
UNIT 2: ASYMPTOTIC BOUNDS

Structure	Page Nos.
2.0 Introduction	62
2.1 Objective	63
2.2 Asymptotic Notations	63
2.2.1 Theta Notation (θ)	
2.2.2 Big Oh Notation (O)	
2.2.3 Big Omega Notation (Ω)	
2.2.4 Small o Notation (o)	
2.2.5 Small Omega Notation (ω)	
2.3 Concept of efficiency analysis of algorithm	76
2.4 Comparison of efficiencies of algorithms	79
2.5 Summary	80
2.6 Model Answers	81
2.7 Further Readings	84

2.0 INTRODUCTION

In previous unit of the block, we have discussed definition of an algorithm and several characteristics to describe an algorithm. An algorithm provides an approach to solve a given problem. The key components of an algorithm are input, processing and output. Generally all algorithms work well for small size input irrespective of the complexity. So we need to analyze the algorithm for large value of input size. It is also possible that one problem has many algorithmic solutions. To select the best algorithm for an instance of task or input we need to compare the algorithm to find out how long a particular solution will take to generate the desired output. We will determine the behavior of function and running time of an algorithm as a function of input size for large value of n . This behavior can be expressed using asymptotic notations. To understand concepts of the asymptotic notations you will be given an idea of lower bound, upper bound and how to represent time complexity expression for various algorithms. This is like expressing cost component of an algorithm. The basic five asymptotic notations will be discussed here to represent complexity expression in this unit.

In the second section, analysis for efficiency of algorithm is discussed. Efficiency of algorithm is defined in terms of two parameters i.e time and space. Time complexity refers to running time of an algorithm and space complexity refers to the additional space requirement for an algorithm to be executed. Analysis will be focused on running time complexity as response time and computation time is more important as computer speed and memory size has been improved by many orders of magnitude. Time complexity depends on input size of the problem and type of input. Based on the type of data input to an algorithm complexity will be categorized as worst case, average case and best case analysis.

In the last section, linear, quadratic, polynomial and exponential algorithm efficiency will be discussed. It will help to identify that at what rate run time will grow with respect of size of the input

2.1 OBJECTIVES

After studying this unit, you should be able to:

- Asymptotic notations
- Worst case, best case and average case analysis
- Comparative analysis of Constant, Logarithmic, Linear, Quadratic and Exponential growth of an algorithm

2.2 ASYMPTOTIC NOTATIONS

Before starting the discussion of asymptotic notations, let us see the symbols that will be used through out this unit. They are summarized in the following table.

Symbol	Name
θ	Theta
Ω	Big Omega
ε	Belongs to
ω	Small Omega
\forall	for all
\exists	there exist
\Rightarrow	Implies

An algorithm is set of instruction that takes some input and after computation it generates an output in finite amount of time. This can be evaluated by a variety of criteria and parameters. For performance analysis of an algorithm, following two complexities measures are considered:

- Space Complexity
- Time Complexity

Space complexity is amount of memory require to run an algorithm. This is sum of fixed part and variable part of a program. Here a fixed part refers to instruction space, constants and variables where as a variable part refers to instance characteristics i.e recursion, run time variables etc. Computer speed and memory size has been improved by many orders of magnitude. Hence for algorithm analysis major focus will be on time complexity.

Time complexity: Total time required to run an algorithm can be expressed as function of input size of problem and this is known as time complexity of algorithm. The limiting behavior of complexity as input size of a problem increases is called asymptotic time complexity. Total time required for completion of solving a problem is equal to sum of compile time and running time. To execute a program, always it is not mandatory that it must be compiled. Hence running time complexity will be under consideration for an evaluation and finding an algorithm complexity analysis.

Before starting with an introduction to asymptotic notations let us define the term asymptote. An asymptote provides a behavior in respect of other function for varying value of input size. An **asymptote** is a line or curve that a graph approaches but does not intersect. An asymptote of a curve is a line in such a way that distance between curve and line approaches zero towards large values or infinity.

The figure 1 will illustrate this.

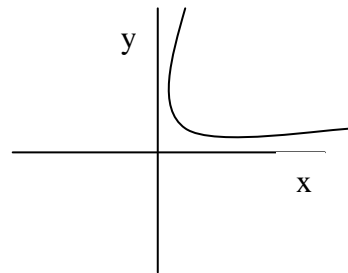


Figure:- 1

In the Figure 1, curve along x-axis and y axis approaches zero. Also the curve will not intersect the x-axis or y axis even for large values of either x or y.

Let us discuss one more example to understand the meaning of asymptote. For example x is asymptotic to $x+1$ and these two lines in the graph will never intersect as depicted in following Figure 2.

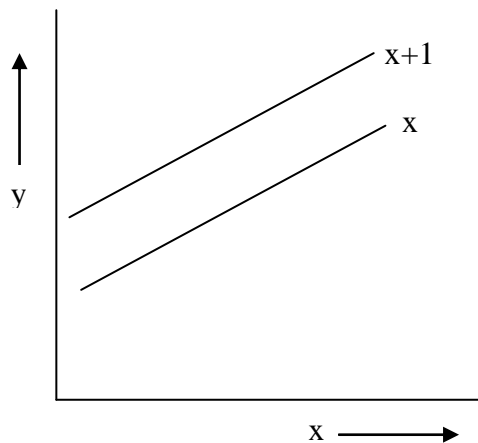


Figure:- 2

In Figure 2, x and $x+1$ are parallel lines and they will never intersect with each other. Therefore it is called as x is asymptotic to $x+1$.

The concept of asymptote will help in understanding the behavior of an algorithm for large value of input.

Now we will discuss the introduction to **bounds** that will be useful to understand the asymptotic notations.

Lower Bound: A non empty set A and its subset B is given with relation \leq . An element $a \in A$ is called lower bound of B if $a \leq x \forall x \in B$ (read as if a is less than equal to x for all x belongs to set B). For example a non empty set A and its subset B is given as $A = \{1, 2, 3, 4, 5, 6\}$ and $B = \{2, 3\}$. The lower bound of $B = 1, 2$ as $1, 2$ in the set A is less than or equal to all element of B .

