(n)$  always lies between  $c_1g(n)$  and  $c_2g(n)$ , both inclusive. The Figure 1.1 gives an idea about function  $f(n)$  and  $g(n)$  where  $f(n) = \Theta(g(n))$ . We will say that the function  $g(n)$  is asymptotically tight bound for  $f(n)$ .

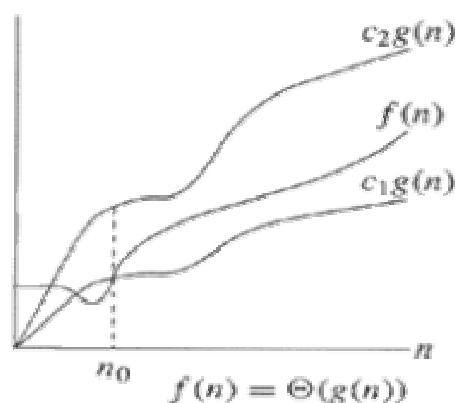


Figure 1.1 : Plot of  $f(n) = \Theta(g(n))$

For example, let us show that the function  $f(n) = \frac{1}{3}n^2 - 4n = \Theta(n^2)$ .

Now, we have to find three positive constants,  $c_1$ ,  $c_2$  and  $n_0$  such that

$$c_1 n^2 \leq \frac{1}{3} n^2 - 4n \leq c_2 n^2 \text{ for all } n \geq n_0$$

$$\Rightarrow c_1 \leq \frac{1}{3} - \frac{4}{n} \leq c_2$$

By choosing  $n_0 = 1$  and  $c_2 \geq 1/3$  the right hand inequality holds true.

Similarly, by selecting  $n_0 = 13$   $c_1 \leq 1/39$ , the right hand inequality holds true. So, for  $c_1 = 1/39$ ,  $c_2 = 1/3$  and  $n_0 \geq 13$ , it follows that  $1/3 n^2 - 4n = \Theta(n^2)$ .

Certainly, there are other choices for  $c_1$ ,  $c_2$  and  $n_0$ . Now we may show that the function  $f(n) = 6n^3 \neq \Theta(n^2)$ .

To prove this, let us assume that  $c_3$  and  $n_0$  exist such that  $6n^3 \leq c_3 n^2$  for  $n \geq n_0$ . But this fails for sufficiently large  $n$ . Therefore  $6n^3 \neq \Theta(n^2)$ .

### The big O notation (Upper Bound)

This notation gives an upper bound for a function to within a constant factor. Figure 1.2 shows the plot of  $f(n) = O(g(n))$  based on big O notation. We write  $f(n) = O(g(n))$  if there are positive constants  $n_0$  and  $c$  such that to the right of  $n_0$ , the value of  $f(n)$  always lies on or below  $cg(n)$ .

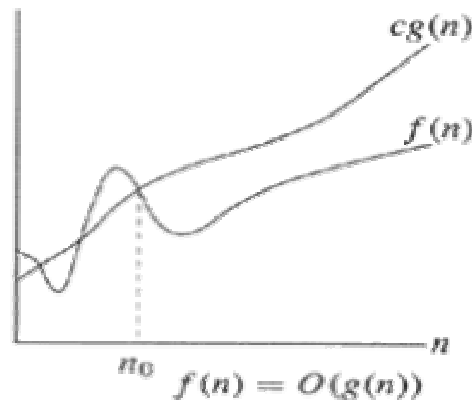


Figure 1.2: Plot of  $f(n) = O(g(n))$

Mathematically for a given function  $g(n)$ , we denote a set of functions by  $O(g(n))$  by the following notation:

$$O(g(n)) = \{f(n) : \text{There exists a positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

Clearly, we use  $O$ -notation to define the upper bound on a function by using a constant factor  $c$ .

We can see from the earlier definition of  $\Theta$  that  $\Theta$  is a tighter notation than big- $O$  notation.

