→ Preliminaries of Algorithm :-

i) Def of Algorithm :- An algorithm is a Finite Set of instructions that are used to accomplish a specific task.

ii) properties of Algorithm :-

* Input :- The algorithm must be supplied with zero or more input that can be specified externally
* output :- The algorithm must produce atleast one result as output
* Definiteness :- Each instruction in the algorithm must be clear and unambiguous.
* Finiteness :- The algorithm should terminate after certain number of steps
* Effectiveness :- Every instruction in an algorithm should be so basic that it can be carried out even with the help of paper and pencil.

iii) How to write an algorithm :-

There are some Rules for writting an algorithm

- Algorithm is a procedure consisting of heading and body. The heading consisting of name of the algorithm and parameter list

Syntax :-

Algorithm name (parameter1, parameter2 ---- parametern)

- The starting and ending of block's are indicated by { and } brackets.

- Delimiters (;) are used at the End of each Statement
- Single line comments are writting using // as beginning of comment
- the Identifier should begin by letter and not by digit. An identifier can be combination of alpha-numeric string
- assignment operator := an assignment statement Can be write as below

"variable := Expression;"

and In other way some of algorithms is used to "←" denote assignment operator.

- other operatores like <u>booleam operates</u> such as "True" or "False" <u>logical operates</u> are "AND", "OR", "NOT" and <u>relational operators</u> are "<", "<=", ">=", ≠.
- Array are stored with in "[" and "]" brackets the Index of array start at zero.
- Inputs and outputs are using "Read" and "write"
- <u>conditional statements</u> are if-then or if-then -else statements

```
if (condition) then
  {
    Statements
    - - - -
    - - - -
  }
```

```
If (Condition) then
{
  Statements block1;
  - - - - -
  - - - - - - -
}
else
{
  Statement block2;
  - - - -
}
```

If-then statement is of compound type then { and } should be used for enclosing block.

- while statement can be written as:

```
while (condition) do
        {
        statement1
        statement2
        - - - -
        - - - - -
        statementn
        }
```

- This condition is true the block enclose with { and } gets executed otherwise statement after } will be executed.
- general form of for loop is

```
For (initialization; condition, inc/decrement/
                                    update)
    {
        statement block;
    }
    statement Y;
```

- Repeat-until statement can be written as

```
repeat
        statement1
        statement2
        = = = = =
        statementn
until (condition)
```

• Break statement is used to exit from inner loop the return statement is use to return control from one point to another and while exiting from function.

example :- write an algorithm for find factorial of 'n' number

Algorithm Fact(n)

{

If (n <= 1) then

return 1

else

return n* Fact(n-1);

}

→ Algorithm analysis and Complexity :-

* the efficiency of an algorithm can be decided by measuring the performance of an algorithm
* we can measure the performance of an algorithm by computing amount of time and storage requirement
* Efficiency of an algorithm is in terms of time and space complexity

Time complexity :-

* The time complexity of an algorithm is the amount of computing time required by an algorithm to run to completion

* There are two types of computing time they are i) compile time ii) Run time. the time complexity is generally computed using run time or execution time.
* It is difficult to compute the time complexity in terms of physically clocked time.
Ex:- multiuser system, executing time depends on many factors such as system load, no of other programs running, instruction set used
* time complexity is given in terms of frequency count
* Frequency count is basically a count denoting no. of times of execution of statement

Example :- Compute the input-size of below program

```
Algorithm Sub (A, B, C, m, n) ------ 0
{ ------------------------ 0
  For i:=1 to m do ------------ m+1
    For j:=1 to n do ---------- m(n+1)
      C[i, j] := A[i, j] - B[i, j]; --- mn
} ------------------------ 0
                              2mn + 2m + 1
```

Here the input size is 2mn + 2m + 1 and also frequency count for above code is 2mn + 2m + 1
the input size gives the no. of accesses made to the programming statements

Space Complexity :-

*Space complexity of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.

* Space needed by a program depends upon the 2 part i) Fixed part ii) variable part

i) Fixed part :- It varies from problem to problem It includes the space needed for storing instruction constants, variables

ii) variable part :- It varies from program to program - It includes the space needed for recursion stack, and the structured variables that are allocated space dynamically during the runtime of a program.

iii) Asymptotic Notations :-

* 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 notations are easy way to represent the time complexity.
* By using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" "average possible" times.