Upper Bound: An element $a \in A$ is called upper bound of B if $x \leq a \forall x \in B$. For example a non empty set A and its subset B is given as $A = \{1, 2, 3, 4, 5, 6\}$ and $B = \{2, 3\}$. The upper bound of $B = 3, 4, 5, 6$ as $3, 4, 5, 6$ in the set A is greater than or equal to all element of B .

A bound (upper bound or lower bound) is said to be tight bound if the inequality is less than or equal to (\leq) as depicted in Figure 3.

Similarly a bound (lower bound or upper bound) is said to be loose bound if the inequality is strictly less than ($<$) as depicted in Figure 4.

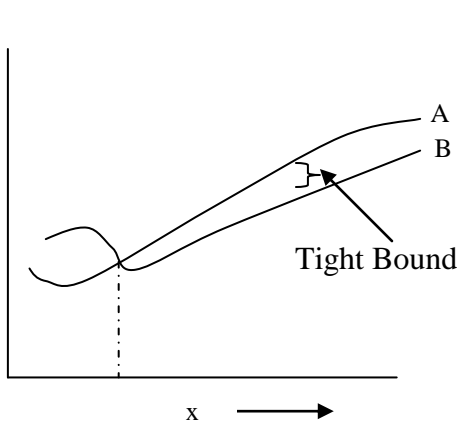


Figure:- 3 Tight Bound

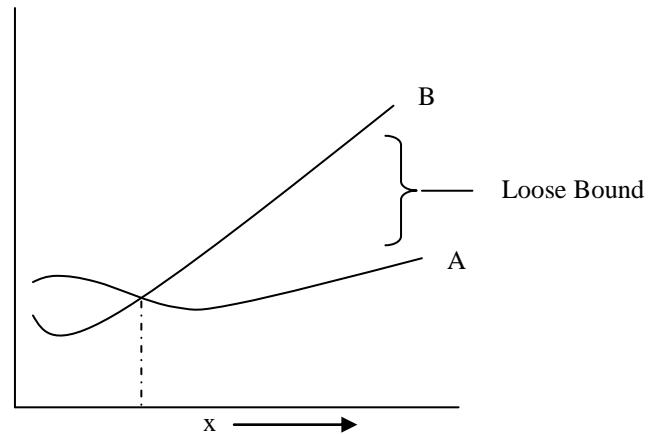


Figure:- 4 Loose Bound

For example in figure 3, distance between lines A and B is less as $B \leq A$. For large value of x, B will approach to A as it is less than or equal to A.

In Figure 4, $A < B$ i.e. distance between A and B is large. For example $A < B$, there will be distance between A and B even for large value of x as it is strictly less than only.

We also require the definition of bounded above or bounded below and bounded above & below both to understand the asymptotic notations.

Bounded above: Let A is non empty set and B is non empty subset of A. Bounded from above on B i.e supremum or least upper bound on B is defined as an upper bound of B which is less than or equal to all upper bounds of B. For example a non empty set A and its subset B is given as $A = \{1, 2, 3, 4, 5, 6\}$ and $B = \{2, 3\}$. The upper bound of B = 3, 4, 5, 6 as 3, 4, 5, 6 in the set A is greater than or equal to all element of B. Least upper bound of B is 3 i.e 3 is less than equal to all upper bounds of B.

Bounded below: Let A is non empty set and B is non empty subset of A. Bounded from below on B i.e infimum or greatest lower bound on B is defined as a lower bound of B which is greater than or equal to all lower bounds of B. For example a non empty set A and its subset B is given as $A = \{1, 2, 3, 4, 5, 6\}$ and $B = \{2, 3\}$. The lower bound of B = 1, 2 as 1, 2 in the set A is less than or equal to all element of B. Greatest lower bound of B is 2 i.e. 2 is greater than equal to all lower bounds of B.

To study the analysis of an algorithm and compute its time complexity we will be computing the total running time of an algorithm. Total running time of an algorithm is dependent on input size of the problem. Hence complexity expression will always be a function in term of input size. Hence we also require understanding the bounds in respect of function.

In respect of function defined on non empty set X , bounded above is written as $f(x) \leq A \forall x \in X$ then we say function is bounded above by A . It is read as function for all elements in the set X is less than or equal to A .

Similarly bounded below is written as $A \leq f(x) \forall x \in X$, it is said to be function is bounded below by A . It is read as function for all elements in the set X is greater than or equal to A .

A function is said to be bounded if it has both bounds i.e bounded above and below both. It is written as $A \leq f(x) \leq B \forall x \in X$.

The bounded above is depicted by figure 5 and bounded below by figure 6. Bounded above and below both is illustrated by figure 7.

