1.8 ALGORITHM

Algorithm is a step wise solution of a problem or algorithm is advanced preparation for executable program or code. For a particular (Single) problem there may be multiple solutions are possible, so multiple algorithms are possible for a single problem. If hundred peoples individually are trying to solve a problem, they use different tricks, techniques and ideas to solve a problem, so output will be same of all peoples but definitely algorithms will be different. Now the problem is

that which better algorithm among all algorithms is, it means all the algorithm have to give test for better performance, in other words it is also called complexity of algorithm.

How to calculate complexity (performance) of algorithm, complexity can be find out by **Asymptotic Notations**.

Types of Asymptotic Notation.

1. Big oh notation.
2. Theta notation.
3. Big omega notation.
4. Little oh notation.
5. Little omega notation.

Practically algorithm cannot compile and run, so always program have a complexity and that complexity will be for equivalent algorithm. Performance of algorithm or program can be find out by two parameters one is running time(Time) and second one is runtime space(Memory in RAM) both complexity measures at **Run** time.

1.9 TYPES OF COMPLEXITY

There are two types of complexity.

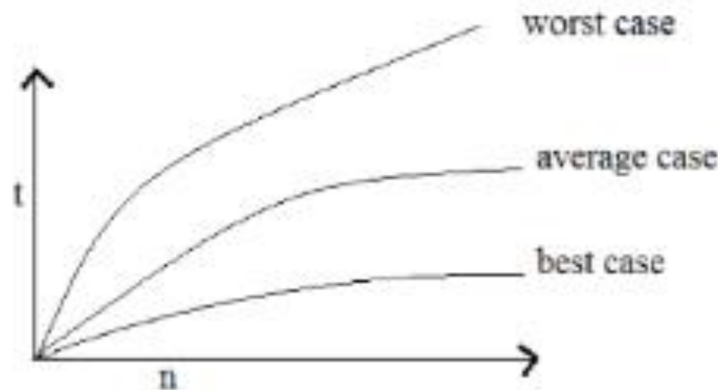
1. Time complexity.
2. Space complexity.

1.9.1 TIME COMPLEXITY

Time complexity is a total time taken by the program for execution. In this case program should produce fast output.

Cases:-Time complexity may be.

1. **Best case:-**required minimum time for solution.
2. **Average case: -** required average time for solution.
3. **Worst case: -** required maximum time for solution.



In other words program should produce more output in very less time.

1.9.1.1 Big oh notation (O)

Big oh notation is a technique to find time complexity of algorithms, Big oh notation is denoted by O (big O).

Big oh notation provide upper bound for function f(n) such that $f(n) <= c*g(n)$ for all $n_0 >= n$.

Here both f(n) and g(n) are polynomial function of n, where n is the size of input, c is a constant and n_0 is the particular point where from n_0 if we increase the value of n then the given equation $f(n) <= c*g(n)$ will be true at any point.

Remember that complexity is not affected by any constant, f(n) is the function of original problem and c*g(n) is upper bound for the

algorithm. Upper bound means **at most time** taken for execution, cannot exceed c*g(n) time for running algorithm .

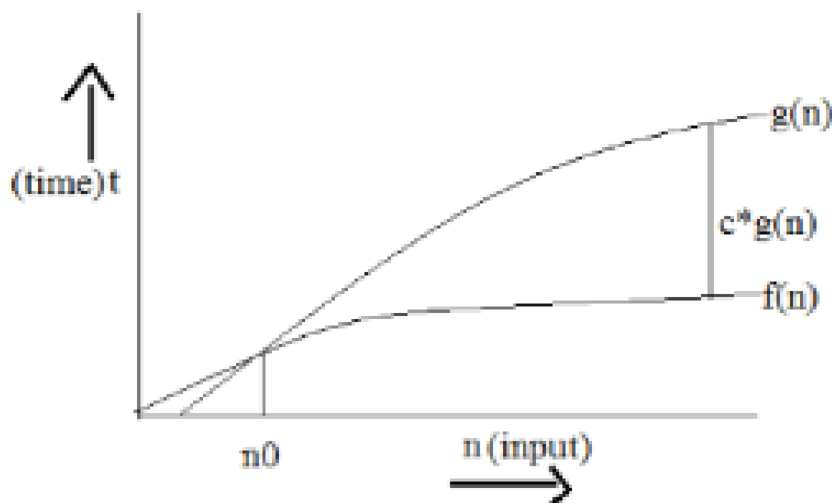
Problem Statement: - If f(n) is polynomial function in terms of n and g(n) is upper bound for function f(n) such that

