Algorithm:-

Algorithm is a finite set of instructions to perform a specific task.

(or)

step by step procedure inorder to solve a computational problem.

characteristics (or) Features:-

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

1. Input:-

Every algorithm has 0 (or) more valid inputs.

2. Output:-

Every algorithm must produce atleast one output./more output.

3. Definiteness:-

Each instruction of an algorithm should be unAmbiguous and clear.

Ex:-

Add 5 to x
Add 6 to x.

4. Finiteness:-

Each algorithm must be terminated after finite no. of steps.

Ex:-

```
flag=0
while(1)
{
    if flag=1
    break;
}
```

5. Effectiveness:-

Algorithm only contains necessary statements.

It is designed to perform its assigned task.

Issues of an Algorithm:- [4 types]

(1) How to device (create) an algorithm.
(2) How to validate an algorithm
(3) How to Analyse an algorithm.
(4) How to test an program.

- → Divide & conquer
- → Greedy method
- → Dynamic programming
- → Back tracking
- → Branch & Bound etc.

- → Time & space complexity
- → Efficiency.

- → Debugging
- → profiling (or) performance Measurement

To calculate time & space complexity values.

Algorithm specification:

- The algorithm can be expressed in notation
- The algorithm can be represented in flowchat.
- Pseudocode convention similar to programming constructs.

10 basic rules for pseudocode convertion:

(1) comments begin with "//" continue until end of the line.

(2) Blocks are indicated with matching braces { & }.

(3) An indentifier begins with a letter.
primary (or) basic datatypes are not expli-

city specified but compound datatypes are expressed as records.

(Record structure)

```
node = record
{
    datatype-1 data-1;
        :
    datatype-n data-n;
    node* link
}
```

link → pointer variable [points to node type]

(4) Assignment of values to the variable is done as follows:

(structure of Assignment stmts)

<variable> := <expression>;

(←)

(5) There are Boolean values [(T) or (F)]. They are true & false.

- Logical operators - AND, OR, NOT
- Releational operators - $<, <=, >, >=, =, \neq$

(6) Elements of an array can be accessed by "[" and "]".

A[i] → i^{th} element in an Array A.

A[i,j] → j^{th} element in i^{th} row of an A.

(7) Looping statements are: While, for, and repeat-until.

(1)
```
While <condition>
{
    <statement 1>
        :
    <statement n>
}
```

(2)
```
for variable := value 1 to value n. step
step do
{
    <statement 1>
        :
    <statement n>
}
```

(3)
```
repeat
    <statement 1>
        :
    <statement n>
Until <condition>.
```

(8) conditional statements are expressed as follows

simple if:

if <condition> then <statement>

if - else:

if <condition> then <statement 1> else <statement 2>

case:

case
{
: <condition 1> : <statement 1>
:
: <condition n> : <statement n>
: else : <statement n+1>
}

(9) Input & output statements are written by using "read" & "write".

I/p & O/p
↓ ↓
read & write

(10) In pseudocode convention, it has single procedure and it is called "Algorithm".

- It has two parts:

i) Algorithm heading.

Algorithm Name (<parameter list>)

↓ Algorithm name

↓ list of parameters or variables to be used in the Alg.

ii) Algorithm body.

- It contains no. of stmts to contain more task.
- They are used to perform user task.

Ex:- sorting, searching.

Ex:-

Algorithm Max (A, n)
// A is an array of size n.

```
{
    Result := A[1];
    for i := 2 to n do
        if Result > A[i] then    Result := A[i];
        return Result;
}
```

Performance Analysis:

To analyze an algorithm, we require 2 factors.