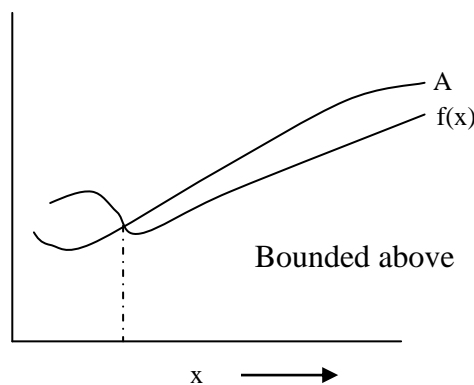


Figure:- 5 Bounded above

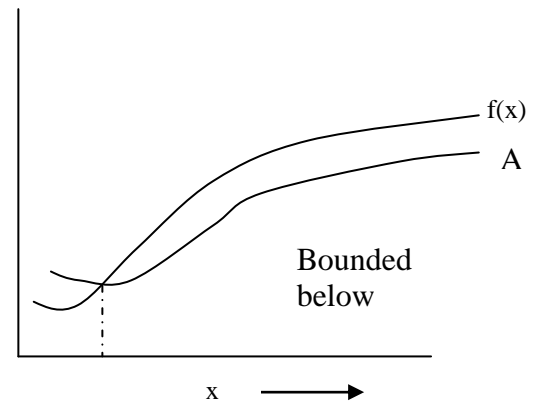


Figure:- 6 Bounded below

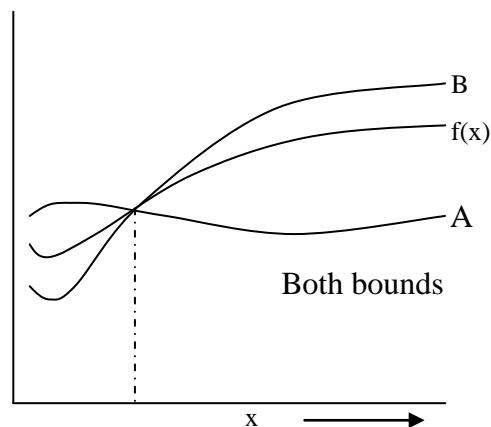


Figure:- 7 Bounded above & below

In figure 5, bounded above indicates that the value of $f(x)$ will never exceed A . It means we know the largest value of function $f(x)$ for any input value for x . Similarly in figure 6 bounded below provide the smallest value of function $f(x)$ for any input value of x . In figure 7 we get the information for smallest and largest value both. The function $f(x)$ will be in the range A and B i.e the smallest value for function $f(x)$ is A and the largest value for $f(x)$ is B . Here we know the both the values A and B i.e minimum and maximum value for $f(x)$ for any input value of x .

Now, Let us discuss the formal definitions of basic asymptotic notation which are named as θ (Theta), O (Big Oh), Ω (Big Omega), o (Small Oh), ω (Small Omega).

Let $g(n)$ be given function i.e a function in terms of input size n . In the following section we will be discussing various asymptotic notations to find the solution represented by function $f(n)$ belongs to which one of basic asymptotic notations.

2.3.1 Theta (θ) Notation

It provides both upper and lower bounds for a given function.

θ (Theta) Notation: means 'order exactly'. Order exactly implies a function is bounded above and bounded below both. This notation provides both minimum and maximum value for a function. It further gives that an algorithm will take this much of minimum and maximum time that a function can attain for any input size as illustrated in figure 7.

Let $g(n)$ be given function. $f(n)$ be the set of function defined as

$\theta(g(n)) = \{f(n): \text{if there exist positive constant } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n, n \geq n_0\}$

It can be written as $f(n) = \theta(g(n))$ or $f(n) \in \theta(g(n))$, here $f(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large values of n . It is described in the following figure 8.

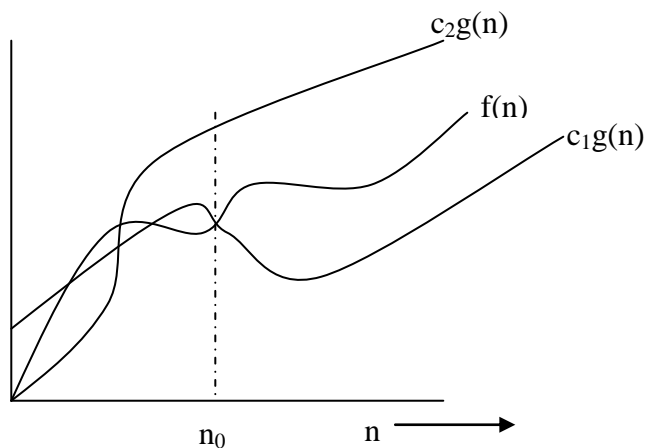


Figure:- 8 $\theta(n)$

In the figure 8 function $f(n)$ is bounded below by constant c_1 times $g(n)$ and above by constants c_2 times $g(n)$. We can explain this by following examples:

Example 1:

To show that $3n+3 = \theta(n)$ or $3n+3 \in \theta(n)$ we will verify that $f(n) \in g(n)$ or not with the help of the definition i.e

$\theta(g(n)) = \{f(n): \text{if there exist positive constant } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n, n \geq n_0\}$

In the given problem $f(n) = 3n+3$ and $g(n) = n$ to prove $f(n) \in g(n)$ we have to find c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$

\Rightarrow to verify $f(n) \leq c_2 g(n)$

We can write $f(n)=3n+3$ as $f(n)=3n+3 \leq 3n+3n$ (write $f(n)$ in terms of $g(n)$ such that mathematically inequality should be true)

$$\begin{aligned} &\leq 6n \text{ for all } n > 0 \\ c_2=6 \text{ for all } n > 0 \text{ i.e. } n_0=1 \end{aligned}$$

To verify $0 \leq c_1 g(n) \leq f(n)$

We can write $f(n)=3n+3 \geq 3n$ (again write $f(n)$ in terms of $g(n)$ such that mathematically inequality should be true)

$$\begin{aligned} &c_1=3 \text{ for all } n, n_0=1 \\ \Rightarrow 3n \leq 3n+3 \leq 6n \text{ for all } n \geq n_0, n_0=1 \end{aligned}$$

i.e we are able to find, $c_1=3, c_2=6, n_0=1$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$
So, $f(n) = \theta(g(n))$ for all $n \geq 1$

Example 2:

To show that $10n^2+4n+2=\theta(n^2)$ or $10n^2+4n+2 \in \theta(n^2)$ we will verify that $f(n) \in g(n)$ or not with the help of the definition i.e

$\theta(g(n)) = \{f(n): \text{if there exist positive constant } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n, n \geq n_0\}$

In the given problem $f(n)=10n^2+4n+2$ and $g(n)=n^2$ to prove $f(n) \in g(n)$ we have to find c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$

\Rightarrow to verify $f(n) \leq c_2 g(n)$

We can write $f(n)=10n^2+4n+2 \leq 10n^2+4n^2+2n^2$ (write $f(n)$ in terms of $g(n)$ such that mathematically inequality should be true)

$$\begin{aligned} &\leq 16n^2 \\ c_2=16 \text{ for all } n \end{aligned}$$

To verify $0 \leq c_1 g(n) \leq f(n)$ We can write $f(n)=10n^2+4n+2 \geq 10n^2$
(write $f(n)$ in terms of $g(n)$ such that mathematically inequality should be true)

$$c_1=10 \text{ for all } n, n_0=1$$

$$\Rightarrow 10n^2 \leq 10n^2+4n+2 \leq 16n^2 \text{ for all } n \geq n_0, n_0=1$$

i.e we are able to find, $c_1=10, c_2=16, n_0=1$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$

So, $f(n) = \theta(g(n))$ for all $n \geq 1$

2.3.2 Big Oh (O) Notation

This notation provides upper bound for a given function.

