UNIT – I

ALGORITHM

Informal Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the i/p into the o/p.

Formal Definition:

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task.
All algorithms should satisfy the following criteria.

1. INPUT → Zero or more quantities are externally supplied.
2. OUTPUT → At least one quantity is produced.
3. DEFINITENESS → Each instruction is clear and unambiguous.
4. FINITENESS → If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. EFFECTIVENESS → Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Issues or study of Algorithm:

- How to device or design an algorithm → creating and algorithm.
- How to express an algorithm → definiteness.
- How to analysis an algorithm → time and space complexity.
- How to validate an algorithm → fitness.
- Testing the algorithm → checking for error.

The study of Algorithms includes many important and active areas of research.

There are four distinct areas of study one can identify

1**. How to device algorithms**-

Creating an algorithm is an art which many never fully automated. A major goal is to study various design techniques that have proven to be useful. By mastering

these design strategies, it will become easier for you to device new and useful algorithms. some of techniques may already be familiar, and some have been found to be useful. Dynamic programming is one technique. Some of the techniques are especially useful in fields other than computer science such as operations research and electrical engineering.

2. How to validate algorithms:

Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. We refer to this process as **algorithm validation**. The algorithm need not as yet be expressed as a program. The purpose of validation is to assure us that this algorithm will work correctly independently. Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is referred to as program proving or sometimes as **program verification**.

A proof of correctness requires that the solution be stated in two forms. One form is usually as a program which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in the predicate calculus. The second form is called a specification, and this may also be expressed in the predicate calculus. A complete proof of program correctness requires that each statement of a programming language be precisely defined and all basic operations be proved correct.

3. How to analyze algorithms:

As an algorithm is executed, it uses the computer's central processing unit (CPU) to perform operations and its memory to hold the program and data. Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage algorithms replace.we analyze the algorithm based on time and space complexity.The amount of time neede to run the

algorithm is called time complexity.The amount of memory neede to run the algorithm is called space complexity

4. How to test a program:

Testing a program consists of two phases

1. Debugging

2. Profiling

Debugging: It is the process of executing programs on sample data sets to determine whether faulty results occur and, if so to correct them. However, as E. Dijkstra has pointed out, “debugging can only point to the presence of errors, but not to the absence".

Profiling*:* Profiling or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

Algorithm Specification:

Algorithm can be described in three ways.

1. **Natural language like English:**

When this way is choused care should be taken, we should ensure that each & every statement is definite.

2. **Graphic representation called flowchart**:

This method will work well when the algorithm is small& simple.

3. **Pseudo-code Method:**

This method describe algorithms as program, which resembles language like Pascal & algol.

Pseudo-Code Conventions for expressing algorithms:

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces {and}.

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

4. Compound data types can be formed with records. Here is an example,

```
Node. Record
{
  data type – 1   data-1;
        .
        .
        .
  data type – n  data – n;
  node * link;
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

5. Assignment of values to variables is done using the assignment statement.

<Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.

→ Logical Operators AND, OR, NOT
→Relational Operators <, <=,>,>=, =, !=

7. The following looping statements are employed.

For, while and repeat-until

While Loop:

```
While < condition > do
{
      <statement-1>
            .
            .
```

.

<statement-n>
}

For Loop:

For variable: = value-1 to value-2 step step do

{
<statement-1>
.
.
.
<statement-n>
}

repeat-until:

repeat
<statement-1>
.
.
.
<statement-n>
until<condition>

8. A conditional statement has the following forms.

→ If <condition> then <statement>
→ If <condition> then <statement-1>
Else <statement-1>

Case statement:

Case
{
: <condition-1> **:** <statement-1>
.
.
.
: <condition-n> **:** <statement-n>

: else **:** <statement-n+1>
}

9. Input and output are done using the instructions read & write.

10. There is only one type of procedure:
Algorithm, the heading takes the form,

Algorithm Name (Parameter lists)

Examples:

→ algorithm for find max of two numbers