1) space complexcity
2) Time complexcity

1) Space complexcity:

- It is the total amount of memory required to complete it's task.
- The algorithm is divided into 2 parts.
  - i) Fixed part
  - ii) Variable part

$$\underbrace{S(P)}_{\text{space complexcity of program}} = \underbrace{c}_{\text{fixed part}} + \underbrace{S_p}_{\text{variable part}}$$

Specifies i/p & o/p variables

- **Fixed part:** contains independent variables and ~~compon~~ constants.
  - → Independent variables are didn't depend on other variable.
  - → It contains static characterstics of the variable
    - can't change w.r.to time.
- **variable part:** It indicates the instance (or) dynamic characterstics of a variable.
  - → These characters changes w.r.t time.
  - → Dependent variables depends on other variable

Ex:- 1

{Independ variable}

```
Algorithm abc (x,y,z)
{
    return (x*y*z +(x-y));
}
```

$S(P) = c + S_p$

$= 3 + 0$

$\therefore S(P) \geq 3$

It is the min. space complexcity taken by an algorithm.

Ex:- 2

Algorithm (x, n)

{

total := 0;

Independent variable

for $i \leftarrow 1$ to n do

{

dependent

total := total + $x[i]$;

n times loop

}

}

$S(P) = C + S_p$

$= 3 + n$

$\therefore S(P) \geq 3 + n$.

Ex:- 3

Formal parameters

Algorithm Rsum (x, n)

{

if $(n \leq 0)$

return 0;

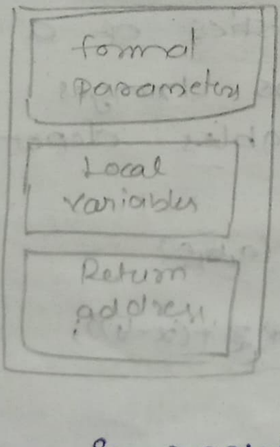
else

return (Rsum $(x, n-1)$ + $x[n]$);

(return address)

}

Recursive procedure {stack}



- Return address is maintained in the memory location containing $x[n]$ value.

$S(P) = C + S_p$

$S(P) = 0 + 3(n+1)$

$\therefore S(P) \geq 3(n+1)$

Time complexcity:

Total amount of time taken by an algorithm to complete its task.

To compute the time complexcity we use two methods.

1. count Method
2. Frequency Method.

1) count method:

- We have to take a global variable "count".
- It is initially initialized as zero.
- For each valid and executable statements we need to increment the count value.
- For non-executable stmts not increment the count value.

Ex:-1

```
Algorithm sum(x,n)
{
    total:=0;
    for i <- 1 to n do
    {
        total:= total + x[i];
    }
}
```

```
count:=0
Algorithm sum(x,n)
{
    total:=0;
    count <- count+1
    for i <- to n do
    {
        count <- count+1
        total:= total + x[i];
        count <- count+1
    };
    count <- count+1
}
```

no. of loops = O(n)

2n times

2 times count

$\therefore T(P) = 2n+2 / 2(n+1)$

$T(P) = O(n)$

Ex:-2

```
Algorithm add (a,b)
{
    count <- count+1
    for i <- 1 to n do
```

{

for j ← 1 to n do

{

c[i,j] := a[i,j] + b[i,j]; $2n^2$

}

}

}

$T.P = 2n^2 + 2n + 1$

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

2) Frequency method:

- In this time complexity is calculated by constructing a table.
- The table consists of 2 parts.
  1) steps per execution (s/e) → No. of steps that are required for an execution
  2) Frequency → No. of times a given stmt can be executed

Ex:- 1

```
Algorithm Sum(x,n) → non executable
{  → non executable
    total := 0;
    for i ← 1 to n do
    {
        total := total + x[i];
    }
}
```

s/e	frequency	Total
-	-	-
-	-	-
1	1	1
1	n+1	n+1
-	-	-
1	n	n
-	-	-
-	-	-

$T(P) = 2n+2$

$T(P) = O(n)$

(2) Algorithm reverse(n)

```
{
   rev: =0;
   While (n>0)
   {
      r:=n% 10
      rev := rev * 10+r;
      n:=n/10;
   }
}
```

s/e	frequency	Total
-	-	-
-	-	-
1	1	1
1	n+1	n+1
-	-	-
1	n	n
1	n	n
1	n	n
-	-	-
-	-	-
		4n+2