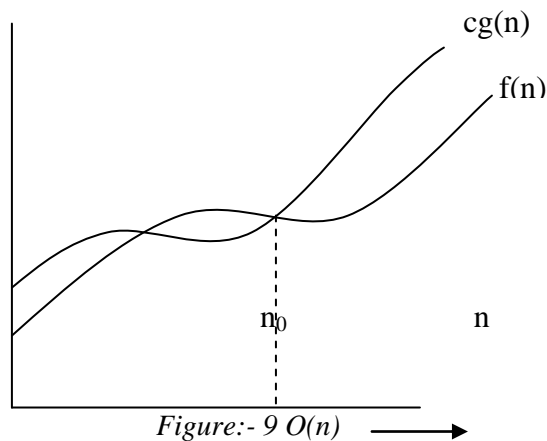
O(Big Oh) Notation: mean 'order at most' i.e bounded above or it will give maximum time required to run the algorithm.

For a function having only asymptotic upper bound, Big Oh 'O' notation is used.

Let a given function $g(n)$, $O(g(n))$ is the set of functions $f(n)$ defined as

$O(g(n)) = \{f(n) : \text{if there exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n, n \geq n_0\}$

$f(n) = O(g(n))$ or $f(n) \in O(g(n))$, $f(n)$ is bounded above by some positive constant multiple of $g(n)$ for all large values of n . The definition is illustrated with the help of figure 9.



In this figure9, function $f(n)$ is bounded above by constant c times $g(n)$. We can explain this by following examples:

Example 3:

To show $3n^2 + 4n + 6 = O(n^2)$ we will verify that $f(n) \in g(n)$ or not with the help of the definition i.e $O(g(n)) = \{f(n) : \text{if there exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n, n \geq n_0\}$

In the given problem

$$f(n) = 3n^2 + 4n + 6$$

$$g(n) = n^2$$

To show $0 \leq f(n) \leq cg(n)$ for all $n, n \geq n_0$

$$f(n) = 3n^2 + 4n + 6 \leq 3n^2 + n^2 \quad \text{for } n \geq 6$$

$$\leq 4n^2$$

$$c = 4 \text{ for all } n \geq n_0, n_0 = 6$$

i.e we can identify , $c=4, n_0=6$

So, $f(n) = O(n^2)$

Example 4:

To show $5n + 8 = O(n)$ we will verify that $f(n) \in g(n)$ or not with the help of the definition i.e $O(g(n)) = \{f(n) : \text{if there exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n, n \geq n_0\}$

In the given problem

$$f(n) = 5n + 8$$

$$g(n) = n$$

To show $0 \leq f(n) \leq cg(n)$ for all $n, n \geq n_0$

$$f(n) = 5n + 8 \leq 5n + 8n$$

$$\leq 13n$$

$$c=13 \text{ for all } n \geq n_0, n_0=1$$

i.e we can identify, $c=13, n_0=1$

So, $f(n)=O(g(n))$ i.e $f(n)=O(n)$

2.3.3 Big Omega (Ω) Notation

This notation provides lower bound for a given function.

Ω (Big Omega): mean 'order at least' i.e minimum time required to execute the algorithm or have lower bound

For a function having only asymptotic lower bound, Ω notation is used.

Let a given function $g(n)$. $\Omega(g(n))$ is the set of functions $f(n)$ defined as

$\Omega(g(n)) = \{f(n): \text{if there exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n, n \geq n_0\}$

$f(n) = \Omega(g(n))$ or $f(n) \in \Omega(g(n))$, $f(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large values of n . It is described in the following figure 10.

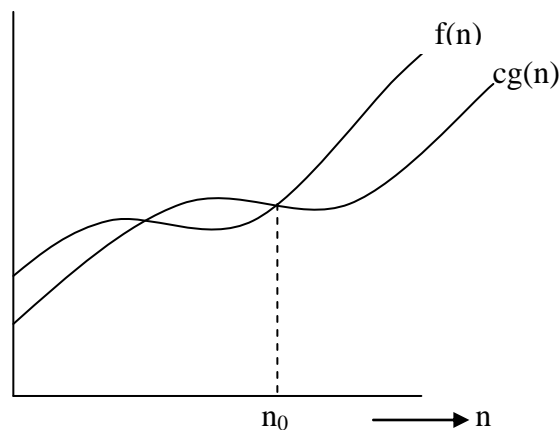


Figure:- 10 $\Omega(n)$

In this figure 10, function $f(n)$ is bounded below by constant c times $g(n)$. We can explain this by following examples:

Example 5:

To show $2n^2+4n+6 = \Omega(n^2)$ we will verify that $f(n) \in \Omega(g(n))$ or not with the help of the definition i.e $\Omega(g(n)) = \{f(n): \text{if there exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n, n \geq n_0\}$

In the given problem

$$f(n) = 2n^2 + 4n + 6$$

$$g(n) = n^2$$

To show $0 \leq cg(n) \leq f(n)$ for all $n, n \geq n_0$

We can write $f(n) = 2n^2 + 4n + 6$

$$0 \leq 2n^2 \leq 2n^2 + 4n + 6 \text{ for } n \geq 0$$

$$c=2 \text{ for all } n \geq n_0, \quad n_0=0$$

i.e we are able to find, $c=2, n_0=0$

$$\text{So, } f(n) = \Omega(n^2)$$

Example 6:

To show $n^3 = \Omega(n^2)$ we will verify that $f(n) \in g(n)$ or not with the help of the definition
i.e $\Omega(g(n)) = \{f(n): \text{if there exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n, n \geq n_0\}$

In the given problem

$$f(n) = n^3$$

$$g(n) = n^2$$

To show $0 \leq cg(n) \leq f(n)$ for all $n, n \geq n_0$

We can write

$$f(n) = n^3$$

$$0 \leq n^2 \leq n^3 \text{ for } n \geq 0$$

$$c=1 \text{ for all } n \geq n_0, \quad n_0=0$$

i.e we can select, $c=1, n_0=0$

$$\text{So, } f(n) = \Omega(n^2)$$

2.3.4 Small o (o) Notation

o(small o) Notation:

For a function that does not have asymptotic tight upper bound, o (small o) notation is used. i.e. It is used to denote an upper bound that is not asymptotically tight.

