UNIT-I : DAA

An algorithm is a step by step procedure inorder to solve a computational problem

(or)

It is a finite set of instructions to perform a particular task.

The algorithm has following characteristics

1. Input :- Algorithm has zero or more valid inputs.
2. output :- It generates atleast one output
3. Definiteness :- Each instruction should be clear or unambiguous (Single meaning)
4. Finiteness :- The algorithm should be ended after some finite number of steps.
5. Effectiveness :- It contains only necessary statements. The algorithm should perform its intended task.

The algorithm has following issues.

① How to devise algorithm:- The process of creating an algorithm is an art. It is not possible to fully automate this task.

There are number of methods to create the algorithm. Some of them are divide and conquer, greedy approach, dynamic programming, backtracking, branch and bound technique etc.

We have to choose right choice based on the problem to be solved.

② How to validate algorithm:- The algorithm must produce correct output for all valid inputs. This process is called an algorithm validation. Validation must be done before the program development.

③ How to analyze algorithms:- When an algorithm is executed, it uses the CPU's time and memory storage space.

In an analysis of the algorithms, users must compute the space complexity and time complexities.

The algorithm which uses less memory and less time is called as an efficient algorithms.

(A) How to test a Program:- The process of testing consists of two steps. They are debugging and profiling.

Debugging is the process of executing programs on sample data sets to determine whether faulty result occur or not.

If faulty results are occured, then we will correct them.

Profiling is also known as performace measurement. It is the process of executing a correct program on data sets and measuring the time and space it takes to compite the results.

Algorithm Specification

Algorithms are specified by natural language like english.

These are also expressed by some graphical representa called flowcharts

Flowchart is a pictorial (diagrammatical) represent of an algorithm.

In general, algorithms are specified by the Pseudocode conventions.

Pseudocode conventions are formed by using programming constructs.

These conventions are similar to the program language.

Pseudocode Convention has following rules.

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

(2) Blocks are indicated with matching braces { and }. ex:- compound statement, procedures

(3) An identifier begins with a letter.
The data types of variables are not explicitely specified.
Compound data types can be formed as follows.
ex:-
```
node = record
{
    datatype-1 data-1;
        :
    datatype-n data-n;
    node * link;
}
```
Here link is a pointer the record of type node.

(4) Assignment of values to variables can be done as follows.
<variable> := <expression>;

(5) There are two boolean values true and false. Logical operators are AND, OR and NOT. Relational operators are <, <=, >, >=, =, and ≠.

(6) Elements of array are accessed by their ~~ind~~. [and].
A[i] → ith element of an array A.
A[i][j] → jth element in the ith row of an array A.

⑦ Looping statements are while, for and repeat until loops

while <condition> do

{ <statement 1>
:
<statement n>
}

for variable := value 1 to value 2 step step do.

{ <statement 1>
:
<statement n>
}

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 2>

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

(9) Input and output statements are done using the instruction read and write.

(10) There is only one procedure called an algorithm.

An algorithm consists of two parts.

(i) Algorithm heading

(ii) Algorithm body

(i) Algorithm heading :- It consists of following structure.

Algorithm Name (<Parameterlist>)

Here Name indicates Procedure name.

Parameterlist indicates list of parameters to be used in the algorithm.

(ii) Algorithm body :- It has one or more statements that are placed with in braces { and }.

Ex:- Algorithm to compute the largest element in the array of elements (n).

```
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;
```

Performance Analysis

To analyze the performance of an algorithm we require two factors

1) Space complexity
2) Time complexity

1) Space complexity : It specifies the total amount of memory required by an algorithm to complete its task.

The algorithm is divided into

i) fixed part
ii) Variable part

i) fixed part :- It contains independent variables or constants. ie., variables that have independent characteristics. These variables are not depend on another variables.

Ex :- constants, inputs, outputs

ii) Variable part :- It contains variables that depend on other variables.

The instance characteristics of variable is called variable part.