T(P) = 4n+2

T(P) = O(n)

(3)

Algorithm Rsum (x,n)

```
{
   if (n≤0) then
   return 0;
   else
   return((x,n-1),x[n]);
}
```

0 to n-1

n (x)

s/e	frequency		Total	
	n=0	n>0	n=0	n>0
-	-	-	-	-
-	-	-	-	-
1	1	1	1	1
1	1	0	1	0
-	-	-	-	-
1+x	0	1	0	1+x
-	-	-	-	-
			2	2+x

If n=0

then T(P) = 2

∴ T(P) = O(n)

else

T(P) = 2+x

T(P) = O(x)

(4)

Algorithm add (a,b)

```
{
   for i ← 1 to n do              →(n+1)
   {
      for j ← 1 to n do           n*(n+1)
      {
          c[i,j] = a[i,j]+b[i,j];     n*n
      }
   }
}
```

$T(P) = n+1+n^2+n+n^2$

$T(P) = 2n^2+2n+1$

$= O(n^2)$

$\therefore S(P) = C + S_p$

$\because C = i+j+n$ | $S_p = c[i,j] + a[i,j] + b[i,j]$
$= 1+1+1$ | $= n \times n + n \times n + n \times n$
$C = 3$ | $= n^2+n^2+n^2$
 | $= 3n^2$

$\therefore S(P) = C + S_p$

$= 3+3n^2$

$S(P) = O(n^2)$

Matrix Multiplication:

```
Algorithm mul(a,b) - non
{ - non
  for i <- 1 to n do - exe                        n+1
  { - non
    for j <- 1 to n do - exe                       n*(n+1)
    { - non
      c[i,j] := 0; - exe                           n*n
      for k <- 1 to n do - exe                     n^2*(n+1)
      { - non
        c[i,j] := c[i,j] + (a[i,k] * b[k,j]); - exe
                                                   n*n*n
      } - non
    } - non
  } - non
} - non
```

$T(P) = n+1 + \underbrace{n*(n+1)}_{n^2+n} + \underbrace{n^2*(n+1)}_{n^3+n^2} + \underbrace{n*n*n}_{n^3}$

$= 2n^3+3n^2+2n+1$

$T(P) = O(n^3)$

$S(P) = C + S_P$

$C = i + j + k + n$
$= 1 + 1 + 1 + 1$
$C = 4$

$S_P = c[i, j] + a[i, k], b[k, j]$
$= n \times n + n \times n + n \times n$
$= n^2 + n^2 + n^2$
$= 3n^2$

$\therefore S(P) = C + S_P$
$= 4 + 3n^2$

$S(P) = O(n^2)$

Asymptotic Notation:

- short hand ~~time~~ form to represent time complexcity. There are 5 notations.

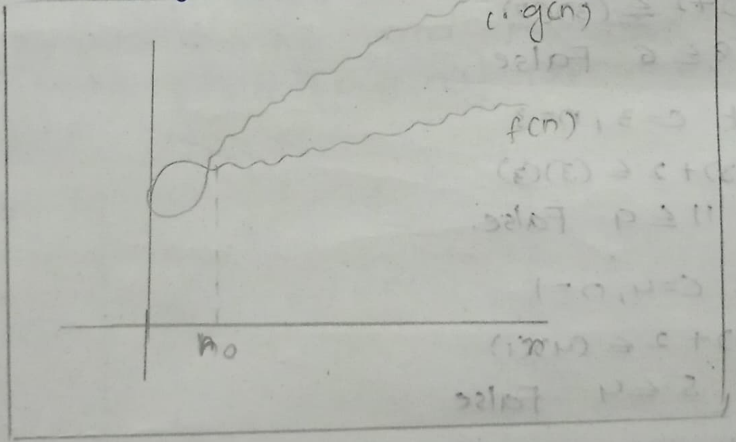
1) **Big-oh (O) notation:**

"O" symbol. It represents the worst case time complexcity. Maximum amount of time to run the given algorithm. It indicates the ~~lower~~ upper bound of time complexcity.