Let a given function $g(n)$, $o(g(n))$ is the set of functions $f(n)$ defined as
 $o(g(n)) = \{f(n): \text{for any positive constant } c \text{ there exist a constant } n_0 > 0 \text{ such that}$
 $0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$

$f(n) = o(g(n))$ or $f(n) \in o(g(n))$, $f(n)$ is loosely bounded above by all positive constant multiple of $g(n)$ for all large n . It is illustrated in the following figure 11.

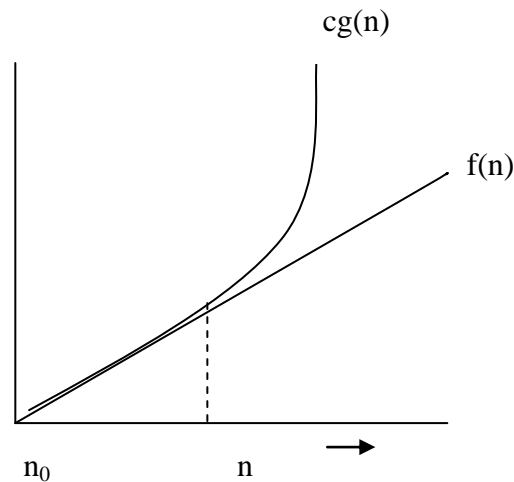


Figure:-11 $o(n)$

In this figure 11, function $f(n)$ is loosely bounded above by constant c times $g(n)$. We can explain this by following example:

Example 7:

To show $2n+4 = o(n^2)$ we will verify that $f(n) \in g(n)$ or not with the help of the definition i.e $o(g(n)) = \{f(n): \text{for any positive constant } c \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$

In the given problem

$$f(n) = 2n+4, g(n) = n^2$$

To show $0 \leq f(n) < cg(n)$ for all $n \geq n_0$ We can write as
 $f(n) = 2n+4 < cn^2$

for any $c > 0$, for all $n \geq n_0$, $n_0 = 1$

i.e we can find, $c=1$, $n_0=1$

Hence, $f(n) = o(g(n))$

Example 8:

To show $2n = o(n^2)$ we will verify that $f(n) \in g(n)$ or not with the help of the definition i.e $o(g(n)) = \{f(n): \text{for any positive constant } c \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$

In the given problem

$$f(n) = 2n, g(n) = n^2$$

To show $0 \leq f(n) < cg(n)$ for all $n \geq n_0$ We can write as

$$f(n) = 2n < cn^2$$

for any $c > 0$, for all $n \geq n_0$, $n_0 = 1$

i.e we can find, $c=1$, $n_0=1$

Hence, $f(n) = o(g(n))$

2.3.5 Small Omega (ω) Notation

ω (Small Omega) Notation:

For a function that does not have asymptotic tight lower bound, ω notation is used. i.e. It is used to denote a lower bound that is not asymptotically tight.

Let a given function $g(n)$. $\omega(g(n))$ is the set of functions $f(n)$ defined as
 $\omega(g(n)) = \{f(n): \text{for any positive constant } c > 0 \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$

$f(n) = \omega(g(n))$ or $f(n) \in \omega(g(n))$, $f(n)$ is loosely bounded below by all positive constant multiple of $g(n)$ for all large n . It is described in the following figure 11.

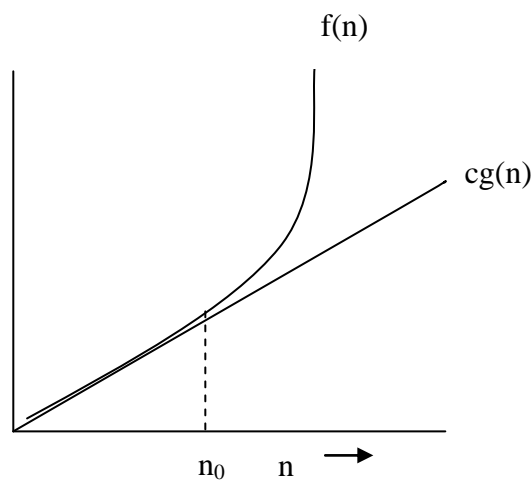


Figure:-11 $\omega(n)$

In this figure function $f(n)$ is loosely bounded below by constant c times $g(n)$. Following example illustrate this notation:

Example 9:

To show $2n^2 + 4n + 6 = \omega(n)$ we will verify that $f(n) \in \omega(g(n))$ or not with the help of the definition i.e. $\omega(g(n)) = \{f(n): \text{for any positive constant } c > 0 \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$

In the given problem

$$f(n) = 2n^2 + 4n + 6$$

$$g(n) = n$$

To show $0 \leq cg(n) < f(n)$ for all $n \geq n_0$ We can write as

$$f(n) = 2n^2 + 4n + 6$$

$cn < 2n^2 + 4n + 6$ for any $c > 0$, for all $n \geq n_0$, $n_0 = 1$
 i.e we can find, $c=1$, $n_0=1$

Hence, $f(n) = \omega(g(n))$ i.e $f(n) = \omega(n)$

Example 10:

To show $2n^3 + 3n^2 + 1 = \omega(n)$ we will verify that $f(n) \in g(n)$ or not with the help of the definition i.e $\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0 \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$

In the given problem

$$f(n) = 2n^3 + 3n^2 + 1$$

$$g(n) = n$$

To show $0 \leq cg(n) < f(n)$ for all $n \geq n_0$ We can write as

$$f(n) = 2n^3 + 3n^2 + 1$$

$$cn < 2n^3 + 3n^2 + 1 \text{ for any } c > 0, \text{ for all } n \geq n_0, n_0 = 1$$

i.e we can find, $c=1$, $n_0=1$

Hence, $f(n) = \omega(g(n))$ i.e $f(n) = \omega(n)$

Let us summarize the above asymptotic notations in the following table.