$f(n) <= c*g(n)$ for all $n_0 >= n$ then

$f(n) = O(g(n))$ where c is constant and $n_0 >= n$

Here f(n) function can generated from algorithms or program step count method and g(n) is calculated from f(n) by constraint satisfactions of Big Oh notation(Remember that g(n) will be highest power of n in f(n)).

If we plot graph between Time (t) and input size n, here x axis represent input (n) and y-axis represent Time (t).



Complexity function f(n) may be look like this.

Types of function:-

S.NO	Function Name	Function Value
1	Constant function	f(n)=101(any constant)
2	Linear function	f(n)=n+5 or f(n)=2n+15
3	Quadratic function	$f(n)=5n^2+2n+4$
4	Cubic function	$f(n)=5n^3+5n^2+2n+4$
5	Exponential function	$f(n)=2^n+5n^3+5n^2+2n+4$
6	Logarithmic function	f(n)=aT(n/b)+f(n) (will see later)

Complexity looks like this.

Types of complexity:-

S.NO	Complexity Name	Complexity Value
1	Constant complexity	f(n)=O(1)
2	Linear complexity	f(n)=O(n)
3	Quadratic complexity	$f(n)=O(n^2)$
4	Cubic complexity	$f(n)=O(n^3)$
5	Exponential complexity	$f(n)=O(2^n)$
6	Logarithmic complexity	f(n)=O(log(n))

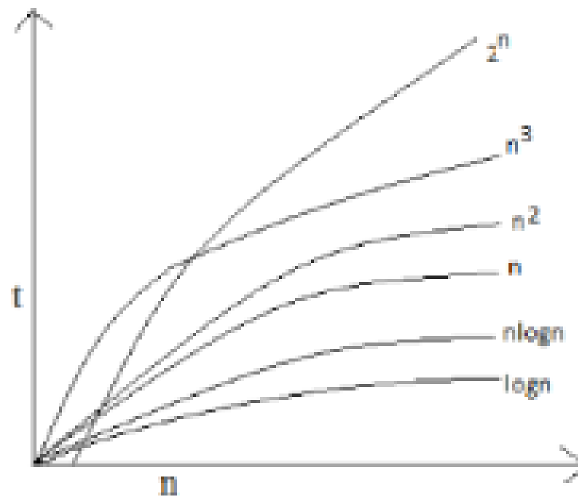
Which function having what complexity looks like this.

S.NO	Complexity Function	Complexity Value
1	f(n)=1024	f(n)=O(1)
2	f(n)=2n+15	f(n)=O(n)
3	$f(n)=5n^2+2n+4$	$f(n)=O(n^2)$
4	$f(n)=5n^3+5n^2+2n+4$	$f(n)=O(n^3)$
5	$f(n)=2^n+5n^3+5n^2+2n+4$	$f(n)=O(2^n)$
6	f(n)=2T(n/2)+n	f(n)=O(log(n))

Which is the best complexity from above?

S.NO	n	log(n)	n^2	n^3	2^n
1	1	0	1	1	2
2	2	1	4	8	4
3	4	2	16	64	16
4	8	3	64	512	256
5	16	4	256	4096	65536

For best complexity evaluation if we increase the input value then the corresponding execution time must not be increase in big amount. If we see from above table in log(n) column value increasing very slow but in 2^n column value increasing very fast in comparison on input increasing ratio. So finally log(n) is best complexity and 2^n is worst complexity.



How to calculate complexity from function f(n)?

Example1. What is the time complexity of given function?

f(n)=2n+2.