```
 algorithm Max(A,n)
// A is an array of size n
{
Result := A[1];
for I:= 2 to n do
   if A[I] > Result then
        Result :=A[I];
  return Result;
}
```

.

➔ Algorithm for Selection Sort:

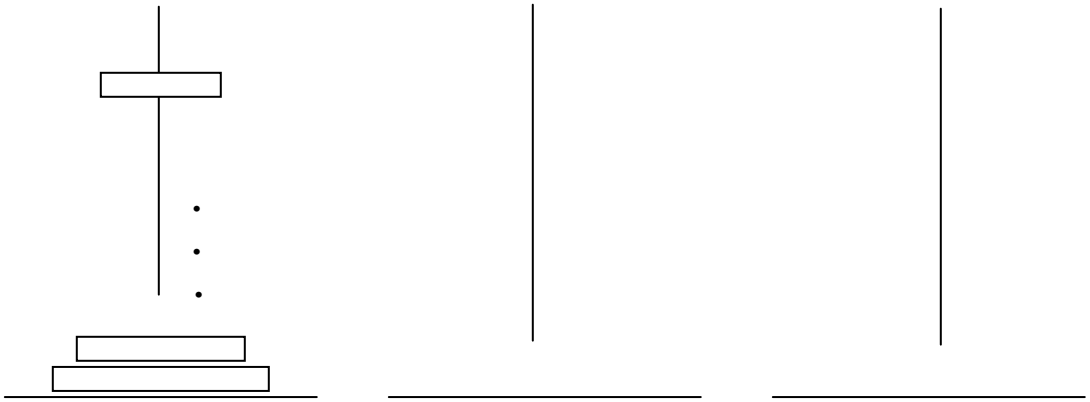
```
Algorithm selection sort (a,n)
      // Sort the array a[1:n] into non-decreasing order.
{
        for i:=1 to n do
        {
                j:=i;
                for k:=i+1 to n do
                        if (a[k]<a[j]) then j:=k;
                        t:=a[i];
                        a[i]:=a[j];
                        a[j]:=t;
        }
}
```

Recursive Algorithms:

- A Recursive function is a function that is defined in terms of itself.
- Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself is Direct Recursive.
- Algorithm 'A' is said to be Indirect Recursive if it calls another algorithm which in turns calls 'A'.
- The Recursive mechanism, are externally powerful, but even more importantly, many times they can express an otherwise complex process very clearly. Or these reasons we introduce recursion here.
- The following 2 examples show how to develop a recursive algorithms.

  → In the first, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.

1. Towers of Hanoi:



Towers of Hanoi is a problem in which there will be some disks which of decreasing sizes and were stacked on the tower in decreasing order of size bottom to top. Besides this there are two other towers (B and C) in which one tower will be act as destination tower and other act as intermediate tower. In this problem we have to move the disks from source tower to the destination tower. The conditions included during this problem are:

1) Only one disk should be moved at a time.
2) No larger disks should be kept on the smaller disks.

Consider an example to explain more about towers of Hanoi:

Consider there are three towers A, B, C and there will be three disks present in tower A. Consider C as destination tower and B as intermediate tower. The steps involved during moving the disks from A to B are

Step 1: Move the smaller disk which is present at the top of the tower A to C.
Step 2: Then move the next smallest disk present at the top of the tower A to B.
Step 3: Now move the smallest disk present at tower C to tower B
Step 4: Now move the largest disk present at tower A to tower C
Step 5: Move the disk smallest disk present at the top of the tower B to tower A.
Step 6: Move the disk present at tower B to tower C.
Step 7: Move the smallest disk present at tower A to tower C
In this way disks are moved from source tower to destination tower.

ALGORITHM FOR TOWERS OF HANOI:

```
Algorithm Towersofhanoi (n, X ,Y, Z)
        {
            if (n>=1) then
                {
                  Towersofhanoi(n-1, X, Z, Y);
                  Write("move top disk from tower ",X, "to top of tower",Y);
                 Towersofhanoi (n-1, Z, Y, X);
                }
        }
```

TIME COMPLEXITY OF TOWERS OF HANOI:

The recursive relation is:

$$t(n)=1; \quad \text{if } n=0$$
$$=2t(n-1)+2 \quad \text{if } n>=1$$

Solve the above recurrence relation then the time complexity of towers of Hanoi is O(2^n)

Performance Analysis:

1. Space Complexity:

The space complexity of an algorithm is the amount of memory it needs to run to compilation.

2. Time Complexity:

The time complexity of an algorithm is the amount of computer time it needs to run to compilation.

Space Complexity:

→ The Space needed by each of these algorithms is seen to be the sum of the following component.

1. A fixed part that is independent of the characteristics (eg:number,size)of the inputs and outputs.

The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

1. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space.

- The space requirement s(p) of any algorithm p may therefore be written as,

  S(P) = c+ Sp(Instance characteristics)

Where 'c' is a constant.

Example 1:

```
Algorithm abc(a,b,c)
{
return a+b++*c+(a+b-c)/(a+b) +4.0;
}
```

In this algorithm sp=0;let assume each variable occupies one word.
Then the space occupied by above algorithm is >=3.

S(P)>=3

Example 2:

```
Algorithm sum(a,n)
{
    s=0.0;
    for I=1 to n do
    s= s+a[I];
    return s;
}
```

In the above algoritm n,s and occupies one word each and array 'a' occupies n number of words so S(P)>=n+3

Example 3:

ALGORITHM FOR SUM OF NUMBERS USING RECURSION:

```
Algorithm RSum (a, n)
  {
     if(n<=0) then
        return 0.0;
     else
        return RSum(a,n-1)+a[n];
  }
```

The space complexity for above algorithm is:

In the above recursion algorithm the space need for the values of n, return address and pointer to array. The above recursive algorithm depth is (n+1). To each recursive call we require space for values of n, return address and pointer to array. So the total space occupied by the above algorithm is S(P) >= 3(n+1)

Time Complexity:

The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)

→The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics).

→ The number of steps any problem statemn t is assigned depends on the kind of statement.

For example, comments → 0 steps.
Assignment statements → 1 steps.
[Which does not involve any calls to other algorithms]

Interactive statement such as for, while & repeat-until→ Control part of the statement.

->We can determine the number of steps needed by a program to solve a particular problem instance in Two ways.

1. We introduce a variable, count into the program statement to increment count with initial value 0.Statement to increment count by the appropriate amount are introduced into the program.

   This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.

Example1:

Algorithm:

```
Algorithm sum(a,n)
{
      s= 0.0;
      count = count+1;
      for I=1 to n do
      {
       count =count+1;
      s=s+a[I];
      count=count+1;
      }
      count=count+1;
      count=count+1;
      return s;
      }
```

→ If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum execute a total of 2n+3 steps.

Example 2:

```
Algorithm RSum(a,n)
{
   count:=count+1;// For the if conditional
   if(n<=0)then
{
   count:=count+1; //For the return
   return 0.0;
}
else
{
   count:=count+1; //For the addition,function invocation and return
  return RSum(a,n-1)+a[n];
}
}
```

Example3:

ALGORITHM FOR MATRIX ADDITION

```
Algorithm Add(a,b,c,m,n)
{
 for i:=1 to m do
{
   count:=count+1; //For 'for i'
   for j:=1 to n do
  {
     count:=count+1; //For 'for j'
     c[i,j]=a[i,j]+b[i,j];
     count:=count+1; //For the assignment
}
count:=count+1; //For loop initialization and last time of 'for j'
```

}

count:=count+1; //For loop initialization and last time of 'for i'

If the count is zero to start with, then it will be 2mn+2m+1 on termination. So each invocation of sum execute a total of 2mn+2m+1 steps

2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

→First determine the number of steps per execution (s/e) of the statement and the
total number of times (ie., frequency) each statement is executed.
→By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Example 1:

Statement	*S/e*	*Frequency*	*Total*
1. Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3

step table for algorithm sum

Example 2:

		frequency		total steps	
Statements	s/e	n=0	n>0	n=0	n>0
1 algorithm Rsum(a,n)	0	_	_	0	

			0
2 {			
3 if(n<=0) then	1	1 1	1 1
4 return 0.0;	1	1 0	1 0
5 else return			
6 Rsum(a,n-1)+a[n];	1+x	0 1	0 1+x
7 }	0	_ _	0 0
Total			2 2+x

step table for algorithm recursive sum

Example 3:

Statements	s/e	frequency	total steps
1 Algorithm Add(a,b,c,m,n)	0	_	0
2 {	0	_	0
3 for i:=1 to m do	1	m+1	m+1
4 for j:=1 to n do	1	m(n+1)	mn+m
5 c[I,j]:=a[I,j]+b[I,j];	1	mn	mn
6 }	0	_	0
Total			2mn+2m+1

step table for matrix addition

Example 4:

Algorithm to find nth fibnocci number

Algorithm Fibonacci(n)

//Compute the nth Fibonacci number