$f(n) = an + c$  is  $O(n)$  is also  $O(n^2)$ , but  $O(n)$  is asymptotically tight whereas  $O(n^2)$  is notation.

Whereas in terms of  $\Theta$  notation, the above function  $f(n)$  is  $\Theta(n)$ . As big- $O$  notation is upper bound of function, it is often used to describe the worst case running time of algorithms.

### The $\Omega$ -Notation (Lower Bound)

This notation gives a lower bound for a function to within a constant factor. We write  $f(n) = \Omega(g(n))$ , if there are positive constants  $n_0$  and  $c$  such that to the right of  $n_0$ , the value of  $f(n)$  always lies on or above  $cg(n)$ . Figure 1.3 depicts the plot of  $f(n) = \Omega(g(n))$ .

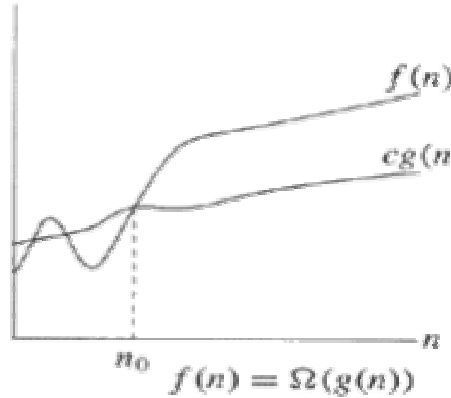


Figure 1.3: Plot of  $f(n) = \Omega(g(n))$

Mathematically for a given function  $g(n)$ , we may define  $\Omega(g(n))$  as the set of functions.

$\Omega(g(n)) = \{f(n) : \text{there exists a constant } c \text{ and } n_0 \geq 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$ .

Since  $\Omega$  notation describes lower bound, it is used to bound the best case running time of an algorithm.

### Asymptotic notation

Let us define a few functions in terms of above asymptotic notation.

Example:  $f(n) = 3n^3 + 2n^2 + 4n + 3$   
 $= 3n^3 + 2n^2 + O(n)$ , as  $4n + 3$  is of  $O(n)$   
 $= 3n^3 + O(n^2)$ , as  $2n^2 + O(n)$  is  $O(n^2)$   
 $= O(n^3)$

Example:  $f(n) = n^2 + 3n + 4$  is  $O(n^2)$ , since  $n^2 + 3n + 4 < 2n^2$  for all  $n > 10$ .

By definition of big- $O$ ,  $3n + 4$  is also  $O(n^2)$ , too, but as a convention, we use the tighter bound to the function, i.e.,  $O(n)$ .

Here are some rules about big- $O$  notation:

1.  $f(n) = O(f(n))$  for any function  $f$ . In other words, every function is bounded by itself.
2.  $a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 = O(n^k)$  for all  $k \geq 0$  and for all  $a_0, a_1, \dots, a_k \in \mathbb{R}$ . In other words, every polynomial of degree  $k$  can be bounded by the function  $n^k$ . Smaller order terms can be ignored in big- $O$  notation.
3. Basis of Logarithm can be ignored in big- $O$  notation i.e.  $\log_a n = O(\log_b n)$  for any bases  $a, b$ . We generally write  $O(\log n)$  to denote a *logarithm*  $n$  to any base.

4. Any logarithmic function can be bounded by a polynomial i.e.  $\log_b n = O(n^c)$  for any  $b$  (base of logarithm) and any positive exponent  $c > 0$ .
5. Any polynomial function can be bounded by an exponential function i.e.  $n^k = O(b^n)$ .
6. Any exponential function can be bound by the factorial function. For example,  $a^n = O(n!)$  for any base  $a$ .