Solution:- Given f(n)=2n+2

- ⇨ 2n+2<=2n+2
- ⇨ 2n+2<=2n+n where n>=2
- ⇨ 2n+2<=3n for all n>=2
- ⇨ f(n)<=3n for all n>=2
- ⇨ Compare with the standard Big oh notation equation that is
- ⇨ f(n)<=c*g(n) for all n_0>=n
- ⇨ so here g(n)=n, c=3 and n_0=2
- ⇨ f(n)=O(g(n)) where c=3 and n_0=2
- ⇨ **f(n)=O(n)** where c=3 and n_0=2

Practical proof:-

N	**f(n)=2n+2**	**c*g(n)=3n**	**f(n)<=c*g(n)**
1	f(n)=4	3	4<=3 false

2	f(n)=6	6	6<=6 true
3	f(n)=8	9	8<=9 true
4	f(n)=10	12	10<=12 true

We can check any value of n>=2 it will produce true combination.

Example2. What is the time complexity of given function?

f(n)=n+4.

Solution:- Given f(n)=n+4

- ⇨ n+4<=n+4
- ⇨ n+4<=n+n where n>=4
- ⇨ n+4<=2n for all n>=4
- ⇨ f(n)<=2n for all n>=4
- ⇨ Compare with the standard Big oh notation equation that is
- ⇨ f(n)<=c*g(n) for all n_0>=n
- ⇨ so here g(n)=n, c=2 and n_0=4
- ⇨ f(n)=O(g(n)) where c=2 and n_0=4
- ⇨ **f(n)=O(n)** where c=4 and n_0=4

Practical proof:-

N	**f(n)=n+4**	**c*g(n)=2n**	**f(n)<=c*g(n)**
1	f(n)=5	2	5<=3 false
2	f(n)=6	4	6<=4 false
3	f(n)=7	6	7<=6 false
4	f(n)=8	8	8<=8 **true**
5	f(n)=9	10	9<=10 **true**
6	f(n)=10	12	10<=12 **true**

Example3. What is the time complexity of given function?

$f(n)= 5n^2+3n+4$

Solution:- Given $f(n)=5n^2+3n+4$

- ⇨ $5n^2+3n+4<= 5n^2+3n+4$
- ⇨ $5n^2+3n+4<= 5n^2+3n+n$ where $n>=4$
- ⇨ $5n^2+2n+4<= 5n^2+4n$ for all $n>=4$
- ⇨ $f(n)<= 5n^2+n^2$ for all $n^2>=4n$
- ⇨ $f(n)<= 6n^2$ for all $n>=4$
- ⇨ Compare with the standard Big oh notation equation that is
- ⇨ $f(n)<=c*g(n)$ for all $n_0>=n$
- ⇨ so here $g(n)= n^2$, $c=6$ and $n_0=4$
- ⇨ $f(n)=O(g(n^2))$ where $c=6$ and $n_0=4$
- ⇨ **$f(n)=O(n^2)$** where $c=6$ and $n_0=4$

Example4. What is the time complexity of given function?

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

Solution:- Given $f(n)= n^3+n^2+n+4$

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

- ⇨ $f(n)<= n^3+n^2+n+n$ for all $n>=4$
- ⇨ $f(n)<= n^3+n^2+2n$ for all $n>=4$
- ⇨ $f(n)<= n^3+n^2+n^2$ for all $n^2>=2n$
- ⇨ $f(n)<=n^3+2n^2$ for all $n>=2$
- ⇨ $f(n)<=n^3+n^3$ for all $n^3>=2n^2$
- ⇨ $f(n)<=2n^3$ for all $n>=2$
- ⇨ Compare with the standard Big oh notation equation that is
- ⇨ $f(n)<=c*g(n)$ for all $n_0>=n$

- ⇨ so here $g(n)= n^3$, $c=2$ and $n_0=2$
- ⇨ $f(n)=O(g(n))$ where $c=2$ and $n_0=2$
- ⇨ **$f(n)=O(n^3)$** where $c=2$ and $n_0=2$

Example5. What is the time complexity of given function?