```
{
   if(n<=1) then
     write (n);
  else
  {
      fnm2:=0;
      fnm1:=1;
     for i:=2 to n do
    {
        fn:=fnm1+fnm2;
```

```
    fnm:=fnm1;
   fnm1:=fn;
 }
 write(fn);
}
```

Asymptotic Notations:

The best algorithm can be measured by the efficiency of that algorithm.The efficiency of an algorithm is measured by computing time complexity.The asymptotic notations are used to find the time complexity of an algorithm.

Asymptotic notations gives fastest possible,slowest possible time and average time of the algorithm.

The basic asymptotic notations are Big-oh(O),Omega(Ω) and theta(Θ).

1:BIG-OH(O) NOTATION:

(i)It is denoted by 'O'.

(ii)It is used to find the upper bound time of an algorithm , that means the maximum time taken by the algorithm.

Definition : Let f(n),g(n) are two non-negative functions. If there exists two positive constants c ,n0 . such that c>0 and for all n>=n0 if f(n)<=c*g(n) then we say that f(n)=O(g(n))

THE GRAPH FOR BIG-OH (O) NOTATION:

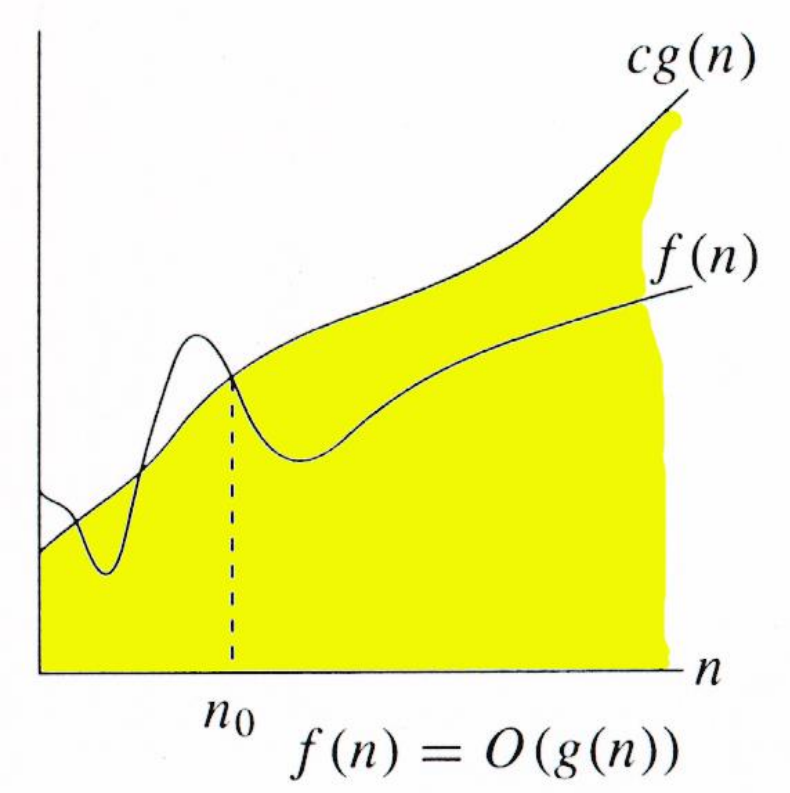


Figure 1

example : consider f(n)=2n+3 and g(n)=n^2
Sol : f(n)<=c*g(n)
let us assuming as c=1,
then f(n)<=g(n)
if n=1,
2n+3<=n^2 = 2(1)+3<=1^2 =>5<=1(false)
If n=2,
2n+3<=n^2=2(2)+3<=2^2= 7<=4(false)
if n=3,
2n+3<=n^2= 2(3)+3<=3^2=9<=9 (true)
if n=4,
2n+3<=n^2=>2(4)+3<=4^2=11<=6 (true)
if n=5,
2n+3<=n^2=2(5)+3<=5^2=13<=25 (true)
If n=6,2n+3<=n^2=2(6)+3<=6^2=15<=36 (true)
.:n>=3, f(n)=O(n^2) i.e, f(n)=O(g(n))

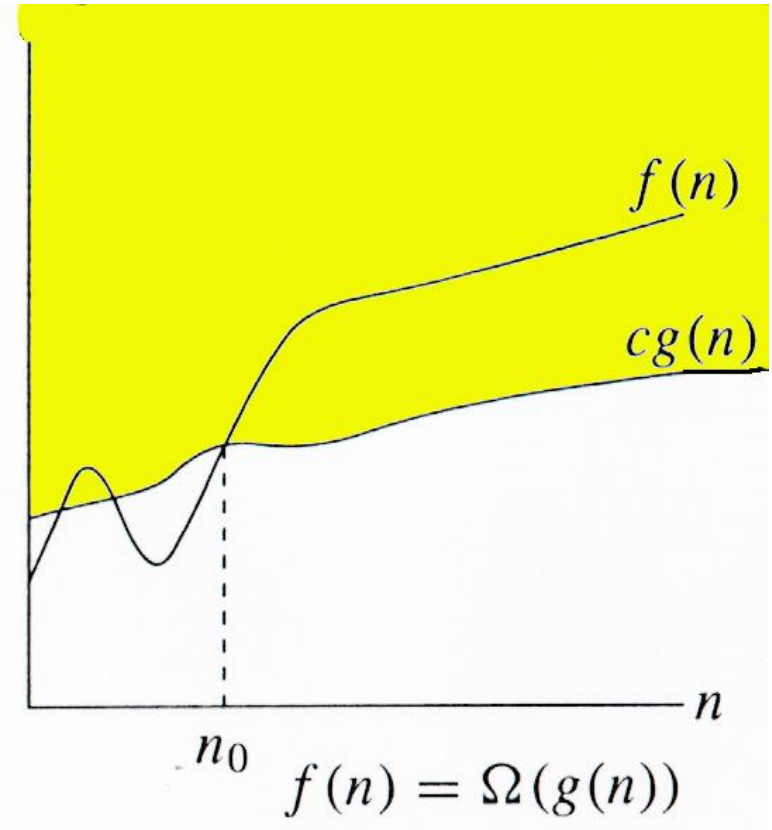
2:OMEGA (Ω) NOTATION:

(i)It is denoted by ' Ω'.

(ii)It is used to find the lower bound time of an algorithm, that means the minimum time taken by an algorithm.

Definition : Let f(n),g(n) are two non-negative functions. If there exists two positive constants c,n0.such that c>0 and for all n>=n0.if f(n)>=c*g(n) then we say that f(n)=Ω(g(n))

THE GRAPH FOR OMEGA NOTATION:



Example : consider f(n)=2n+5, g(n)=2n
Sol : Let us assume as c=1

If n=1:2n+5>=2n => 2(1)+5>=2(1) => 7>=2 (true)

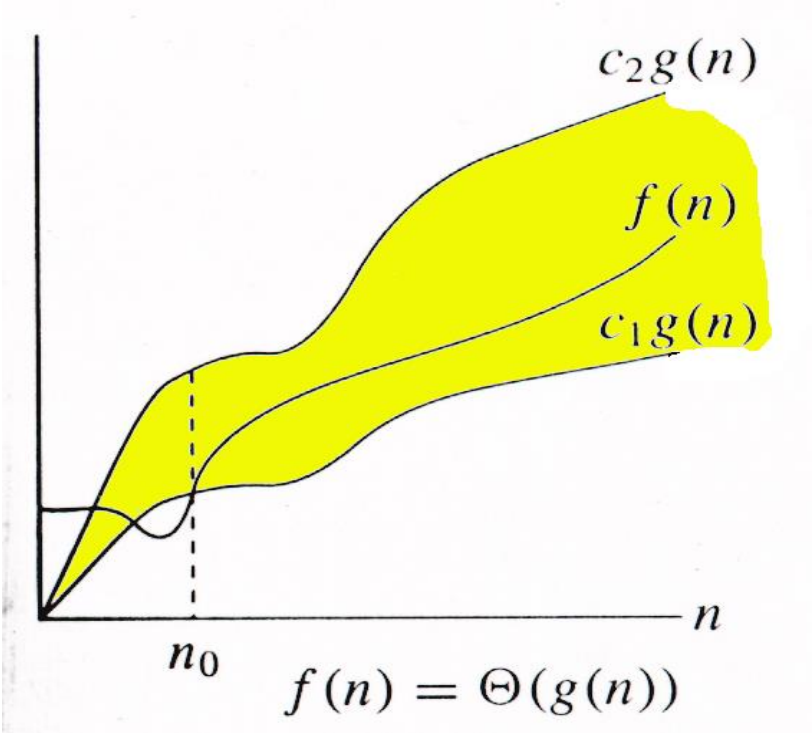
if n=2:2n+3>=2n=> 2(2)+5>=2(2)=> 9>=4 (true)