### Check Your Progress 1

- 1) The function  $9n+12$  and  $1000n+400000$  are both  $O(n)$ . True/False
- 2) If a function  $f(n) = O(g(n))$  and  $h(n) = O(g(n))$ , then  $f(n)+h(n) = O(g(n))$ . True/False
- 3) If  $f(n) = n^2 + 3n$  and  $g(n) = 6000n + 34000$  then  $O(f(n)) < O(g(n))$ . True/False
- 4) The asymptotic complexity of algorithms depends on hardware and other factors. True/False
- 5) Give simplified *big-O* notation for the following growth functions:
  - $30n^2$
  - $10n^3 + 6n^2$
  - $5n \log n + 30n$
  - $\log n + 3n$
  - $\log n + 32$

.....

.....

.....

.....

---

## 1.3 PROCESS OF ANALYSIS

---

The objective analysis of an algorithm is to find its efficiency. Efficiency is dependent on the resources that are used by the algorithm. For example,

- CPU utilization (Time complexity)
- Memory utilization (Space complexity)
- Disk usage (I/O)
- Network usage (bandwidth).

There are two important attributes to analyse an algorithm. They are:

*Performance:* How much time/memory/disk/network bandwidth is actually used when a program is run. This depends on the algorithm, machine, compiler, etc.

*Complexity:* How do the resource requirements of a program or algorithm scale (the growth of resource requirements as a function of input). In other words, what happens to the performance of an algorithm, as the size of the problem being solved gets larger and larger? For example, the time and memory requirements of an algorithm which

computes the sum of 1000 numbers is larger than the algorithm which computes the sum of 2 numbers.

*Time Complexity:* The maximum time required by a *Turing machine* to execute on any input of length **n**.

*Space Complexity:* The amount of storage space required by an algorithm varies with the size of the problem being solved. The space complexity is normally expressed as an order of magnitude of the size of the problem, e.g.,  $O(n^2)$  means that if the size of the problem (**n**) doubles then the working storage (memory) requirement will become four times.

### Determination of Time Complexity

#### *The RAM Model*

The random access model (RAM) of computation was devised by John von Neumann to study algorithms. Algorithms are studied in computer science because they are independent of machine and language.

We will do all our design and analysis of algorithms based on RAM model of computation:

- Each “simple” operation (+, -, =, if, call) takes exactly 1 step.
- Loops and subroutine calls are *not* simple operations, but depend upon the size of the data and the contents of a subroutine.
- Each memory access takes exactly 1 step.

The complexity of algorithms using big-O notation can be defined in the following way for a problem of size **n**:

- Constant-time method is “order 1”:  $O(1)$ . The time required is constant independent of the input size.
- Linear-time method is “order **n**”:  $O(n)$ . The time required is proportional to the input size. If the input size doubles, then, the time to run the algorithm also doubles.
- Quadratic-time method is “order **N** squared”:  $O(n^2)$ . The time required is proportional to the square of the input size. If the input size doubles, then, the time required will increase by four times.

The process of analysis of algorithm (program) involves analyzing each step of the algorithm. It depends on the kinds of statements used in the program.

Consider the following example:

#### *Example 1: Simple sequence of statements*

Statement 1;  
Statement 2;  
...  
...  
Statement **k**;

The total time can be found out by adding the times for all statements:

Total time = time(statement 1) + time(statement 2) + ... + time(statement **k**).

It may be noted that time required by each statement will greatly vary depending on whether each statement is simple (involves only basic operations) or otherwise. Assuming that each of the above statements involve only basic operation, the time for each simple statement is constant and the total time is also constant:  $O(1)$ .

*Example 2: if-then-else statements*

In this example, assume the statements are simple unless noted otherwise.

if-then-else statements

```
if (cond) {  
    sequence of statements 1  
}  
else {  
    sequence of statements 2  
}
```

In this, if-else statement, either sequence 1 will execute, or sequence 2 will execute depending on the boolean condition. The worst-case time in this case is the slower of the two possibilities. For example, if sequence 1 is  $O(N^2)$  and sequence 2 is  $O(1)$ , then the worst-case time for the whole if-then-else statement would be  $O(N^2)$ .