Ex :- Algorithm abc (x, y, z)

return x * y * z + (x - y);

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

C : fixed part, S_p : Variable part

x, y, and z are fixed variables because these are not depended on other

It doesn't contain variable part

Suppose each variable requires 1 unit of memory.

Total it contains 3 units of memory.

So $S(P) \geq C + Sp \Rightarrow 3 + 0 \therefore S(P) \geq 3$

Ex:-2

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

$S(P) = C + Sp$

x, n, total required 3 units of memory so $C = 3$

The number of elements that are stored in an array depend on the variable n value. This requires n units of memory.

So $Sp \geq n$.

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

Ex3:-

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

Recursive algorithm use stack
Each stack stores formal parameters,
Local variables
return address.
To store these details we need 3units of memory.

Each call of Rsum requires following memory requirement.

x requires 1Unit of ~~time~~ memory.
n requires 1Unit of memory
x represents base address.
x[n] represents value.
x[n] requires 1Unit of memory

Here each call requires 3Units of memory
This Rsum function calls itself n number of times.
It doesn't contain any fixed part.
The function Rsum calls itself until condition fails.
For each Function call it also requires if statement. So totally requires 3(n+1) Units of memory

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

$$C = 0$$

$$Sp = 3(n+1)$$

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

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

2) Time Complexity:- It is the time required by an algorithm to complete its task.

Time complexity is calculated by using step count method.

Step count is determined by two methods i) Count method ii) Frequency

i) Count method:-

In this method, initialize a variable Count with 0 globally

For each valid step of an algorithm we need to increment the count by 1

valid step if the step that is needed by execution.

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

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

```
    count ← count + 1;
}
```

```
Algorithm Sum(a, n)
{
    count ← count + 2    ⟶ 2 times
    for i ← 1 to n do
    {
        Count ← count + 2    → 2n times.
    }
```

So, Total time complexity = $2n + 2$

Ex:-2

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

↳

```
Algorithm Add(a, b)
{
    for i ← 1 to n do
    {   count ← count + 1
        for j ← 1 to n do
        {   count ← count + 1
            c[i,j] := a[i,j] + b[i,j]
```

Ex:- Algorithm RSum(x,n)

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

S/e	Freq n=0	Freq n>0	t n=0	t n>0
—	—	—	—	—
—	—	—	—	—
1	1	1	1	1
1	1	0	1	0
—	—	0	—	—
—	—	—	—	—
1+x	0	1	0	1+x
—	—	—	—	—
			2	2+x

Total :- 2or2+x

Total x = tRSum(n-1)

It is the time required for RSum with

```
Algorithm mul ( a, b)
{
   for i ← 1 to n do                          — n+1
    { for j ← 1 to n do                       — n x (n+1)
      {                                       — n x n
        C[i][j] = 0;
        for K ← 1 to n do                     — n x n x (n+1)
        { C[i][j] += a[i][k] * b[k][j];        ← n x n x n
        }
      }
    }
}
```

$T(P) = n+1 + n^2 + n + n^2 + n^3 + n^2 + n^3$

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

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

To compute the space complexity of this algorithm, we have to list out all variables.

~~Present in~~

$a \rightarrow n^2$
$b \rightarrow n^2$
$c \rightarrow n^2$
$n \rightarrow 1$
$i \rightarrow 1$
$j \rightarrow 1$
$k \rightarrow 1$

$S(P) = C + Sp.$

$C = 4$

$Sp = 3n^2$

$S(P) = 4 + 3n^2$

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

$= 3n^2 + 4$

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

```
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
    }
}
```

S/e	Freq	Total
-	-	-
-	-	-
1	n+1	n+1
-	-	-
1	n^2+n	n^2+n
-	-	-
1	n^2	n^2
-	-	-
-	-	-
-	-	-

So time complexity $T(P) = 2n^2+2n+1$
$= O(n^2)$

TO compute the space complexity of this algorithm, we write the list of variables.

i ← 1 unit of memory
j ← 1 unit of memory
n ← 1 unit of memory

Here a, b, c array stores n rows and n columns data. For this each array requires n^2 units of memory. So a,b,c all $3n^2$ units of memory.