Defination:

→ The function $f(n) = O(g(n))$. if and only if there exists [$\exists$] constant c. and positive integers n, n_0, such that $\boxed{f(n) \leq c.\, g(n)}$ for all ($\forall$) $n \geq n_0$, $n_0 \geq 1$ and $c > 0$.

→ Here $f(n)$ & $g(n)$ are non-negative functions.



Ex:- 1

$f(n) = 3n + 2$ and $g(n) = n$. Then prove that $f(n) = O(g(n))$

Sol:

$f(n) \leq c.g(n)$

$3n + 2 \leq c.n$

Select $c = 1$, $n = 1$

$3(1) + 2 \leq (1)(1)$

$5 \leq 1$ false.

Select $c=1, n=2$

$3(2)+2 \leq (1)(2)$

$8 \leq 2$ False.

Select $c=1, n=3$

$3(3)+2 \leq (1)(3)$

$11 \leq 3$ False.

Select $c=2, n=1$

$3(1)+2 \leq (2)(1)$

$5 \leq 3$ False

Select $c=2, n=2$

$3(2)+2 \leq (2)(2)$

$8 \leq 4$ False.

Select $c=2, n=3$

$3(3)+2 \leq (2)(3)$

$11 \leq 6$ False.

Select $c=3, n=1$

$3(1)+2 \leq (3)(1)$

$5 \leq 3$ False

Select $c=3, n=2$

$3(2)+2 \leq (3)(2)$

$8 \leq 6$ False

Select $c=3, n=3$

$3(3)+2 \leq (3)(3)$

$11 \leq 9$ False.

Select $c=4, n=1$

$3(1)+2 \leq (4)(1)$

$5 \leq 4$ False

Select $c=4, n=2$

$3(2)+2 \leq (4)(2)$

$8 \leq 8$ True

for $c=4$ and $n \geq 2$

we can say that $f(n) \leq c.g(n)$

$\therefore f(n) = O(g(n))$

Ex:- 2

$f(n) = 2n+2$ and $g(n) = n^2$. Then prove that $f(n) = O(g(n))$

Sol: $f(n) \le c \cdot g(n)$

$2n+2$

select $c=1$, $n=1$

$2(1)+2 \le (1)(1)^2$

$4 \le 1$ False

select $c=1$, $n=2$

$2(2)+2 \le (1)(2)^2$

$6 \le 4$ False

select $c=1$, $n=3$

$2(3)+2 \le (1)(3)^2$

$8 \le 9$ True

For $c=1$, & $n \ge 3$.

we say than $f(n) \le c \cdot g(n)$

$\therefore f(n) = O(g(n))$

2) omega (Ω) Notation:

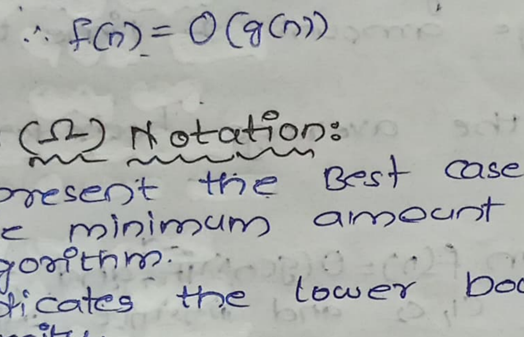
→ It represent the Best case time complexcity.

→ It's the minimum amount of time to run an algorithm.

→ It indicates the lower boundary of time complexcity.