*Example 3: for loop*

```
for (i = 0; i < n; i++) {  
    sequence of statements  
}
```

Here, the loop executes  $n$  times. So, the sequence of statements also executes  $n$  times. Since we assume the time complexity of the statements are  $O(1)$ , the total time for the loop is  $n * O(1)$ , which is  $O(n)$ . Here, the number of statements does not matter as it will increase the running time by a constant factor and the overall complexity will be same  $O(n)$ .

*Example 4: nested for loop*

```
for (i = 0; i < n; i++) {  
    for (j = 0; j < m; j++) {  
        sequence of statements  
    }  
}
```

Here, we observe that, the outer loop executes  $n$  times. Every time the outer loop executes, the inner loop executes  $m$  times. As a result of this, statements in the inner loop execute a total of  $n * m$  times. Thus, the time complexity is  $O(n * m)$ . If we modify the conditional variables, where the condition of the inner loop is  $j < n$  instead of  $j < m$  (i.e., the inner loop also executes  $n$  times), then the total complexity for the nested loop is  $O(n^2)$ .

Example 4: Now, consider a function that calculates partial sum of an integer  $n$ .

```
int psum(int n)  
{  
    int i, partial_sum;
```



```

partial_sum = 0; /* Line 1 */
for (i = 1; i <= n; i++) { /* Line 2 */
    partial_sum = partial_sum + i*i; /* Line 3 */
}
return partial_sum; /* Line 4 */
}

```

This function returns the sum from  $i = 1$  to  $n$  of  $i$  squared, i.e.  $psum = 1^2 + 2^2 + 3^2 + \dots + n^2$ . As we have to determine the running time for each statement in this program, we have to count the number of statements that are executed in this procedure. The code at line 1 and line 4 are one statement each. The **for loop** on line 2 are actually  $2n+2$  statements:

- $i = 1$ ; statement : simple assignment, hence one statement.
- $i \leq n$ ; statement is executed once for each value of  $i$  from 1 to  $n+1$  (till the condition becomes false). The statement is executed  $n+1$  times.
- $i++$  is executed once for each execution of the body of the loop. This is executed  $n$  times.

Thus, the sum is  $1 + (n+1) + n+1 = 2n+3$  times.

In terms of big-O notation defined above, this function is  $O(n)$ , because if we choose  $c=3$ , then we see that  $cn > 2n+3$ . As we have already noted earlier, big-O notation only provides a upper bound to the function, it is also  $O(n \log(n))$  and  $O(n^2)$ , since  $n^2 > n \log(n) > 2n+3$ . However, we will choose the smallest function that describes the order of the function and it is  $O(n)$ .

By looking at the definition of Omega notation and Theta notation, it is also clear that it is of  $\Theta(n)$ , and therefore  $\Omega(n)$  too. Because if we choose  $c=1$ , then we see that  $cn < 2n+3$ , hence  $\Omega(n)$ . Since  $2n+3 = O(n)$ , and  $2n+3 = \Omega(n)$ , it implies that  $2n+3 = \Theta(n)$ , too.

It is again reiterated here that smaller order terms and constants may be ignored while describing asymptotic notation. For example, if  $f(n) = 4n+6$  instead of  $f(n) = 2n+3$  in terms of big-O,  $\Omega$  and  $\Theta$ , this does not change the order of the function. The function  $f(n) = 4n+6 = O(n)$  (by choosing  $c$  appropriately as 5);  $4n+6 = \Omega(n)$  (by choosing  $c = 1$ ), and therefore  $4n+6 = \Theta(n)$ . The essence of this analysis is that in these asymptotic notation, we can count a statement as one, and should not worry about their relative execution time which may depend on several hardware and other implementation factors, as long as it is of the order of 1, i.e.  $O(1)$ .