Notation Name	Mathematical inequality	Meaning
θ	+ve constant c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$	Minimum and maximum time that a function f can take
O	+ve constant c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n, n \geq n_0$	Maximum time that a function f can take
Ω	+ve constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n, n \geq n_0$	Minimum time that a function f can take
o	$c > 0$ there exist $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$	Function f will take strictly less than Maximum time
ω	$c > 0$ there exist $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$	Function f will take strictly greater than Maximum time

An algorithm complexity can be written in the form of asymptotic notations discussed in this section depending upon algorithm will fall under which notation. For example let us take a part of any algorithm where we read n element of an array.

```

1.    scanf("%d",&n);
2.    printf("Enter element for an array");
3.    for(i=0;i<n;i++)
4.        scanf("%d",&a[i]);

```

Line Number	Number of times
1	1
2	1
3	n
4	n-1

$$f(n) = 1 + 1 + n + (n-1)$$

$$f(n) = 2n + 1$$

Now to compute the complexity for the above construct of an algorithm, let us find the bounds for above function $f(n)$ i.e $2n+1$.

Let us verify whether $f(n)$ is $O(n)$, $\Omega(n)$ and $\theta(n)$.

To show $f(n) = O(n)$
 $f(n) = 2n + 1$
 $g(n) = n$
 $f(n) = 2n + 1 \leq 2n + n$ for all $n \geq 1$
 $\leq 3n$
 $c = 3$ for all $n \geq n_0$, $n_0 = 1$
i.e we can identify, $c = 3$, $n_0 = 1$

So, $f(n) = O(g(n))$ i.e $f(n) = O(n)$

To show $f(n) = \Omega(n)$

$$f(n) = 2n + 1$$

$$g(n) = n$$

$$f(n) = 2n + 1$$

$$0 \leq n \leq 2n + 1 \text{ for } n \geq 0$$

$$c = 1 \text{ for all } n \geq n_0, n_0 = 0$$

i.e we can select, $c = 1$, $n_0 = 0$

So, $f(n) = \Omega(n)$

To show $f(n) = \theta(n)$

$$f(n) = 2n + 1 \text{ and } g(n) = n$$

$$\Rightarrow f(n) = 2n + 1 \leq 2n + n \text{ for all } n \geq 1$$

$$\leq 3n$$

$$c_2 = 3 \text{ for all } n$$

$$\text{Also } f(n) = 2n + 1 \geq n \text{ for all } n \geq 1$$

$$c_1 = 1 \text{ for all } n$$

i.e we are able to find, $c_1 = 1$, $c_2 = 3$, $n_0 = 1$

So, $f(n) = \theta(g(n))$ i.e $f(n) = \theta(n)$ for all $n \geq 1$

For this construct complexity will be $f(n) = O(n)$, $f(n) = \Omega(n)$, $f(n) = \theta(n)$.

However, we will generally be most interested in the Big Oh time analysis as this analysis can lead to computation of maximum time required for the algorithm to solve the given problem.

In the next section, we will discuss about concept of efficiency analysis of an algorithm.

☞ Check Your Progress 1

1. Define the following:

a) Algorithm

.....

.....

.....

b) Time Complexity

.....

.....

.....

c) Space Complexity

.....

.....

.....

2. Define basic five asymptotic notations.

.....

.....

.....

.....

.....

3. Give an example for each asymptotic notations as defined in Q2

.....

.....

.....

.....

2.3 CONCEPT OF EFFICIENCY ANALYSIS OF ALGORITHM

If we are given an input to an algorithm we can exactly compute the number of steps our algorithm executes. We can also find the count of the processor instructions. Usually, we are interested in identifying the behavior of our program w.r.t input supplied to the algorithm. Based on type of input, analysis can be classified as following:

- Worst Case
- Average Case
- Best Case

In the **worst case** - we need to look at the input data and determine an upper bound on how long it will take to run the program. Analyzing the efficiency of an algorithm in the worst case scenario speaks about how fast the maximum runtime grows when we increase the input size. For example if we would like to sort a list of n numbers in ascending order and the list is given in descending order. It will lead to worst case scenario for the sorting algorithm.

In **average case** – we need to look at time required to run the algorithm where all inputs are equally likely. Analyzing the efficiency of an algorithm speaks about probabilistic analysis by which we find expected running time for an algorithm. For example in a list of n numbers to be sorted in ascending order, some numbers may be at their required position and some may be not in order.

In **Best case**- Input supplied to the algorithm will be almost similar to the format in which output is expected. And we need to compute the running time of an algorithm. This analysis will be referred as best case analysis. For example we would like to sort the list of n numbers in ascending order and the list is already in ascending order.

During efficiency analysis of algorithm, we are required to study the behavior of algorithm with varying input size. For doing the same, it is not always required to execute on a machine, number of steps can be computed by simulating or performing dry run on an algorithm.

For example: Consider the linear search algorithm in which we are required to search an element from a given list of elements, let's say size of the list is n .

Input: An array of n numbers and an element which is required to be searched in the given list

Output: Number exists in the list or not.

Algorithm:

1. Input the size of list i.e. n
2. Read the n elements of array A
3. Input the item/element to be searched in the given list.
4. for each element in the array $i=1$ to n
5. if $A[i]==\text{item}$
6. Search successful, return
7. if $i==n+1$
8. Search unsuccessful.
9. Stop

Efficiency analysis of the above algorithm in respect of various cases is as follows:

Worst Case: In respect of example under consideration, the worst case is when the element to be searched is either not in the list or found at the end of the list. In this case algorithm runs for longest possible time i.e maximum running time of the algorithm depends on the size of an array so, running time complexity for this case will be $O(n)$.

Average case: In this case expected running time will be computed based on the assumption that probability of occurrence of all possible input is equal i.e array elements could be in any order. This provides average amount of time required to solve a problem of size n . In respect of example under consideration, element could be found at random position in the list. Running time complexity will be $O(n)$.

Best Case: In this the running time will be fastest for given array elements of size n i.e. it gives minimum running time for an algorithm. In respect of example under consideration, element to be searched is found at first position in the list. Running time complexity for this case will be $O(1)$.

In most of the cases, average case analysis and worst case analysis plays an important role in comparison to best case. Worst case analysis defines an upper bound on running time for any input and average case analysis defines expected running time for input of given size that are equally likely.