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

Solution:-

Given **$f(n)= 2^n+n^3+n^2+n+4$**

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

- ⇨ $f(n)<= 2^n+n^3+n^2+n+n$ for all $n>=4$
- ⇨ $f(n)<= 2^n+n^3 +n^2 +2n$ for all $n>=4$
- ⇨ $f(n)<= 2^n+n^3 +n^2+n^2$ for all $n^2>=2n$
- ⇨ $f(n)<= 2^n+n^3+2n^2$ for all $n>=2$
- ⇨ $f(n)<= 2^n+n^3+n^3$ for all $n^3>=2n^2$
- ⇨ $f(n)<= 2^n+2n^3$ for all $n>=2$
- ⇨ $f(n)<=2^n+2*2^n$ for all $n^3>=2^n$
- ⇨ $f(n)<=3*2^n$ for all $n>=10$
- ⇨ Compare with the standard Big oh notation equation that is
- ⇨ $f(n)<=c*g(n)$ for all $n_0>=n$
- ⇨ so here $g(n)= 2^n$, $c=3$ and $n_0=10$
- ⇨ $f(n)=O(g(n))$ where $c=3$ and $n_0=10$
- ⇨ **$f(n)=O(2^n)$** where $c=3$ and $n_0=10$

Example6:-prove that: if $f(n)\epsilon O(n)$ then $[f(n)]^2\epsilon O(n^2)$

Solution:- If $f(n)\epsilon O(n)$, let $f(n)=n+1=O(n)$

So, $[f(n)]^2=(n+1)^2$

$[f(n)]^2 \leq(n+1)(n+1)$

$\leq n^2+2n+1$

$\leq n^2+2n+n,\ n\geq 1$

$\leq n^2+3n \quad ,\ n\geq 1$

$\leq n^2+n^2 \quad ,\ n\geq 3$

$\leq 2n^2 \quad ,\ n\geq 3$

⇨ $[f(n)]^2=O(n^2)$ where c=2 and $n\geq 3$

Example7. What is the time complexity of given function?

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

Solution:- Given $f(n)=n^2+4$

- ⇨ $n^2+4<= n^2+4$
- ⇨ $n^2+4<= n^2+n^2$ where $n^2>=4$
- ⇨ $n^2+4<= 2n^2$ for all $n>=2$
- ⇨ $f(n)<= n^2+n^2$ for all $n^2>=4$
- ⇨ $f(n)<= 2n^2$ for all $n>=2$
- ⇨ Compare with the standard Big oh notation equation that is
- ⇨ $f(n)<=c*g(n)$ for all $n_0>=n$
- ⇨ so here $g(n)= n^2$, c=2 and $n_0=2$
- ⇨ $f(n)=O(g(n^2))$ where c=2 and $n_0=2$
- ⇨ **$f(n)=O(n^2)$** where c=2 and $n_0=2$

Example8. What is the time complexity of given function?

$f(n)= n^3+8$

Solution:- Given $f(n)= n^3+8$

$f(n)<= n^3+8$

⇨ $f(n)<= n^3+ n^3$ for all $n^3>=8$

⇨ $f(n) <= 2n^3$ for all n>=2
⇨ Compare with the standard Big oh notation equation that is
⇨ $f(n) <= c*g(n)$ for all $n_0 >= n$
⇨ so here $g(n) = n^3$, c=2 and $n_0=2$
⇨ $f(n)=O(g(n))$ where c=2 and $n_0=2$
⇨ **$f(n)=O(n^3)$** where c=2 and $n_0=2$

How to build function f(n) from algorithm or program?

In any application most of the common operations are fetching values and assign into some parameters, data transfer, data update, data search, data delete, validations and calculations etc. majorly above factors affected execution time of program. Remember any number of variables only declared in our program will not affect execution time until they are not initialize because when variable initialize then variable occupied memory in RAM.

Rules of step count method?

Element	Step count	Complexity
Variable declaration	0	O(1)
Variable initialize	1	O(1)
Condition(if-else)	1	O(1)
Loops	n+1	O(n)
Loop inside loop	(n+1)(n+1)	$O(n^2)$

Example

Element	Step count	Complexity
int a,b,c;	0	O(1)
a=10;	1	O(1)
if(a<b)	1	O(1)
for(i=1;i<=n;i++)	n+1(times condition checked)	O(n)
for(i=1;i<=n;i++){ for(j=1;j<=n;j++)}	(n+1)(n+1)(times condition checked)	$O(n^2)$