*Exact analysis of insertion sort:*

Let us consider the following pseudocode to analyse the exact runtime complexity of insertion sort.

| <u>Line</u> | <u>Pseudocode</u>                 | <u>Cost factor</u> | <u>No. of iterations</u> |
|-------------|-----------------------------------|--------------------|--------------------------|
| 1           | For j=2 to length [A] do          | $c1$               | $(n-1) + 1$              |
| 2           | { key = A[j]                      | $c2$               | $(n-1)$                  |
| 3           | i = j - 1                         | $c3$               | $(n-1)$                  |
| 4           | while (i > 0) and (A[i] > key) do | $c4$               | $\sum_{j=2}^n T_j$       |
| 5           | { A[i+1] = A[i]                   | $c4$               | $\sum_{j=2}^n T_j - 1$   |

|   |                          |      |                        |
|---|--------------------------|------|------------------------|
| 6 | $i = I - 1 \}$           | $c5$ | $\sum_{j=2}^n T_j - 1$ |
| 7 | $A[I+1] = \text{key} \}$ | $c6$ | $n-1$                  |
|   | }                        |      |                        |

$T_j$  is the time taken to execute the statement during  $j^{\text{th}}$  iteration.

The statement at line 4 will execute  $T_j$  number of times.

The statements at lines 5 and 6 will execute  $T_j - 1$  number of times (one step less) each

Line 7 will execute  $(n-1)$  times

So, total time is the sum of time taken for each line multiplied by their cost factor.

$$T(n) = c1n + c2(n-1) + c3(n-1) + c4 \sum_{j=2}^n T_j + c5 \sum_{j=2}^n T_j - 1 + c6 \sum_{j=2}^n T_j - 1 + c7(n-1)$$

Three cases can emerge depending on the initial configuration of the input list. First, the case is where the list was already sorted, second case is the case wherein the list is sorted in reverse order and third case is the case where in the list is in random order (unsorted). The best case scenario will emerge when the list is already sorted.

**Worst Case:** Worst case running time is an upper bound for running time with any input. It guarantees that, irrespective of the type of input, the algorithm will not take any longer than the worst case time.

**Best Case :** It guarantees that under any circumstances the running time of algorithms will at least take this much time.

**Average case :** This gives the average running time of algorithm. The running time for any given size of input will be the average number of operations over all problem instances for a given size.

**Best Case :** If the list is already sorted then  $A[i] \leq \text{key}$  at line 4. So, rest of the lines in the inner loop will not execute. Then,

$T(n) = c1n + c2(n-1) + c3(n-1) + c4(n-1) = O(n)$ , which indicates that the time complexity is linear.

**Worst Case:** This case arises when the list is sorted in reverse order. So, the boolean condition at line 4 will be true for execution of line 1.

So, step line 4 is executed  $\sum_{j=2}^n j = n(n+1)/2 - 1$  times

$$T(n) = c1n + c2(n-1) + c3(n-1) + c4(n(n+1)/2 - 1) + c5(n(n-1)/2) + c6(n(n-1)/2) + c7(n-1)$$

$$= O(n^2).$$

**Average case :** In most of the cases, the list will be in some random order. That is, it neither sorted in ascending or descending order and the time complexity will lie somewhere between the best and the worst case.

$$T(n)_{\text{best}} < T(n)_{\text{Avg.}} < T(n)_{\text{worst}}$$

Figure 1.4 depicts the best, average and worst case run time complexities of algorithms.