$S(P) = C + S_P$
C = 3 units of memory
$S_P = 3n^2$ units of memory
$S(P) = 3 + 3n^2 \Rightarrow O(n^2)$.

Range of Time complexities

$1 < \log n < \sqrt{n} < (n) < n\log n < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n$

Lower bound — Average bound — UPPer bound.

Time complexity can be classified in five types. They are as follows.

1. Constant
2. Linear
3. Quadratic
4. Cubic
5. Logarithmic

(1) Constant :- The statement will be executed by the compiler only once.

ex:- statement;

$\therefore T(P) = O(1)$

(2) Linear :- The statement will be executed by the compiler n number of times.

ex:- for i:=1 to n do
        statement;

$\therefore T(P) = O(n)$

(3) Quadratic :- The statement will be executed by the compiler n×n (n^2) number of times.

ex:- for i:=1 to n do
        for j:=1 to n do

(4) Cubic :- The statement will be executed by the compiler $n \times n \times n$ (n^3) times. This type of time complexity is known as cubic time complexity.

ex:-
```
for i =:1 to n do i = i+1 ;
for j =:1 to n do j = j+1 ;
for k =:1 to n do k = k+1 ;
statement ;
```
$\therefore T(P) = O(n^3)$

(5) Logarithmic :- In each and every step, the work area is divided into half of its original area. In binary search, for every time the search space is reduced to half.

ex:-
```
while ( low ≤ high)
{ mid := (low + high) / 2 ;
  if (key < a[mid])
  high = mid - 1 ;
else if (key > a[mid])
  low = mid + 1 ;
else
  return mid ;
}
```
$\therefore T(P) = O(\log n)$

The time complexity can be represented in 3 ways : i) Best case ii) Average case iii) Worst case

(i) Best case (Ω) :- It is the minimum amount time to complete an algorithm for a set of specic inputs.

ex:- In the linear search, we have to find out first element in the given list.

ii) Average case (Θ) :- It is the average amount of time to complete an algorithm

ex:- In the given linear search, we have to find out the middle element in the given list.

iii) Worst case (O):- It is the maximum amount of time to complete an algorithm for a set of specific inputs.

ex:- In the given linear search, we have to find out the last element in the given list.

Asymptotic Notations

The complexities are represented by a shorthand notation called asymptotic notion.

① Bigho

① Big oh (O) :-

consider two functions $f(n)$ & $g(n)$ and two integers n_0, c such that

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

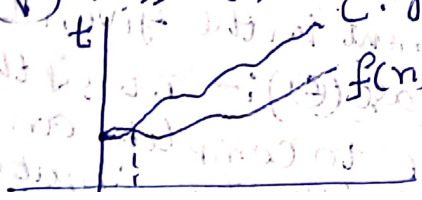
Time complexity may be Best, worst, Average case

① Big-oh (O) Notation:-

Big-oh notation is indicated with "O".

This notation is used to represent worst case. It is the maximum amount of time to run the algorithm.

Definition:- The function $f(n) = O(g(n))$ if and only if there exists ($\exists$) constant c and positive integers n_0 and n such that $f(n) \leq c \cdot g(n)$ for all ($\forall$) $n \geq n_0$, $n \geq 1$, $c > 0$



Here $f(n)$ and $g(n)$, are two non negative functions.

c is a constant

n and n_0 are two positive integers

This notation specifies the upper boundary.

eg:- $f(n) = 3n+2$ $g(n) = n$ Prove that

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

For Bigoh Notation the relation ship is

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

select $c=1, n=1$

$3n+2 \leq c.n$

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

$5 \leq 1$ False.

select

$c=1, n=2$.