Program1:-Sum of two numbers

```
void main() --------step count
{
int a,b,c;  ------------0
a=10;    -------------1
b=20; ---------------1
c=a+b; ---------------1
}
```

So f(n)=0+1+1+1

f(n)=3

⇨ f(n)=O(1)

Program2:-Sum of two numbers using function

```
void add() --------step count
{
int a,b,c;  ------------0
a=10;     -------------1
b=20; ----------------1
c=a+b; ---------------1
}
```

So f(n)=0+1+1+1

f(n)=3

⇨ f(n)=O(1)

Program3:-Sum of two numbers using function

```
int add() --------step count
{
int a,b,c;  ------------0
a=10;     -------------1
b=20; ----------------1
c=a+b; ---------------1
return c;--------------1
}
```

So f(n)=0+1+1+1+1

f(n)=4

⇨ f(n)=O(1)

Program4:-print 1 to 10 using loop

void main() --------step count

{

int i ; ----------------------0

i=1; ---------------------1

for(i<=n){ ----------------n+1

printf("%d",i++)---------n

}}

So f(n)=0+1+n+1+n

f(n)=2n+2

⇨ f(n)=O(n)

Program5:-sum of array elements

int add(a,n) --------step count

{

int i,s ; ----------------------0

s=0; ----------------------1

for(i=1;i<=n;i++){ --------n+1

s=s+a[i]; ------------------n

2.1 The Efficiency of Algorithm

GTU : Winter-12, Marks 7

The efficiency of algorithm can be specified using **time efficiency** and **space efficiency** which is known as time complexity and space complexity

Time complexity of an algorithm means the amount of time taken by an algorithm to run.

Space complexity of an algorithm means amount of space(memory) taken by an algorithm

Importance of Efficiency Analysis

Performing efficiency analysis is important for these following two reasons -

1. By computing the time complexity we come to know whether algorithm is slow or fast.
2. By computing the space complexity we can analyze whether an algorithm requires more or less space.

Concept of Frequency Count

The time complexity of an algorithm can be computed using the frequency count.

Definition : The frequency count is a count that denotes how many times particular statement is executed.

Consider following code for counting the frequency count

```
void fun()
{
 int a;
 a=10;
 printf("%d",a);
}
```

```
void fun()
{
 int a;
 a=10; ..................................................Executes once
 printf("%d",a); ........................................Execute Once
}
```

The frequency count of above program is **2.**

```
for(i=0;i<n;i++)
 {
    a  = a+i;
 }
 printf("%d",a); }
```

Solution :

```
void fun()
{
  int a;
  a=0; ....................................................1
  for(i=0;i<n;i++) ....................................n+1
   {
      a  = a+i; ..........................................n
   }
   printf("%d",a); .......................................1
}
```

The frequency count of above code is **2n + 3**.

The for loop in above given fragment of code is executed n times when the condition is true and one more time when the condition becomes false. Hence for the for loop the frequency count is n + 1. The statement inside the for loop will be executed only when the condition inside the for loop is true. Therefore this statement will be executed for n times. The last printf statement will be executed for once.

Example 2.1.2 *Obtain the frequency count for the following code.*

OR

*Using Step count method analyze the time complexity when two m*n matrices are added.*

GTU : Winter-14, Marks 7

```
void fun(int a[][],int b[][])
{
 int c[3][3];
  for(i=0;i<m;i++)
  {
    for(j=0;j<n;j++)
    {
      c[i][j]=a[i][j]+b[i][j];
    }        }
```

Solution :

```
void fun(int a[][],int b[][])
{
 int c[3][3];
  for(i=0;i<m;i++) .......................................m+1
  {
    for(j=0;j<n;j++) ....................................m(n+1)
    {
```

```
   c[i][j]=a[i][j]+b[i][j]; ..........................m.n
  }
 }
```

The frequency count = (m + 1) + m(n + 1) + mn = 2m + 2mn + 1 = **2m(1 + n) + 1**

Example 2.1.3 *Obtain the frequency count for the following code.*

```
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
c[i][j]=0;
for(k=1;k<=n;k++)
c[i][j]=c[i][j]+a[i][k]*b[k][j];
}
}
```

Dec.-2012

Solution :