Defination:

The function $f(n) = \Omega(g(n))$ if and only if there exists [∃] constant c and positive integers n, n_0 such that $\boxed{f(n) \ge c \cdot g(n)}$ for all (∀) $n \ge n_0$, $n_0 \ge 1$ & $c > 0$.



Ex:- 1 $f(n) = 2n+2$ and $g(n) = n$. then prove that $f(n) = \Omega(g(n))$.

Sol: $f(n) \ge c \cdot g(n)$

$2n+2 \ge c \cdot n$

Select $c=1$, $n=1$

$2(1)+2 \ge (1)(1)$

$4 \ge 2$ True

For $c=1$, $n \ge 1$.

We say that $f(n) \ge c\,(g(n))$

$\therefore f(n) = \Omega(g(n))$.

Ex:- 2 $f(n) = 3n+2$, $g(n) = n$, than prove that $f(n) = \Omega(g(n))$

Sol:

$f(n) \geq c \cdot g(n)$

$3n+2 \geq c \cdot n$

Select $c=1$, $n=1$

$3(1)+2 \geq (1)(1)$

$5 \geq 2$ True.

for $c=1$, $n \geq 1$

we say that $f(n) \geq c g(n)$.

$\therefore f(n) = \Omega(g(n))$

3) Theta (Θ) Notation:

→ It represents average time complexcity.

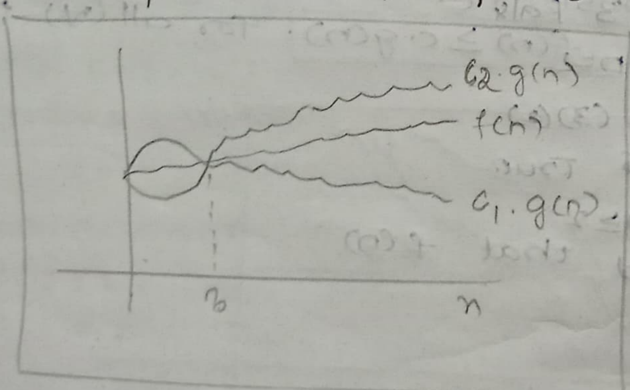
→ It's the average amount of time to run the algorithm.

→ It indicates the average boundary.

Defination:

The function $f(n) = \Theta(g(n))$ if and only if there exists [$\exists$] constants c_1, c_2 and positive integers n_1, n_0 such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all ($\forall$) $n \geq n_0$, $n \geq 1$, $c_1 > 0$ and $c_2 > 0$.

Graphical representation of 'Θ' notation:



Ex: 1 $f(n) = 2n+2$ and $g(n) = n$. Then prove that $f(n) = \Theta(g(n))$

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

$c_1(n) \leq 2n+2 \leq c_2 \cdot (n)$

$c_1 = 1$, $c_2 = 1$, $n = 1$

$(1)(1) \leq 2(1)+2 \leq (1)(1)$

$1 \leq 4 \leq 1$ False

$c_1 = 1,\ c_1 = 2,\ n = 2$

$(1)(2) \le 2(2)+2 \le (1)(2)$

$2 \le 6 \le 2$ False

$c_1 = 2,\ c_2 = 2,\ n = 1$

$2(1) \le 2(1) + 2 \le 2(1)$

$2 \le 4 \le 2$ False

$c_1 = 3,\ c_2 = 3,\ n = 1$

$(1)(3) \le 2(1)+2 \le (1)(3)$

$3 \le 4 \le 3$ False.

$c_1 = 4 \quad c_2 = 4,\ n = 1$

$(4)(1) \le 2(1)+2 \le (4)(1)$

$4 \le 4 \le 4$ True.

$\therefore$ For $c_1 = 4,\ c_2 = 4$ & $n = 1$.