* various notations are Ω, Θ and O are called asymptotic notations.

Big-oh Notation :- Big-oh notation denoted by 'O' (where 'O' stands for "order of") is a method of representing the upper bound algorithm running time

using Big-oh notation we can give longest amount of time taken by the algorithm to complete

let $f(n)$ and $g(n)$ are two non-negative functions and if there exists an integer number and constant 'c' such that $c>0$ and for all integers $n>n_0$, $f(n) \leq c * g(n)$ then $f(n)$ is big oh of $g(n)$. It is also denoted as "$f(n) = O(g(n))$"

let 'n' be the size of the input and $f(n)$ $g(n)$ be +ve functions of n

Ex :- $f(n) = 2n+2$ we have to find constant 'c'

$g(n) = n^2$

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

$2n+2 \leq n^2$ $\quad c = 1$ or 2

$n=1$ $\quad 2 \cdot 1 + 2 \leq 1^2$ $\quad n=2$ $\quad 2 \cdot 2 + 2 \leq 2^2$

$2+2 \leq 1$ $\quad 4+2 \leq 2^2$

$4 \leq 1$ $\quad 6 \leq 4$

when $n>2$ we obtain $f(n) < g(n)$ $\therefore O(n^2)$ for $n>2$ $[n>n_0]$

* various meanings associated with big-oh are.

$O(1)$	Constant Computing time
$O(n)$	linear
$O(n^2)$	Quadratic
$O(n^3)$	cubic
$O(2^n)$	Exponential
$O(\log n)$	logarithmic
$O(n \log n)$	nlogn time

* The Relation ship among these computing time is

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$

Omega Notation :- Omega notation denoted as 'Ω' is a method of representing the lower bound of algorithm's running time. using omega notation we can denote shortest ~~and~~ amount of time taken by algorithm to complete

let $f(n)$ and $g(n)$ are 2 non-negative functions and if there exists contant C and integer number such that $C \geq 0$ and $n > n_0$ then

$f(n) > C * g(n)$

f(n) is omega of g of n and denoted as

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

Ex: $f(n) = 2n+5$

$g(n) = 2n \quad n>1$

$2n+5 = \Omega(2^c n)$

$2n+5 = \Omega(n)$

Theta Notation :-

Theta Notation denoted by 'Θ' is a method of representing running time between upper bound and lower bound

let f(n) and g(n) be two non-negative functions there exit positive constant c_1 and c_2 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ and f(n) is theta of g of n

Ex :- $f(n) = 2n+8 > 5n$ where $n \geq 2$

$= 2n+8 \geq 6n$ where $n \geq 2$

$= 2n+8 < 7n$ where $n \geq 2$

$2n+8 = \Theta(n)$

$c_1 = 5$, $c_2 = 7$ and $n_0 = 2$

little-oh Notation:-

The little oh is denoted as o. It is defined as let f(n) and g(n) be the non-negative functions then

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

$f(n) = o(g(n))$ i.e f of n is little oh of g of n

$f(n) = o(g(n))$ it and only if $f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$

Best, worst, Average case analysis :-

Best case time complexity :- If an algorithm takes minimum amount of time to run complet for a specific set of input then it is called best case time complexity

Eg :- searching element by using sequential Search we get the desired element at first place itself then it is called best case time complexity

worst 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

Eg :- 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

Average time complexity :-

-the time complexity that we get for certain set of inputs is of a average same. then for corresponding input such a time complexity is called average case time complexity

Data structure :- (Def) A data structure is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.