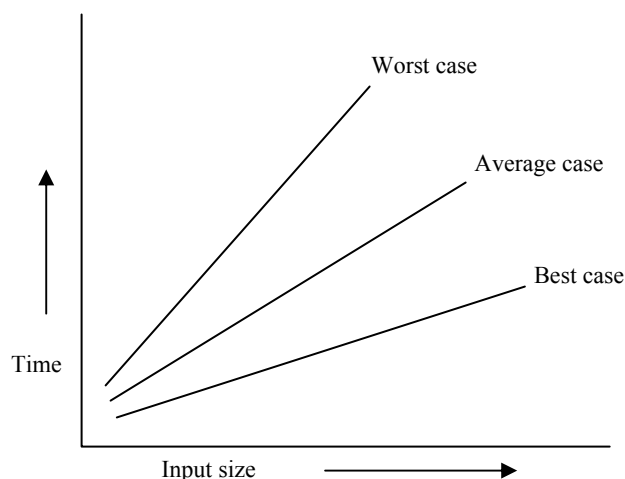


Figure 1.4 : Best, Average and Worst case scenarios

## 👉 Check Your Progress 2

- 1) The set of algorithms whose order is  $O(1)$  would run in the same time. True/False
- 2) Find the complexity of the following program in big O notation:

```
printMultiplicationTable(int max){
    for(int i = 1 ; i <= max ; i ++ )
    {
        for(int j = 1 ; j <= max ; j ++ )
            cout << (i * j) << " ";
        cout << endl ;
    } //for
}
```

- 3) Consider the following program segment:

```
for (i = 1; i <= n; i *= 2)
{
    j = 1;
}
```

What is the running time of the above program segment in big O notation?

- 4) Prove that if  $f(n) = n^2 + 2n + 5$  and  $g(n) = n^2$  then  $f(n) = O(g(n))$ .

- 5) How many times does the following **for loop** will run

```
for (i=1; i<= n; i*2)
    k = k + 1;
end;
```

---

## 1.4 CALCULATION OF STORAGE COMPLEXITY

---

As memory is becoming more and more cheaper, the prominence of runtime complexity is increasing. However, it is very much important to analyse the amount of memory used by a program. If the running time of algorithms is not good then it will

take longer to execute. But, if it takes more memory (the space complexity is more) beyond the capacity of the machine then the program will not execute at all. It is therefore more critical than run time complexity. But, the matter of respite is that memory is reutilized during the course of program execution.

We will analyse this for recursive and iterative programs.

For an iterative program, it is usually just a matter of looking at the variable declarations and storage allocation calls, e.g., number of variables, length of an array etc.

The analysis of recursive program with respect to space complexity is more complicated as the space used at any time is the total space used by all recursive calls active at that time.

Each recursive call takes a constant amount of space and some space for local variables and function arguments, and also some space is allocated for remembering where each call should return to. General recursive calls use linear space. That is, for  $n$  recursive calls, the space complexity is  $O(n)$ .

Consider the following example: *Binary Recursion* (A *binary-recursive* routine (potentially) calls itself twice).

1. If  $n$  equals 0 or 1, then return 1
2. Recursively calculate  $f(n-1)$
3. Recursively calculate  $f(n-2)$
4. Return the sum of the results from steps 2 and 3.

Time Complexity:  $O(\exp n)$

Space Complexity:  $O(\exp n)$

**Example:** Find the greatest common divisor (GCD) of two integers,  $m$  and  $n$ .  
The algorithm for GCD may be defined as follows:

While  $m$  is greater than zero:  
If  $n$  is greater than  $m$ , swap  $m$  and  $n$ .  
Subtract  $n$  from  $m$ .  
 $n$  is the GCD

#### Code in C

```
int gcd(int m, int n)
/* The precondition are :  $m > 0$  and  $n > 0$ . Let  $g = gcd(m, n)$ . */
{ while(  $m > 0$  )
  {
    if(  $n > m$  )
      { int  $t = m$ ;  $m = n$ ;  $n = t$ ; } /* swap  $m$  and  $n$  */
    /*  $m \geq n > 0$  */
     $m -= n$ ;
  }
  return  $n$ ;
}
```

The space-complexity of the above algorithm is a constant. It just requires space for three integers  $m$ ,  $n$  and  $t$ . So, the space complexity is  $O(1)$ .

The time complexity depends on the loop and on the condition whether  $m > n$  or not. The real issue is, how many iterations take place? The answer depends on both  $m$  and  $n$ .

Best case: If  $m = n$ , then there is just one iteration.  $O(1)$

Worst case : If  $n = 1$ , then there are  $m$  iterations; this is the worst-case (also equivalently, if  $m = 1$  there are  $n$  iterations)  $O(n)$ .