For solving a problem we have more than one solution. Comparison among different solutions provides which solution is much better than the other i.e. which one is more efficient to solve a problem. Efficiency of algorithm depends on time taken to run the algorithm and use of memory space by the algorithm. As already discussed, focus will be on time efficiency rather than space.

Execution time of an algorithm will be computed on different sizes of n . For large values of n , constant factor will not affect the complexity of an algorithm. Hence it can be expressed as a function of size n .

For example $O(n) = O(n/2) = O(n+2)$ etc. It is read as order will be defined in terms of n irrespective of the constant factor like divide by 2 or plus 2 etc. As while discussing complexity analysis we are interested in order of algorithm complexity.

Some algorithm solution could be quadratic function of n . Other solution may be linear or exponential function of n . Different algorithm will fall under different complexity classes.

Behavior of quadratic, linear and exponential in respect of n will be discussed in next section.

Check Your Progress 2

1. Define Best case Time Complexity.

.....

.....

.....

2. Define Worst case Time Complexity.

.....

.....

.....

3. Define Average case time complexity.

.....

.....

.....

4. Write an algorithm for bubble sort and write its worst case, average case and best case analysis.

.....

.....

2.4 COMPARASION OF EFFICIENCIES OF AN ALGORITHM

Running time for most of the algorithms falls under different efficiency classes.

- | | | | |
|----|----------|---------------|---|
| a) | 1 | Constant Time | When instructions of program are executed once or at most only a few times , then the running time complexity of such algorithm is know as constant time. it is independent of the problem's size. It is represented as $O(1)$. For example, linear search best case complexity is $O(1)$ |
| b) | $\log n$ | Logarithmic | The running time of the algorithm in which large problem is solved by transforming into smaller sizes sub problems is said to be Logarithmic in nature. In this algorithm becomes slightly slower as n grows. It does not process all the data element of input size n . The running time does not double until n increases to n^2 . It is represented as $O(\log n)$. For example binary search algorithm running time complexity is $O(\log n)$. |
| c) | n | Linear | In this the complete set of instruction is executed once for each input i.e input of size n is processed. It is represented as $O(n)$. This is the best option to be used when the whole input has to be processed. In this situation time requirement increases directly with the size of the problem. For example linear search Worst case complexity is $O(n)$. |
| d) | n^2 | Quadratic | Running time of an algorithm is quadratic in nature when it process all pairs of data items. Such algorithm will have two nested loops. For input size n , running time will be $O(n^2)$. Practically this is useful for problem with small input size or elementary sorting problems. In this situation time requirement increases fast with the size of the problem. For example insertion sort running time complexity is $O(n^2)$. |
| e) | 2^n | Exponential | Running time of an algorithm is exponential in nature if brute force solution is applied to solve a problem. In such algorithm all subset of an n -element set is generated. In this situation time requirement increases very fast with the size of the problem. For input size n , running time complexity expression will be $O(2^n)$. For example Boolean variable equivalence of n variables running time complexity is $O(2^n)$. Another familiar example is Tower of Hanoi problem where running time complexity is $O(2^n)$. |

For large values of n or as input size n grows, some basic algorithm running time approximation is depicted in following table. As already discussed, worst case

analysis is more important hence O Big Oh notation is used to indicate the value of function for analysis of algorithm.

n	Constant	Logarithmic	Linear	Quadratic	Exponential
	$O(1)$	$O(\log n)$	$O(n)$	$O(n^2)$	$O(2^n)$
1	1	1	1	1	2
2	1	1	2	4	4
4	1	2	4	16	16
8	1	3	8	64	256
10	1	3	10	10^2	10^3
10^2	1	6	10^2	10^4	10^{30}
10^3	1	9	10^3	10^6	10^{301}
10^4	1	13	10^4	10^8	10^{3010}

The running time of an algorithm is most likely to be some constant multiplied by one of above function plus some smaller terms. Smaller terms will be negligible as input size n grows. Comparison given in above table has great significance for analysis of algorithm.

☞ Check Your Progress 3

1. Define basic efficiency classes.

.....

2. Write a function for implementing binary search. Also give an expression for running time complexity in terms of input size n for Worst case, Best case and average case.

.....

2.5 SUMMARY

Analysis of algorithms means to find out an algorithm's efficiency with respect to resources: running time and memory space. Time efficiency indicates how fast the algorithm executes; space efficiency deals with additional space required running the algorithm. Algorithm running time will depend on input size. This will be the number of basic operation executed for an algorithm. For the algorithm we can define worst case efficiency, best case efficiency and average case efficiency. Worst case efficiency means the algorithm runs the longest time among all possible inputs of size n. Best case efficiency the algorithm runs the fastest among all possible inputs of size n. Average case efficiency means running time for a typical/random input of size n. For example for sorting a set of element in ascending order, input given in descending order is referred as worst case and input arranged in ascending order referred as best case. Input data of mixed type/random type i.e some elements are in order and some are not in order is referred as average case.

Among all worst case analysis is important as it provides the information about maximum amount of time an algorithm requires for solving a problem of input size n . The efficiency of some algorithm may differ significantly for input of the same size.

In this unit, five basic Asymptotic notations are defined: θ (Theta), O (Big Oh), Ω (Big Omega), o (Small Oh), ω (Small Omega).

These notations are used to identify and compare asymptotic order of growth of function in respect of input size n to express algorithm efficiency.

For visualization of growth of function with respect to input size, comparison among values of some functions for analysis of algorithm is provided. In this comparison input size is taken as $2^0, 2^1, 2^2, 2^3, 10^1, 10^2, 10^3$, and 10^4 for constant, logarithmic, linear, quadratic and exponential functions.

These notation and comparison for growth of function defined and used here will be used throughout the design and analysis of an algorithm.

2.6 MODEL ANSWERS

Check Your Progress 1:

Answers:

1)

(a) Algorithm: An algorithm is set of instructions to be executed for a given input to solve a problem or generate the required output in finite amount of time. The algorithm should solve the problem correctly.

(b) Time complexity: Time complexity of an algorithm tells the amount of time required to run the algorithm as a function of input size n . Generally it is expressed using O Big Oh notation which ignores constant and smaller terms.