*Data structures are widely ~~used~~ applied in the following areas

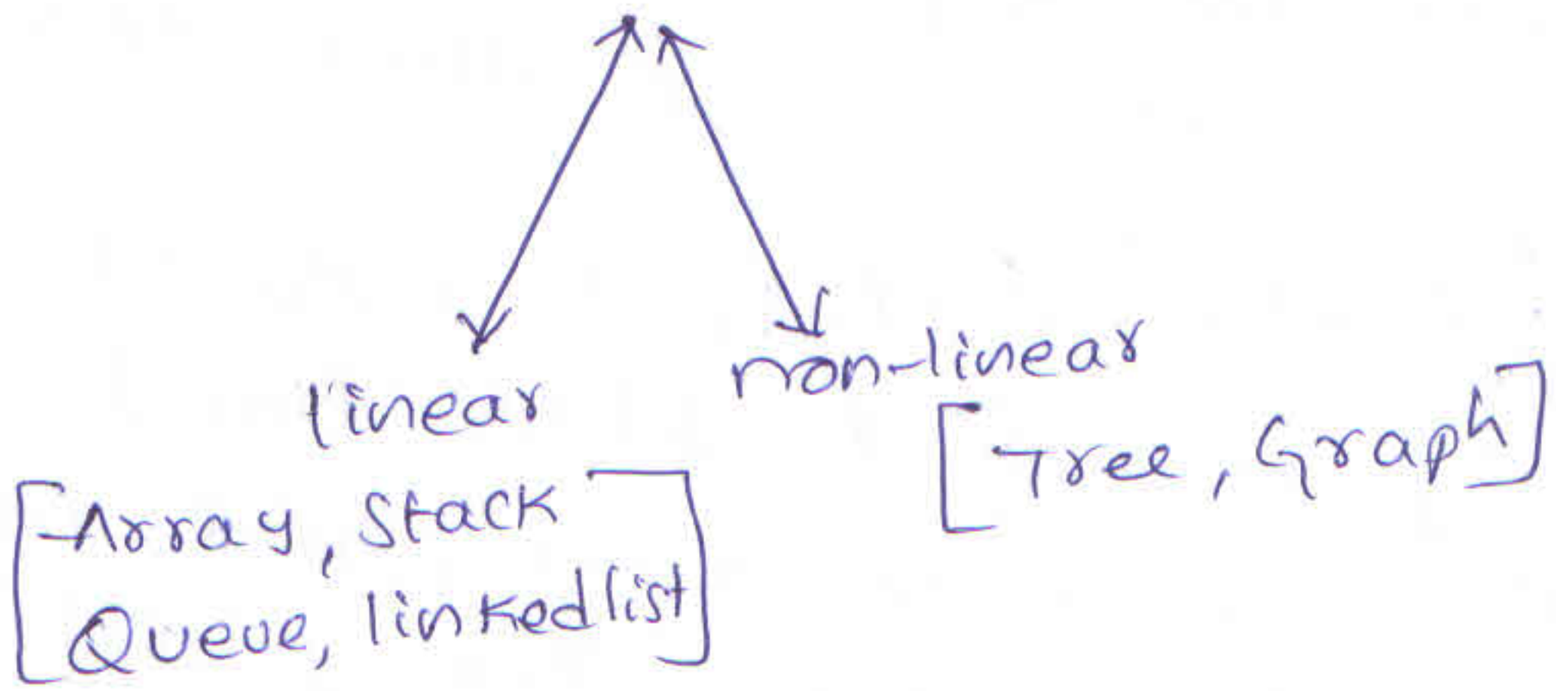
compiler Design, DBMS, Graphics
Operating System, Artificial Intelligence

*Data structures are generally categorized into 2 classes i) primitive and ii) non-primitive

primitive data structures are the fundamental data types which are supported by a programming languages some basic data types are integer, real, character & boolean.

*Non-primitive data structures are those data structures which are created using primitive data structures.

* non-primitive data structures are classified into two categories linear and non-linear



Data Structure Operations:-

Traversing:- Accessing each record exactly once so that certain items in the record may be processed

Searching:- Finding the location of the record with a given key value

Insertion:- Add a new record to the structure

Deletion:- Removing a record from the structure.

→ Recursion :- Def :- Recursion is the process of calling functions again and again is called Recursion.

→ ~~Recess~~

→ Design methodology and Implementation of Recursive Algorithm :-

* For every recursive call there are 2 types of elements. One element is a kind of element that solves the problem
* Second element is that reduces the size of the problem recursively
* The statement which solves the problem is called Base case
* Rest of the program is called general case

Eg :-

```
int fact(int n)
{
    int x, y;
    if (n==0)
        return 1;
}
```

} Base case

```
x = n-1;
y = fact(x); // recursive call
return(n*y);
}
```

} General case.