The *space complexity* of a computer program is the amount of memory required for its proper execution. The important concept behind space required is that unlike time, space can be reused during the execution of the program. As discussed, there is often a trade-off between the time and space required to run a program.

In formal definition, the space complexity is defined as follows:

*Space complexity* of a Turing Machine: The (worst case) maximum length of the tape required to process an input string of length  $n$ .

In complexity theory, the class *PSPACE* is the set of decision problems that can be solved by a Turing machine using a polynomial amount of memory, and unlimited time.

### Check Your Progress 3

- 1) Why space complexity is more critical than time complexity?

.....  
.....

- 2) What is the space complexity of Euclid Algorithm?

.....  
.....

## 1.5 CALCULATION OF TIME COMPLEXITY

**Example 1: Consider the following of code :**

```
x = 4y + 3
z = z + 1
p = 1
```

As we have seen,  $x$ ,  $y$ ,  $z$  and  $p$  are all scalar variables and the running time is constant irrespective of the value of  $x, y, z$  and  $p$ . Here, we emphasize that each line of code may take different time, to execute, but the bottom line is that they will take constant amount of time. Thus, we will describe run time of each line of code as  $O(1)$ .

**Example 2: Binary search**

Binary search in a sorted list is carried out by dividing the list into two parts based on the comparison of the key. As the search interval halves each time, the iteration takes place in the search. The search interval will look like following after each iteration  $N, N/2, N/4, N/8, \dots, 8, 4, 2, 1$

The number of iterations (number of elements in the series) is not so evident from the above series. But, if we take logs of each element of the series, then

$\log_2 N, \log_2 N - 1, \log_2 N - 2, \log_2 N - 3, \dots, 3, 2, 1, 0$

As the sequence decrements by 1 each time the total elements in the above series are  $\log_2 N + 1$ . So, the number of iterations is  $\log_2 N + 1$  which is of the order of  $O(\log_2 N)$ .

### Example 3: Travelling Salesman problem

Given:  $n$  connected cities and distances between them

Find: tour of minimum length that visits every city.

Solutions: How many tours are possible?

$$n * (n-1) * \dots * 1 = n!$$

Because  $n! > 2^{(n-1)}$

So  $n! = \Omega(2^n)$  (lower bound)

As of now, there is no algorithm that finds a tour of minimum length as well as covers all the cities in polynomial time. However, there are numerous very good heuristic algorithms.

*The complexity Ladder:*

- $T(n) = O(1)$ . This is called constant growth.  $T(n)$  does not grow at all as a function of  $n$ , it is a constant. For example, array access has this characteristic.  $A[i]$  takes the same time independent of the size of the array  $A$ .
- $T(n) = O(\log_2(n))$ . This is called logarithmic growth.  $T(n)$  grows proportional to the base 2 logarithm of  $n$ . Actually, the base of logarithm does not matter. For example, binary search has this characteristic.
- $T(n) = O(n)$ . This is called linear growth.  $T(n)$  grows linearly with  $n$ . For example, looping over all the elements in a one-dimensional array of  $n$  elements would be of the order of  $O(n)$ .
- $T(n) = O(n \log(n))$ . This is called **nlogn** growth.  $T(n)$  grows proportional to  $n$  times the base 2 logarithm of  $n$ . Time complexity of Merge Sort has this characteristic. In fact, no sorting algorithm that uses comparison between elements can be faster than  $n \log n$ .
- $T(n) = O(n^k)$ . This is called polynomial growth.  $T(n)$  grows proportional to the  $k$ -th power of  $n$ . We rarely consider algorithms that run in time  $O(n^k)$  where  $k$  is bigger than 2, because such algorithms are very slow and not practical. For example, selection sort is an  $O(n^2)$  algorithm.
- $T(n) = O(2^n)$ . This is called exponential growth.  $T(n)$  grows exponentially. Exponential growth is the most-danger growth pattern in computer science. Algorithms that grow this way are basically useless for anything except for very small input size.