We can say that $c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$

$\therefore\ f(n) = \Theta(g(n))$

4) Little-oh (o) notation:

The function $f(n) = o(g(n))$ if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Ex:- $f(n) = 3n+2$ and $g(n) = n^2$. then prove that $f(n) = o(g(n))$

$$\lim_{n\to\infty} \frac{3n+2}{n^2} \Rightarrow \lim_{n\to\infty}\left(\frac{3}{n} + \frac{2}{n^2}\right) \Rightarrow \frac{3}{\infty} + \frac{2}{\infty} = 0+0 = 0$$

$\therefore\ f(n) = o(g(n))$

5) Little-omega (ω) notation:

The function $f(n) = \omega(g(n))$ if and only if

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0.$$

Ex:- $f(n) = 8n^2$ and $g(n) = 3n+2$ then prove that $f(n) = \omega(g(n))$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \Rightarrow \lim_{n\to\infty} \frac{3n+2}{n^2} \Rightarrow \lim_{n\to\infty} \left(\frac{3}{n} + \frac{2}{n^2}\right)$$

$$= \frac{3}{\infty} + \frac{2}{\infty} = 0+0 = 0$$

$\therefore\ f(n) = \omega(g(n))$

Amortized analysis (or) complexcity:

→ It is a method for analysing sequence of operations.

→ some are expensive (more time) & some are inexpensive (less time).

→ It is used to adjust the time (or) cost of expensive operations to inexpensive operation so that avg cost for each operation should be small.

→ It has 3 Methods:

1) Aggregate Method.
2) Accounting Method.
3) Potential Method.

1) Aggregate Method:

* In this Method, we determine the upperbound $T(n)$. on the total cost of sequence of operations.

* Then the amortized cost per each operation is $\frac{T(n)}{n}$

Ex:- 5 10 15

upper bound = 15

Total no. of operations = 3

$\therefore \frac{T(n)}{n} = \frac{15}{3} = 5$

$\therefore$ Amortized cost = 5

2) Acounting Method:

→ It is a one Aggregate analysis in which we can assign the amortized cost to each operation in the sequence.

→ We have to find the earlier operations whose amortized cost is greater than their actual cost.

→ The difference b/w amortized cost & actual cost calculated for every operation in sequence.

→ The difference b/w amortized cost & actual cost can be used as saved credit. This saved credit is used for remaining operations, whose amortized cost is less than actual cost

3) Potential Method:

→ It is one form of accounting Analysis, in which saved credit is computed as the potential function of the data structure.

→ potential functions in stack: push, pop.

→ potential functions in Queue: enqueue, dequeue

→ for every potential we need to calculate the amortized & actual cost.

Sequence of operations:

$I_1, I_2, D_1, I_3, I_4, I_5, I_6, D_2, I_7$

1 1 8 1 1 1 1 10 1

$7+8+10 = 25$

I = Insertion operations

D = Delection operations

I_1	I_2	D_1	I_3	I_4	I_5	I_6	D_2	I_7
1+1	1+1	6	1+1	1+1	1+1	1+1	6	1
2	2		2	2	2	2		

$2+2+6+2+2+2+2+6+1 = 25$

The sum of amortized complexities of all operations be ≥ the sum of their actual complexities.

$\sum_{1\le i\le n} \text{amortized}(i) \ge \sum_{1\le i\le n} \text{actual}(i)$; We define $p(i)$

$\underbrace{p(i)}_{\text{cost per operation } i} = \text{amortized}(i) - \text{actual}(i) + \underbrace{p(i-1)}_{\text{cost of } (i-1) \text{ operation}}$

Taking 'Σ' on B.S

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

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

$\therefore p(n) - p(0) \ge 0.$

Performance Measurement:-

Determine space & time complexities by running the program on computer.

⇒ The space & time requirements depend on the computer & machine.

⇒ Compare performance of an algorithm with another by computing the worst case time complexity of an algorithm

Ex:-

```
Algorithm Seqsearch(a,x,n)
// Search for x in a[1:n]. a[0] is used as an
{
    i=n;
    a[0]=x;
    while(a[i] ≠ x) do; i:=i-1;
    return i;
}
```

$T(P) = 1+1+n+1+n+1$

$= 2n+4$

$= O(n)$

linear time complexity.