Statement	Frequency Count
for(i=1;i<=n;i++)	n + 1
for(j=1;j<=n;j++)	n.(n + 1)
c[i][j]=0;	n.(n)
for(k=1;k<=n;k++)	n.n(n + 1)
c[i][j]=c[i][j]+a[i][k]*b[k][j];	n.n.n
Total	$2n^3 + 3n^2 + 2n + 1$

After counting the frequency count, the constant terms can be neglected and only the order of magnitude is considered. The time complexity is denoted in terms of algorithmic notations. The Big oh notation is a most commonly used algorithmic notation. For the above frequency count all the constant terms are neglected and only the order of magnitude of the polynomial is considered. Hence the time complexity for the above code can be $O(n^3)$. The higher order is the polynomial is always considered.

After counting the frequency count, the constant terms can be neglected and only the order of magnitude is considered. The time complexity is denoted in terms of algorithmic notations. The Big oh notation is a most commonly used algorithmic notation. For the above frequency count all the constant terms are neglected and only the order of magnitude of the polynomial is considered. Hence the time complexity for the above code can be $O(n^3)$. The higher order is the polynomial is always considered.

Example 2.1.4 *Obtain the frequency count for the following code*

```
i=1;
do
{
a++;
 if(i==5)
break;
i++;
 }while(i<=n)
```

Solution :

Statement	Frequency Count
i=1;	1
a++;	5
if(i==5)	5
break;	1
i++;	5
while(i<=n)	5
Total	22

Example 2.1.5 *What is space complexity ? How algorithms can be analyzed in terms of space complexity ? Will it depend on type of instance (input) or will change ?*

GTU : Summer -14, Marks 4

Solution : The space complexity can be defined as amount of memory required by an algorithm to run.

To compute the space complexity we use two factors : **Constant and instance characteristics**. The space requirement S(p) can be given as,

$$S(p) = C + Sp$$

where **C** is a constant i.e. fixed part and it denotes the space of inputs and outputs. This space is an amount of space taken by instruction, variables and identifiers. And **Sp** is a space dependent upon instance characteristics. This is a variable part whose space requirement depends on particular problem instance. Thus space complexity is dependant upon the type of input.

Consider an example of algorithm to compute the space complexity.

```
Algorithm Sum(a,n)
{
    s : = 0.0 ;
        For i : = 1 to n do
            s : = s + a[i] ;
    return s
}
```

In the given code we require space for

```
s : = 0         ←    O(1)
For i : = 1 to n   ←    O(n)
```

```
s : = s + a[i] ;    ←    O(n)
returns ;    ←    O(1)
```

Hence the space complexity of given algorithm can be denoted in terms of big-oh notation. It is **O(n).**

Review Question

1. *Explain why analysis of algorithms is important ?* — **GTU : Winter-12, Marks 7**

2.2 Average and Worst Case Analysis

GTU : Dec.-10, May-12, Marks 4

If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called **best case** time complexity.

For example : While searching a particular element by using sequential search we get the desired element at first place itself then it is called best case time complexity.

If an algorithm takes maximum amount of time to run to completion for a specific set of input then it is called **worst case** time complexity.

For example : While searching an element by using linear searching method if desired element is placed at the end of the list then we get worst time complexity.

The time complexity that we get for certain set of inputs is as a average same. Then for corresponding input such a time complexity is called **average case** time complexity.

Consider the following algorithm

```
Algorithm Seq_search(X[0 ... n – 1 ],key)
// Problem Description : This algorithm is for searching the
//key element from an array X[0...n – 1] sequentially.
//Input : An array X[0...n – 1] and search key
//Output : Returns the index of X where key value is present
for i ← 0 to n – 1 do
 if(X[i]=key)then
  return i
```

Best case time complexity

Best case time complexity is a time complexity when an algorithm runs for short time. In above searching algorithm the element **key** is searched from the list of **n** elements. If the **key** element is present at first location in the list(X[0...n–1]) then algorithm runs for a very short time and thereby we will get the best case time complexity. We can denote the best case time complexity as

$$C_{best} = 1$$

Worst case time complexity

Worst case time complexity is a time complexity when algorithm runs for a longest time. In above searching algorithm the element **key** is searched from the list of **n** elements. If the **key** element is present at n^{th} location then clearly the algorithm will run for longest time and thereby we will get the worst case time complexity. We can denote the worst case time complexity as

$$C_{worst} = n$$

The algorithm guarantees that for any instance of input which is of size n, the running time will not exceed $C_{worst}(n)$. Hence the worst case time complexity gives important information about the efficiency of algorithm.