→ linear and Binary Recursion :-

linear Recursion :- this recursion is the most commonly used. In this recursion a function call itself in a simple manner and by termination condition it terminates. this process called "winding" and when it returns to caller that is called "un-winding" Termination condition also known as Base condition

Factorial calculation by linear Recursion :-

```
~~int fact(long~~ int Fact (long n)
{
    if (0>n)
    return -1;
    if (0==n)
     return 1;
    else
  {
     return (n* Fact(n-1));
  }
}
```

winding process :-

Fact (6) 6 * Fact (5)
Fact (5) 5 * Fact(4)
Fact (4) 4* Fact(3)
Fact (3) 3* Fact(2)
Fact(2) 2* Fact(1)
Fact(1) 1 * Fact (0)

terminating point
Fact(0) 1

unwinding process :-

Fact(1) 1*1
Fact (2) 2*1
Fact(3) 3*2*1
Fact(4) 4*3*2*1
Fact(5) 5*4*3*2*1
Fact-(6) 6*5*4*3*2*1

Binary Recursion :- Binar recursion is a process where function is called twice at a time inplace of once at a time.

```
int FibNum (int n)
{
    if (n<=1)
      return -1;
    if (1==n || 2==n)
      return1;
    return FibNum(n-1) + Fibnum(n-2);
}
```

→ Recursive Algorithm For Factorial Function:-

* The Factorial function often denoted as n! describes the operation of multiplying a number by all the positive integers smaller than it.

Example $5! = 5*4*3*2*1$

$9! = 9*8*7*6*5*4*3*2*1$

$5! = 5*4!$

$5! = 5*4*3*2*1$

$= 5*4!$

$4! = 4*3!$

$4! = 4*3*2*1$

$= 4*3!$

Factorial of $n! = n*(n-1)!$

```
int Factorial (int n)
{
    return n * Factorial (n-1);
}
```

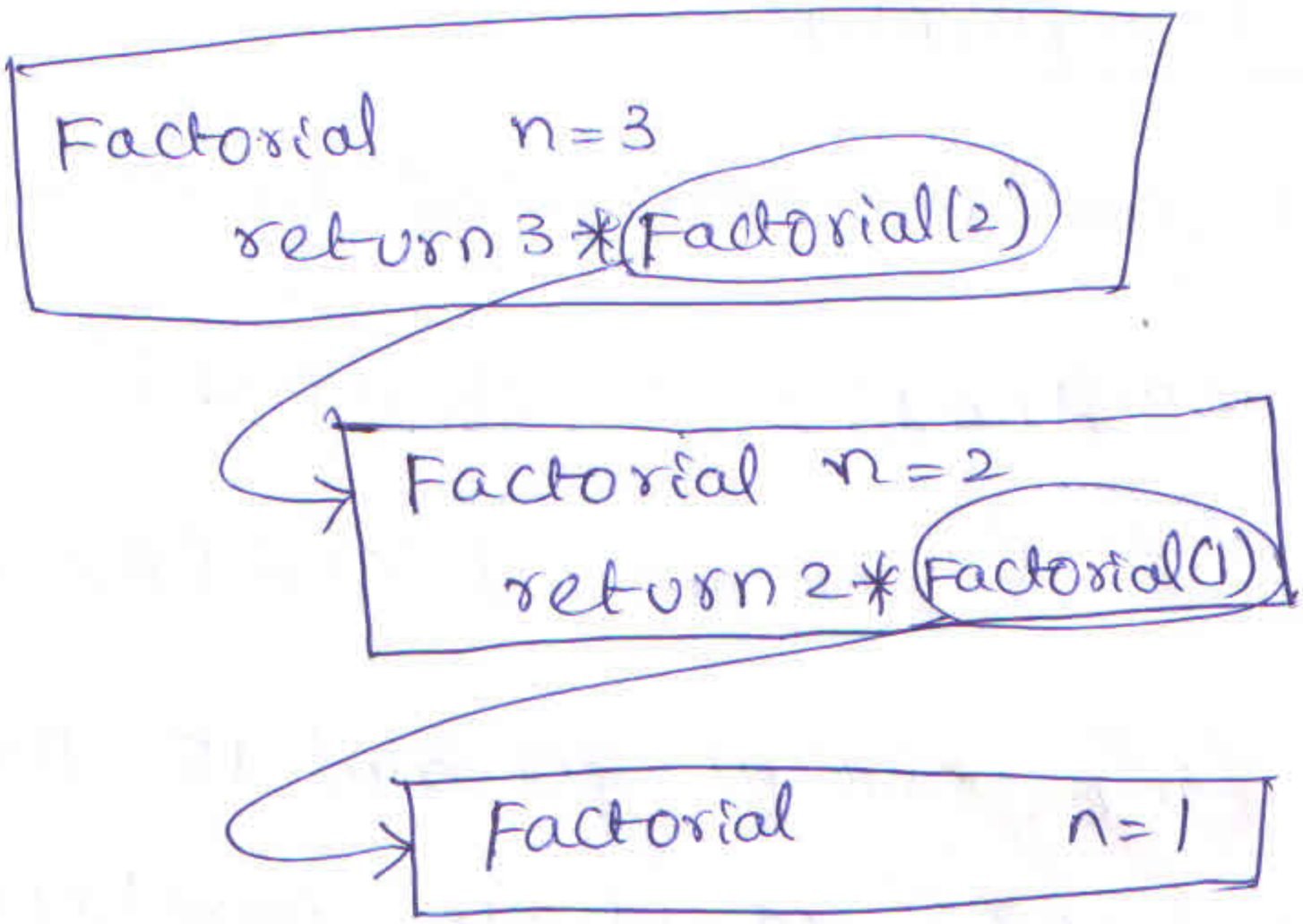
Factorial of 3!