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

$8 \leq 2$ False

Select $c=1, n=3$

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

$11 \leq 3$

if $c=2, n=1$

$5 \leq 2$ False

if $c=2, n=2$

$8 \leq 4$ false

if $c=3, n=1$

$5 \leq 3$ false

if $c=3, n=2$

$8 \leq 6$ false

if $c=4, n=2$

$8 \leq 8$ TRUE

if $c=5$ $n=1$

$6 \leq 5$ TRUE

if $c=5$ $n=2$

$8 \leq 10$ TRUE

For $C=4$, $n \geq 2$ we said that $f(n) \leq C \cdot g(n)$. So $f(n) = O(g(n))$

ex:- $f(n) = 2n+2$ and $g(n) = n^2$ then prove that $f(n) \leq O(g(n))$

$2n+2 \leq n^2$

if $C=1$, $n=1$

$2+2 \leq 1$ False.

if $C=1$, $n=2$

$4+2 \leq 4$

$6 \leq 4$ False

if $C=1$, $n=3$

$6+2 \leq 3^2$

$8 \leq 9$ True

if $C=1$, $n=4$

$8+2 \leq 4^2$

$10 \leq 16$ True.

So it is proved for $C=1$ and $n \geq 3$

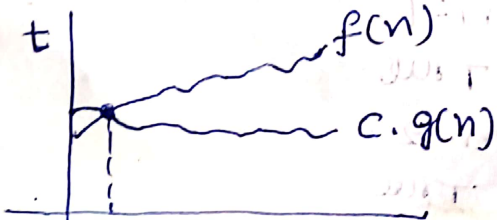
(2) omega(Ω) Notation :-

omega(Ω) is used to represent best case time complexity

It takes minimum amount of time to run an algorithm.

So, it is called as lower boundary

Definition:- The function $f(n) = \Omega(g(n))$ if and only if there exists ($\exists$) constant C, positive integers n, n_0 such that $f(n) \geq C \cdot g(n)$ for all ($\forall$) $n \geq n_0$, $n_0 \geq 1$ and $C > 0$



ex1:- $f(n) = 2n+2$ and $g(n) = n$ prove that $f(n) \geq \Omega(g(n))$.

for $c=1, n=1$

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

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

$4 \geq 1$ True.

So $f(n) = \Omega(g(n))$ for all $c=1, n \geq 1$.

ex2:- $f(n) = 3n+2 \quad g(n) = n$

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

for $c=1, n=1$

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

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

$5 \geq 1$ True.

So it is proved $f(n) = \Omega(g(n))$ where $c=1, n=1, n_0=1$. $[n \geq n_0 \rightarrow n \geq 1]$

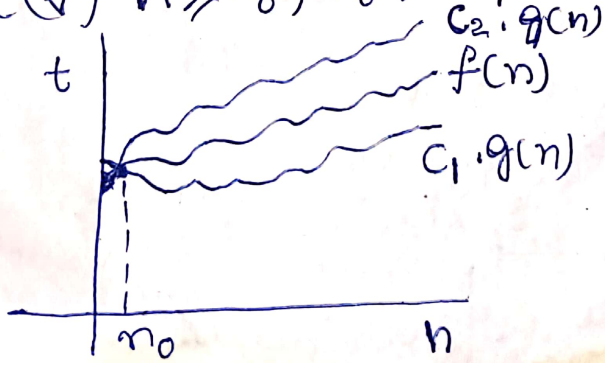
(3) Theta Notation(Θ) :-

Theta notation is indicated with "Θ". It is used to represent average case time complexity.

This is the average time required to run an algorithm. It is called as average boundary

Definition:-

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



ex:- $f(n) = 2n+2$, $g(n) = n$ prove that $f(n) = \Theta(g(n))$

$f(n) = 2n+2 \qquad g(n) = n$

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

$c_1 * n \leq 2n+2 \leq c_2 * n$

for $c_1 = 1$, $n = 1$, $c_2 = 4$.