Average case time complexity

This type of complexity gives information about the behaviour of an algorithm on specific or random input. Let us understand some terminologies that are required for computing average case time complexity.

Let the algorithm is for sequential search and

P be a probability of getting successful search.

n is the total number of elements in the list.

The first match of the element will occur at i^{th} location. Hence probability of occurring first match is P/n for every i^{th} element.

The probability of getting unsuccessful search is $(1 - P)$.

Now, we can find average case time complexity $C_{avg}(n)$ as -

$$C_{avg}(n) = \text{Probability of successful search (for elements 1 to n in the list)} + \text{Probability of unsuccessful search}$$

$$C_{avg}(n) = \left[1 \cdot \frac{P}{n} + 2 \cdot \frac{P}{n} + \ldots + i \cdot \frac{P}{n}\right] + n \cdot (1 - P)$$

$$= \frac{P}{n}[1 + 2 + \ldots + i \ldots n] + n\,(1 - P)$$

There may be n elements at which chances of 'not getting element' are possible.

$$= \frac{P}{n}\,\frac{n\,(n+1)}{2} + n\,(1 - P)$$

$$C_{avg}(n) = \frac{P\,(n+1)}{2} + n\,(1 - P)$$

Thus we can obtain the general formula for computing average case time complexity.

Suppose if P = 0 that means there is no successful search i.e. we have scanned the entire list of n elements and still we do not found the desired element in the list then in such a situation,

$$C_{avg}(n) = 0(n + 1)/2 + n\ (1 - 0)$$

$$C_{avg}(n) = n$$

Thus the average case running time complexity becomes equal to n.

Suppose if P = 1 i.e. we get a successful search then

$$C_{avg}(n) = 1\ (n + 1)/2 + n\ (1 - 1)$$

$$C_{avg}(n) = (n + 1)/2$$

That means the algorithm scans about half of the elements from the list.

For calculating average case time complexity we have to consider probability of getting success of the operation. And any operation in the algorithm is heavily dependent on input elements. Thus computing average case time complexity is difficult than computing worst case and best case time complexities.

Review Questions

1. *What is an algorithm ? Explain various properties of an algorithm.* **GTU : Dec.-10, Marks 3**
2. *Explain why analysis of algorithms is important ? Explain worst case, best case and average case Complexity.* **GTU : May-12, Marks 4**
3. *Explain : Worst case, best case and average case complexity.* **GTU : Winter- 12, Marks 7**

2.3 Elementary Operations

Elementary operations are those operations whose execution time is bounded by a constant which depends upon the type of implementation used. The elementary operations are **addition, multiplication and assignment**.

Let, for implementing an algorithm requires some elementary operations such as addition, multiplication and assignment.

Let,

a be the number of additions

b be the number of multiplications

c be the number of assignments

t1 be the total amount of time required by addition operations

t2 be the total amount of time required by multiplication operations

t3 be the total amount of time required by assignment operations

The total time required by the algorithm to execute can be expressed as,

$$t \le at1 + bt2 + ct3 \quad \text{or}$$

$$t \le \max\{t1, t2, t3\} \times \{a + b + c\}$$

Thus **t** is bounded by a constant multiple of time taken by elementary operations to execute.

Example

```
Algorithm SUM(n)
{
    //Problem Description: This algorithm calculates the sum of integers from 1 to n
    //Input: Integer values from 1 to n
    //Output: returns the sum value
    sum ← 0
    for (i ← 1 to n ) do
        sum ← sum+i
    return sum;
}
```

In above algorithm the elementary operation is **addition**. If a machine of 32-bit words is used to execute above algorithm, then all the additions can be executed directly provided n with no greater than 65535. Theoretically we consider that the additions costs for **n units.**

2.4 Asymptotic Notations

GTU : May-11,12, Dec.-10,11, Summer-13,14, Winter-14,15, Marks 6

To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity.