if n=3:2n+3>=2n=> 2(3)+5>=2(3)=> 11>=6 (true)
for all .:n>=1, f(n)=Ω(n) i.e , f(n)=Ω(g(n))

3:**THETA (Θ) NOTATION:**

(i)It is denoted by the symbol called as (Θ).

(ii)It is used to find the time in-between lower bound time and upper bound time of an algorithm.

Definition : Let f(n),g(n) are two non-negative functions. If there exists positive constants c1,c2,n0.such that c1>0,c2>0 and for all n>=n0.if c1*g(n)<=f(n)<=c2*g(n) then we say that f(n)=Θ(g(n))



Example : consider f(n)=2n+5, g(n)=n
Sol :c1*g(n)<=f(n)<=c2*g(n)
let us assuming as c1=3 then c1*g(n)=3n
if n=1,
3n<=2n+5=>3(1)<=2(1)+5=>3<=7 (true)
If n=2,
3n<=2n+5=>3(2)<=2(2)+5=>6<=9 (true)

If n=3,
3n<=2n+5=>3(3)<=2(3)+5=>9<=11 (true)

c2=4 c2*g(n)=4n
if n=1,
2n+5<=4n=>2(1)+5<=4(1)=>7<=4
If n=2,
2n+5<=4n=>2(2)+5<=4(2)=>9<=8
If n=3,
2n+5<=4n=>2(3)+5<=4(3)=>11<=12 (true)
If n=4,
2n+5<=4n=>2(4)+5<=4(4)=>13<=16 (true)
for all .:n>=3 f(n)=Θ(n) f(n)= Θ (g(n))

4:**LITTLE-OH (o) NOTATION:**

Definition : Let f(n),g(n) are two non-negative functions
if $\lim_{n\to\infty} [f(n) / g(n)] = 0$ then we say that f(n)=o(g(n))

example : consider f(n)=2n+3, g(n)=n^2
sol : let us
lim f(n)/g(n) = 0
n->$^{\infty}$
lim (2n+3) / (n^2)
n->$^{\infty}$
=lim n(2+(3/n)) / (n^2)
n->$^{\infty}$
=lim (2+(3/n)) /n
n->$^{\infty}$
=2/$^{\infty}$
=0
.:f(n)=o(n^2).

5:**LITTLE OMEGA NOTATION:**

Definition: Let f(n) and g(n) are two non-negative functions.
if lim g(n)/f(n) = 0 then we say that f(n)=ω(g(n))
n->$^{\infty}$
example : consider f(n)=n^2, g(n)=2n+5
sol : let us

lim g(n)/f(n) = 0
(n->∞)
=lim (2n+5) /(n^2)
(n->∞)
=lim n(2+(5/n)) / (n^2)
(n->∞)
=lim (2+(5/n)) / n =2/∞=0
(n->∞)
.:f(n)= ω(n).

Amortized analysis:

Amortized analysis means finding average running time per operation over a worst case sequence of operations.

Suppose a sequence I1,I2,D1,I3,I4,I5,I6,D2,I7 of insert and delete operations is performed on a set.

Assume that the actual cost of each of the seven inserts is one and for delete operations D1 and D2 have an actual cost of 8 and 10 so the total cost of sequence of operations is 25.