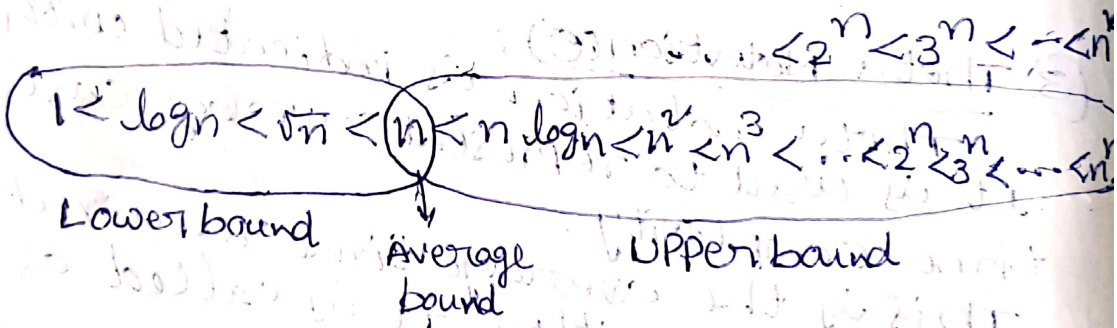
$1 * 1 \leq 2(1)+2 \leq 4 * 1$

$1 \leq 4 \leq 4$

So it is proved $f(n) = \Theta(g(n))$

where $c_1 = 1$, $n = 1$, $c_2 = 4$ $n_0 = 1$

Time complexities can be ranged from $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots$



Performance Measurement

Performance measurement is used to obtain the space and time complexity by running a program the space and time requirements depend on the compilers and computer system.

It is used to determine which algorithm is better than the other one.

This can be done by calculating worst case time complexity (O notation)

Suppose we want to measure the worst case performance of the sequential search. algorithm as follows.

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

$T(P) = 2n+4 = O(n)$.

The worst case time complexity of this Sequential search is $O(n)$.

This is a linear time complexity.

So asymptotic time complexity curve can be drawn as straight line interms of n value and time t.

Straight line can be obtained by knowing the atleast two values.

All other values are automatically predicted along the line path.

The asymptotic analysis has two limitations.

1) This analysis does not follow the asymptotic curve for small values of n.

We have to find out what value of n, the run time follows the asymtotic curve.

2) Some times, the times may not lie exactly on the curve path (straight line) because of elimination of the low-order terms in this asymptotic analysis.

For example, the time complexity $c_1 n + c_2 \log n + c_3$, we have to consider higher-order term $c_1 n$ as the time complexity for some constant $c_1 > 0$.

We expect that the asymptotic behaviour of Algorithm that is specified above begins for some $n < 100$.

So, for $n > 100$ we obtain the runtime of few values.

Better choice of $n = 200, 300, 400 \cdots 1000$

Above algorithm expresses worst-case behaviour when x is selected such that it is not one of the $a[i]$'s.

To obtain the definiteness, we set $a[i] = i$, $1 \le i \le n$ and $x = 0$.

By considering all these points, we design other algorithm that is specified below. This measures worst-case time complexity.

```
Algorithm TimeSearch( )
{
  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 to 20 do
    {
      h := GetTime( );
      K := SeqSearch(a, 0, n[j]);
      h1 := GetTime( );
      t := h1 - h;
```

Here GetTime() method returns the current time in milliseconds.

The timing results of this algorithm is shown below:

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

Here times are very small. Most of the times are 0.

This indicates that the precision of clock is not suffient.

Here nonzero times are just noise. so they are not useful to measure the time taken by an algorithm.

To measure the time for short event, it is required to repeat it several times.

In this case, we divide the total time for events by the number of repitions.

To estimate the time complexity, the sequential search algorithm can be modified as follows:

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

Here DESIRED-TIME is clock time.

Generate the Test data:- We have to generate the test data to measure the worst and average case time. This can be done by taking the random

Here GetTime() method returns the current time in milliseconds.

The timing results of this algorithm is shown below:

n	time	n	time
0	0	100	0
100	0	200	0
200	0	300	1
300	0	400	0
400	0	500	1
500	0	600	0
600	0	700	0
700	0	800	1
800	0	900	0
900	0	1000	0

Here times are very small. Most of the times are 0.

This indicates that the precision of clock will not sufficient.