Using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" or "average time".

Various notations such as Ω, Θ and O used are called **asymptotic notions**.

2.4.1 Big oh Notation

The **Big oh** notation is denoted by 'O'. It is a method of representing the **upper bound of** algorithm's running time. Using big oh notation we can give longest amount of time taken by the algorithm to complete.

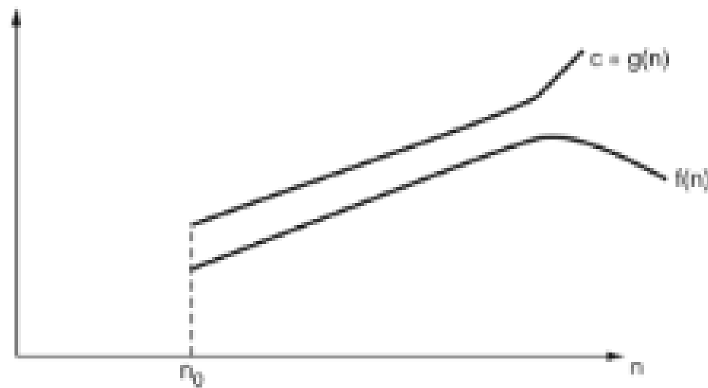
Definition

Let f(n) and g(n) be two non-negative functions.

Let n_0 and constant c are two integers such that n_0 denotes some value of input and $n > n_0$. Similarly c is some constant such that $c > 0$. We can write

$$f(n) \le c * g(n)$$

then f(n) is big oh of g(n). It is also denoted as $f(n) \in O(g(n))$. In other words f(n) is less than g(n) if g(n) is multiple of some constant c.



$f(n) \in O(g(n))$

Fig. 2.4.1

Example : Consider function f(n) = 2n + 2 and $g(n) = n^2$. Then we have to find some constant c, so that $f(n) \le c * g(n)$. As f(n) = 2n + 2 and $g(n) = n^2$ then we find c for

n = 1 then,

$$f(n) = 2n + 2$$
$$= 2(1) + 2$$
$$f(n) = 4$$

and
$$g(n) = n^2$$
$$= (1)^2$$
$$g(n) = 1$$

i.e. $f(n) > g(n)$

If n = 2 then,

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

i.e. $f(n) > g(n)$

If n = 3 then,

$$f(n) = 2(3) + 2$$
$$= 8$$

$g(n) = (3)^2$

$g(n) = 9$

i.e. $f(n) < g(n)$ is true.

Hence we can conclude that for $n > 2$, we obtain

$f(n) < g(n)$

Thus always upper bound of existing time is obtained by big oh notation.

2.4.2 Omega Notation

Omega notation is denoted by 'Ω'. This notation is used to represent the **lower bound** of algorithm's running time. Using omega notation we can denote shortest amount of time taken by algorithm.

Definition

A function f(n) is said to be in Ω (g(n)) if f(n) is bounded below by some positive constant multiple of g(n) such that

$f(n) \geq c * g(n)$ For all $n \geq n_0$

It is denoted as $f(n) \in \Omega(g(n))$. Following graph illustrates the curve for Ω notation.

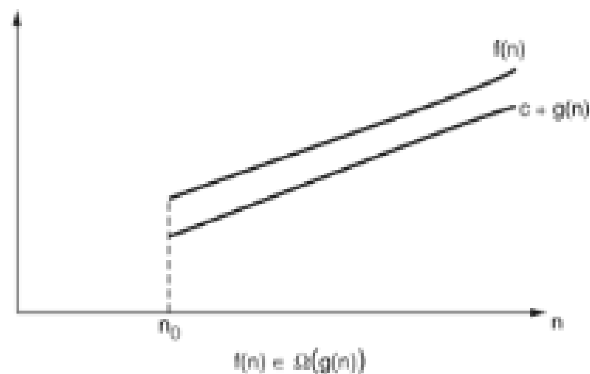


Fig. 2.4.2

Example :

Consider $f(n) = 2n^2 + 5$ and $g(n) = 7n$

Then if $n = 0$

$f(n) = 2(0)^2 + 5$

$= 5$