(c) Space complexity: Space complexity of an algorithm speaks about additional space required to run the algorithm. Good algorithm will keep this amount of additional memory used as small as possible.

2). Asymptotic Notation: It is the formal way to speak about function and classify them. Basic five notations are:

θ (Theta), O (Big Oh), Ω (Big Omega), o (Small Oh), ω (Small Omega).

θ (Theta): Let $g(n)$ be given function. $f(n)$ is the set of function defined as $\theta(g(n)) = \{f(n) : \text{if there exist positive constant } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n, n \geq n_0\}$

O (Big Oh): For a given function $g(n)$, $O(g(n))$, $f(n)$ is the set of functions defined as $O(g(n)) = \{f(n) : \text{if there exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n, n \geq n_0\}$

Ω (Big Omega): For a given function $g(n)$, $\Omega(g(n))$, $f(n)$ is the set of functions defined as $\Omega(g(n)) = \{f(n) : \text{if there exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq c g(n) \leq f(n) \text{ for all } n, n \geq n_0\}$

o (Small Oh): For a given function $g(n)$, $o(g(n))$, $f(n)$ is the set of functions defined as $o(g(n)) = \{f(n): \text{for any positive constant } c \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$

ω (Small Omega): For a given function $g(n)$, $\omega(g(n))$, $f(n)$ is the set of functions defined as $\omega(g(n)) = \{f(n): \text{for any positive constant } c > 0 \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$

3) Example for above define basic asymptotic notation:

θ (Theta): $10n^3 + 5n^2 + 17 \in \theta(n^3)$

$$10n^3 \leq 10n^3 + 5n^2 + 17 \leq (10 + 5 + 17)n^3 = 32n^3$$

$$c_1 = 10, c_2 = 32, n_0 = 1$$

$$10n^3 \leq 10n^3 + 5n^2 + 17 \leq 32n^3 \text{ for all } n \geq n_0 = 1$$

O (Big Oh): $10n^3 + 5n + 17 \in O(n^3)$

$$10n^3 + 5n + 17 \leq (10 + 5 + 17)n^3 \text{ for all } n \geq n_0 = 1 = 32n^3$$

$$c = 32, n_0 = 1$$

$$10n^3 + 5n + 17 \leq 32n^3 \text{ for all } n \geq n_0 = 1$$

Ω (Big Omega): $2n^3 + 37 \in \Omega(n^3)$

$$2n^3 \leq 2n^3 + 37 \text{ for all } n \geq n_0 = 1$$

$$c = 2, n_0 = 1$$

$$2n^3 \leq 2n^3 + 37 \text{ for all } n \geq n_0 = 1$$

o (Small Oh): $3n^2 \in o(n^3)$

$$3n^2 < n^3 \text{ for all } n \geq n_0 = 4$$

$$c = 1, n_0 = 4$$

$$3n^2 < n^3 \text{ for all } n \geq n_0 = 4$$

ω (Small Omega): $3n^3 \in \omega(n^2)$

$$cn^2 < 3n^3 \text{ for all } n \geq n_0 = 1$$

$$c = 1, n_0 = 1$$

$$n^2 < 3n^3 \text{ for all } n \geq n_0 = 1$$

4) The algorithm for bubble sort is as below:

// a is the list or an array of n elements to be sorted

function bubblesort(a,n)

{

 int i,j,temp,flag=true;

 for(i=0; i<n-1 && flag==true; i++)

 {

 flag=false

 for(j=0; j<n-i-1; j++)

 {

 if(a[j]>a[j+1])

 {

 flag=true

```

        temp = a[j];
        a[j] = a[j+1];
        a[j+1] = temp;
    }
}
}

```

Complexity analysis of bubble sort is as follows.

Best-case:

When the given data set in an array is already sorted in ascending order the number of moves/exchanges will be 0, then it will be clear that the array is already in order because no two elements need to be swapped. In that case, the sort should end, which takes $O(1)$. The total number of key comparisons will be $(n-1)$ so complexity in best case will be $O(n)$.

Worst-case:

In this case the given data set will be in descending order that need to be sorted in ascending order. Outer loop in the algorithm will be executed $n-1$ times. The number of exchanges will be $3*(1+2+\dots+n-1) = 3 * n*(n-1)/2$ i.e $O(n^2)$. The number of key comparison will be $(1+2+\dots+n-1) = n*(n-1)/2$ i.e $O(n^2)$. Hence complexity in worst case will be $O(n^2)$.


Average –case:

In this case we have to consider all possible initial data arrangement. So as in case of worst case ,outer loop will be executed $n-1$ times. The number of exchanges will be $O(n^2)$. The number of key comparison will be i.e $O(n^2)$. So the complexity will be $O(n^2)$.

Check Your Progress 3:

1) Basic efficiency classes is depicted in following table:

Running time	Function class
1	constant
$\log n$	logarithmic
n	linear
n^2	quadratic
2^n	exponential



Fast and high time efficiency

Slow and low time efficiency

2) Function for binary search is given below:

```
int binarysearch(int a[], int size, int element)
{
    int beg = 0;
    int end = size - 1;
    int mid;          // mid will be the index of target when it's found.
    while (beg <= end)
    {
        mid = (beg + end) / 2;
        if (a[mid] < element)
            beg = mid + 1;
        else if (a[mid] > element)
            end = mid - 1;
        else
            return mid;
    }
    return -1;
}
```

For unsuccessful search running time complexity will be $O(\log n)$.

For Successful search that is element to be searched is found in the list, running time complexity for different cases will be as follows:

Worst Case- $O(\log n)$

Best Case – $O(1)$

Average Case - $O(\log n)$

2.7 FURTHER READINGS

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, Clifford Stein, "Introduction to Algorithms", 2nd Ed., PHI, 2004.
2. Robert Sedgewick, "Algorithms in C", 3rd Edition, Pearson Education, 2004
3. Ellis Horowitz, Sartaj Sahani, Sanguthevar Rajasekaran, "Fundamentals of Computer algorithms", 2nd Edition, Universities Press, 2008
4. Anany Levitin, "Introduction to the Design and Analysis of Algorithm", Pearson Education, 2003.