Here many zero times are just noise. So they are not useful to measure the time taken by an algorithm.

To measure the time for short event, it is required to repeat it several times.

In this case, we divide the total time for event by the number of repitions.

To estimate the time complexity, the sequential search algorithm can be modified as follows

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

Here DESIRED-TIME is clock time.

Generate the Test data:- We have to generate this test data to measure the worst and average case time. This can be done by taking the random

Randomised Algorithms

A randomised algorithm makes use of a randomizer (random number generator)

In an algorithm, we need to take some decisions.

These decisions depend on the output of the randomizer

The output of a randomizer may vary from run to run.

So the output of the randomized algorithm also differ from run to run for the same input.

The execution time is also changed according to it

Randomized algorithms can be categorized into two classes.

i) Las Vegas algorithms ii) Monte Carlo algorithms

i) Las Vegas algorithms:-

These algorithms produce the same output for the same input.

The execution time of a Las Vegas algorithm depends on the output of a randomizer.

i.e., execution time can be characterized as a random variable. i.e output of randomizer.

Ex:- Randomized quick sort.

ii) Monte Carlo algorithms:-

These algorithms produce different outputs from run to run.

They produce mostly or probably correct output.

Ex:- Randomized primality Testing

Advantages:-

i) Simplicity:- These are simple in nature because they use pseudo random number generator.

ii) Very efficient:- Randomized algorithms are efficient than deterministic algorithms. In case of deterministic algorithm the inputs and outputs are fixed. They are not fixed in case of randomized algorithms.

iii) Randomized algorithms are faster than deterministic algorithms.

iii) Computational complexity is better than the deterministic algorithms. There is no fixed path from input to output like deterministic algorithms. So they can take less time to produce output.

Disadvantages:-

i) Less quality:- It depends on the quality of a random number generator that is used as part of the algorithm.

ii) Reliability is an issue:- This type of algorithm may not give any guarantee about the solution.

iii) Hardware fail:- These algorithms use randomizer which is a part of the hardware. Some times it may fail.

We have two applications of randomized algorithms.

(I) Primality Testing 2) Identifying the repeated elements.

(1) Primality Testing:- Any integer greater than one is said to be prime if and only if only divisor are 1 and the integer itself.

Fermat's Theorem:-

If P is prime number and $0 < A < P$ then $(A^{P-1} - 1) \% P = 0$.

Ex: $A = 2, P = 7$

$2^{7-1} - 1 = 63$ [$63 = 7 \times 9$]

$63 \% 7 = 0$

So 7 is a prime number.

If a number is prime then theorem holds.

If a number is composite then theorem sometimes holds.

We have to select smaller numbers.

if the test fails the number is definitely not prime number.

otherwise, it is likely to be prime number

Algorithm primalityTest (n, K)

```
{ isprime := True;
  for i := 1 to K do
  { A := randomInt (2, n-2)
    if (A^(n-1) - 1 % n ≠ 0) then
    isprime = False
  }
  return isprime;
}
```

(3) Identifying the repeated elements:-

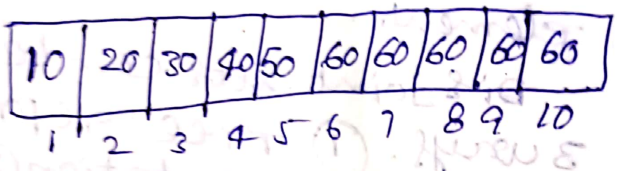
Randomized algorithms are also used to detect repeated elements that are present in an array.

Consider an array a[] of n numbers that has $n/2$ distinct elements and $n/2$ copies of another element.

Here we have to findout repeated element in an array.

Algorithm RepeatedElement (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 the range [1,n]
    if ((i ≠ j) and (a[i] = a[j])) then
      return i;
  }
}
```



We have to take two random indices i and j such that they are not equal. But the elements in i and j indices (a[i] & a[j]) are same.

Here the time taken by the algorithm to search repeated element based on i and j values that are randomly generated.

//