In amortized scheme we charge some of the actual cost of an operation to other operations. This reduce the charge cost of some operations and increases the cost of other operations. The amortized cost of an operation is the total cost charge to it.

The only requirement is that the some of the amortized complexities of all operations in any sequence of operations be greater than or equal to their some of actual complexities i.e.,

$\sum_{1 \le i \le n} amortized(i) \ge \sum_{1 \le i \le n} actual(i) \longrightarrow$ (1)

Where amortized(i) and actual(i) denote the amortized and actual complexities of the i^{th} operations in a sequence on n operations.

To define the potential function p(i) as:

p(i)=amortized(i)-actual(i)+p(i-1) ⟶ (2)

If we sum equation (2) for 1≤i≤n we get

$\sum_{1\le i\le n} p(i) = \sum_{1\le i\le n}(amortized(i) - actual(i) + p(i-1))$

$$\sum_{1\le i\le n} p(i) - \sum_{1\le i\le n} p(i-1) = \sum_{1\le i\le n} (amortized(i) - actual(i))$$

$P(n)-p(0) = \sum_{1\le i\le n}(amortized(i) - actual(i))$

From equation (1) we say that

P (n)-p (0) ≥0 ⟶ (3)

Under assumption p(0)=0,p(i) is the amount by which the first 'i' operations have been over charged (i.e., they have been charged more than the actual cost).

The methods to find amortized cost for operations are:

1. Aggregate method.
2. Accounting method.
3. Potential method.

1. Aggregate method:

The amortized cost of each operation is set equal to Upper Bound On Sum Of Actual Costs(n)/n.

2. Accounting method:

In this method we assign amortized cost to the operations (possibly by guessing what assignment will work),compute the p(i) using equation(2) and show that p(n)-p(0)>=0.

3.Potential method:

Here we start with potential function that satisfies equation(3) and compute amortized complexities using equation(2).

Example:
Let assume w**e** pay $50 for each month other than March, June, September**,** and December $100 for every June, September. calculate cost by using aggregate, accounting and potential method .