Table 1.1 compares various algorithms in terms of their complexities.

Table 1.2 compares the typical running time of algorithms of different orders.

The growth patterns above have been listed in order of increasing size.

That is,  $O(1) < O(\log(n)) < O(n \log(n)) < O(n^2) < O(n^3), \dots, O(2^n)$ .

| Notation | Name     | Example               |
|----------|----------|-----------------------|
| $O(1)$   | Constant | Constant growth. Does |

|               |                                   |  |
|---------------|-----------------------------------|--|
|               |                                   | not grow as a function of n. For example, accessing array for one element A[i] |
| $O(\log n)$   | Logarithmic                       | Binary search  |
| $O(n)$        | Linear                            | Looping over n elements, of an array of size n (normally).                     |
| $O(n \log n)$ | Sometimes called “linearithmic”   | Merge sort   |
| $O(n^2)$      | Quadratic                         | Worst time case for insertion sort, matrix multiplication                      |
| $O(n^c)$      | Polynomial, sometimes “geometric” |  |
| $O(c^n)$      | Exponential                       |  |
| $O(n!)$       | Factorial                         |  |

Table 1.1 : Comparison of various algorithms and their complexities

| Array size | Logarithmic:<br>$\log_2 N$ | Linear:<br>N | Quadratic:<br>$N^2$ | Exponential:<br>$2^N$  |
|------------|----------------------------|--------------|---------------------|------------------------|
| 8          | 3                          | 8            | 64                  | 256                    |
| 128        | 7                          | 128          | 16,384              | $3.4 \times 10^{38}$   |
| 256        | 8                          | 256          | 65,536              | $1.15 \times 10^{77}$  |
| 1000       | 10                         | 1000         | 1 million           | $1.07 \times 10^{301}$ |
| 100,000    | 17                         | 100,000      | 10 billion          | .....                  |

Table 1.2: Comparison of typical running time of algorithms of different orders

## 1.6 SUMMARY

Computational complexity of algorithms are generally referred to by space complexity (space required for running program) and time complexity (time required for running the program). In the field of computer of science, the concept of runtime complexity has been studied vigorously. Enough research is being carried out to find more efficient algorithms for existing problems. We studied various asymptotic notation, to describe the time complexity and space complexity of algorithms, namely the *big-O*, *Omega* and *Theta* notations. These asymptotic orders of time and space complexity describe how best or worst an algorithm is for a sufficiently large input.

We studied about the process of calculation of runtime complexity of various algorithms. The exact analysis of insertion sort was discussed to describe the best case, worst case and average case scenario.

## 1.7 SOLUTIONS / ANSWERS

### Check Your Progress 1

- 1) True

- 2) True
- 3) False
- 4) False
- 5)  $O(n^2)$ ,  $O(n^3)$ ,  $O(n \log n)$ ,  $O(\log n)$ ,  $O(\log n)$

### Check Your Progress 2

- 1) True
- 2)  $O(\max*(2*\max))=O(2*\max*\max) = O(2 * n * n) = O(2n^2) = O(n^2)$
- 3)  $O(\log(n))$
- 5)  $\log n$

### Check Your Progress 3

- 1) If the running time of algorithms is not good, then it will take longer to execute. But, if it takes more memory (the space complexity is more) beyond the capacity of the machine then the program will not execute.
- 2)  $O(1)$ .

---

## 1.8 FURTHER READINGS

---

- 1. *Fundamentals of Data Structures in C++*; E.Horowitz, Sahni and D.Mehta; Galgotia Publications.
- 2. *Data Structures and Program Design in C*; Kruse, C.L.Tonodo and B.Leung; Pearson Education.

### Reference Websites

[http://en.wikipedia.org/wiki/Big\\_O\\_notation](http://en.wikipedia.org/wiki/Big_O_notation)  
<http://www.webopedia.com>