(1) This analysis doesnot follow the asymptotic curve for small n values.

Find out what n values runtimes follows the assymptotic curve.

(2) Some times, runtimes may not lie on the curve due to elimination of low-order terms in the calculation of time complexity.

$\boxed{c_1 n + c_2 . \log n + c_3 (1)}$

$c_1.n$ for constant $c_1 > 0$

→ The assymptotic behaviour starts with o where $n < 100$.

→ for $n > 100$ → $n = 100, 200, 300, 400 \ldots 1000$.

→ $a(i) = i$, $1 \leq n$,

Modified algorithm for seq. search:

```
Algorithm Time search()
{
  for j: =1 to 1000 do a[j]: = j:
  for j: =1 to 10 do
  {
      n[j]: = 10 * (j-1);
      n[j+10]: = 100 * j;
  }
  for j:1 =1to 20 do
  {
      h: = Get Time();
      K: = Seq search (a, 0, n[j]);
      h1: = Get time ();
      t: =h1-h;
      write (n [j], t);
  }
}
```

Get Time () method.

return the current time in Milliseconds

Output of timesearch Algorithm:

n	time	n	time
0	0	100	0
10	0	200	0
20	0	300	1
30	0	400	0
40	0	500	1
50	0	600	0
60	0	700	0
70	0	800	1
80	0	900	0
90	0	1000	0

```
h:= Gettime()
t:=0;
While(t < DESIRED.TIME) do
{
    k:= seasearch(a,0,n[j]);
    h1 = GetTime();
    t:= h1-h;
}
```

Get the Test data:

→ We need large amount of i/p data to compute worst case & average case time complexity

Randomized algorithms:

→ use randomizer(s) random number generator.
→ It depends on o/p of the randomizer.
→ Valid randomized o/p is varied from run-run
→ randomized algorithm o/p is also varied from one run to another run based on the i/p.

→ Randomized algorithms are classified into 2 types:

(1) Las vegas Algorithm.
(2) Monte carlo Algorithm.

Los vegas Algorithm:

→ These algorithms produce some o/p for same i/p.
→ The ex time of this algorithm based randomized ex time can be characterized as a randomized.

Ex:- Randomized quick sort.

Monte Carlo Algorithm:

→ These algorithm produce different o/p's from run-to-run.
→ They produce most probably convert o/p

Ex:- Randomized probability Testing

Advantages:

1) simplicity
2) very efficient
3) Better computational complexcity.

Disadvantages:-

1) Quality
2) Releability is an issue
3) H/w failure

Applications:

1. primality testing.
2. Identifying the repeated elements.

Fermat's Theorem:

If p is prime number and $0<A<p$ then $(A^{p-1}-1)\%p=0$. Here 'A' is a composite number.

Ex:

$A=2$, $p=7$

$(2^{7-1}-1)\%7$

$(2^{6}-1)\%7$

$63\%7=0$

$\therefore$ 7 is prime.

```
Algorithm primalityTest (n,K)
{
    is prime:=True;
    for i:1 to K do
    {
        A:= random Int (2,n-1);
        if (A^(n-1)-1%n≠0) then
        is prime:= False;
    }
    return ~~ispp~~ isprime;
}
```

2. Identifying the repeated elements

10	20	30	40	50	60	60	60	60	60
1	2	3	4	5	6	7	8	9	10

Algorithm Repeated Element (a,n)

// Find the repeated element from an array a[1,n]

```
While (true) do
{
   i: = Random ()mod n+1;
   j: = Random ()mod n+1;
//i and j are random numbers in range [1,n]
   if (i≠j) and a[i] = a[j] then
   return i;
 }
}
```

- if $i \neq j$ and $a[i] = a[j]$ then $a[i]$ and $a[j]$ are duplicates.