Month	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Actual cost:	**50**	**50**	**100**	**50**	**50**	**100**	**50**	**50**	**100**	**50**	**50**	**200**	**50**	**50**	**100**	**50**
Amortized cost:	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**
P( ):	**25**	**50**	**25**	**50**	**75**	**50**	**75**	**100**	**75**	**100**	**125**	**0**	**25**	**50**	**25**	**50**

Aggregate Method:

$=200 \times \lfloor n/12 \rfloor + 100(\lfloor n/3 \rfloor - \lfloor n/12 \rfloor) + 50(n- \lfloor n/3 \rfloor)$
$=100 \times \lfloor n/12 \rfloor +50 \lfloor n/3 \rfloor +50n$
$\leq 100 \times (n/12) + 50 \times (n/3) + 50 \times n$
$=50\ n\ ((1/6) + (1/3) + (1))$
$=50\ n\ ((1+2+6)/6)$
$=50\ n\ (9/6)$
$=75n.$

In the above problem the actual cost for 'n' months does not exceed 200n from the aggregate method the amortized cost for 'n' months does not exceed $75. The amortized cost for each month is set to $75.
Let assume p(0)=0 the potential for each and every month.

Accounting method:
From the above table we see that using any cost less than $75 will result in $p(n)-p(0) \leq 0$.
The amortized cost must be ≥ 75.
If the amortized cost ≤ 75 then only the condition p(n)-p(0)<=0.

Potential method:

To the given problem we start with the potential function as:

P (n) =0 n mod 12=0
P (n) =25 n mod 12=1 or 3

P (n) =50 n mod 12=4, 6, 2
P (n) =75 n mod 12=5, 7, 9
P (n) =100 n mod 12=8, 10
P (n) =125 n mod 12=4

From the above potential function the amortized cost for operation is evaluated for amortized(i)=p(i)-p(i-1)+actual(i).

Probabilistic analysis:

In probabilistic analysis we analyze the algorithm for finding efficiency of the algorithm.The efficiency of algorithm is also depend upon distribution of inputs.In this we analyze algorithm by the concept of probability.

For example the company wants to recruiting k persons from the n persons.To do this the company assigns ranking to all n persons depend upon their performance.The rankings of n persons from r_1 to r_n.To n persons we get n! permutations out of n! permutations the company selects any one combination that is from r_1 to r_k