Factorial (3) : $3*2!$

Factorial (3) return $3*$ Factorial (2)

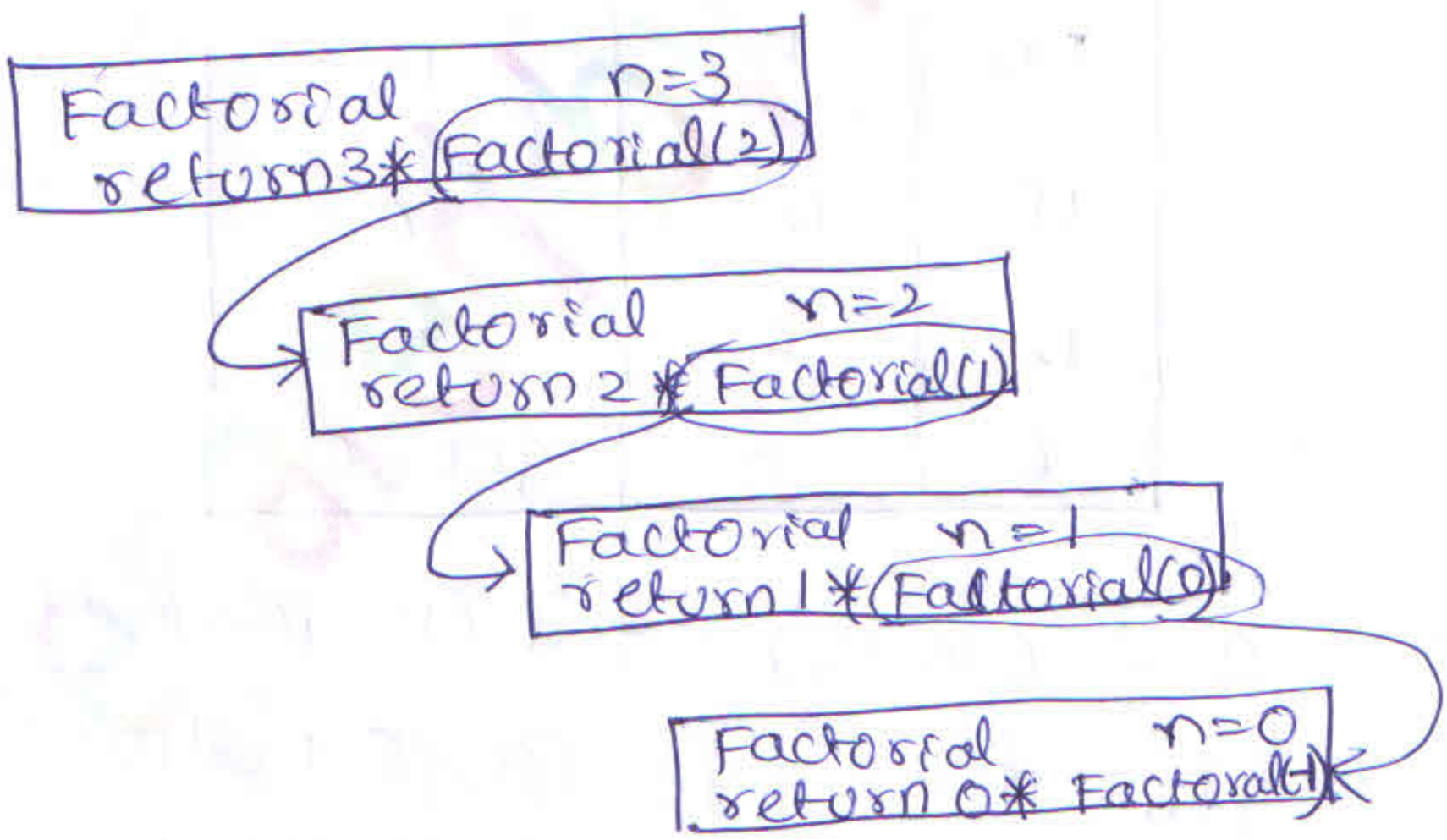
Factorial (2) : $2*1!$

Factorial (2) return $2*$ Factorial (1)



1! = 1 * 0!

Factorial(1) returns 1 * Factorial(0)




```
int Factorial (int n)
{
    if (n<0) return 0;
    else if (n<=1) return 1;
    else return n * Factorial (n+1);
}
```

→ GCD Computation :-

* GCD can be calcolated by using following function

$$gcd(a,b) = gcd(b, a \bmod b) \quad [a > b]$$

then we get gcd(a,0) then GCD = a

Eg :- two numbers 30 and 18 then we have to find GCD of those numbers

a	b	Reminder after a/b
30	18	12
18	12	6
12	6	0
6	0	GCD = 6

Eg :- gcd (10,15) = gcd (15, 10 mod 15) = gcd (15,10)
gcd (15,10) = gcd (10, 15 mod 10) = gcd (10,5)
gcd (10,5) = gcd (5, 10 mod 5) = gcd (5,0)
gcd (5,0) = 5

```
int gcd(int a, int b)
{
  while (a != b)
  {
    if (a > b)
      return gcd(a-b, b);
    else
      return gcd(a, b-a);
  }
  return a;
}
```

→ Fibnacci Sequence:-

*Fibnacci sequence is the series of numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ------

* The next number is found by adding up the two numbers before it.

The 2 is found by adding 2 numbers before it (1+1)
Similarly the 3 is found by adding the 2 numbers before it (1+2)

And the 5 is (2+3)

and soon.

* Example: The next number in the sequence above would be 21+34=55

* Fibonacci sequence can be written as a sequence

First terms are numbered from 0 onwards

n =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	----
x_n =	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	----

* Term 6 is called as x_6 (which is equals to 8)

Example: the 8th term is the 7th term plus the 6th term.

$$x_8 = x_7 + x_6$$
$$= 8 + 13$$
$$x_8 = 21$$

* The Rule can be written as:

$$X_n = X_{n-1} + X_{n-2}$$

where as X_n is term number "n"
X_{n-1} is the previous term (n-1)
X_{n-2} is the term before that (n-2)

* Example :- term 9 can be calculate as below

$$X_9 = X_{9-1} + X_{9-2}$$
$$= X_8 + X_7$$
$$= 21 + 13$$
$$= 34$$

Recursive solution

```
int Fib(int n)
{ if(n==0)
  {
    return 0;
  }
  elseif (n==1)
  {return 1;
  }
  else if (n>1)
  {
  return Fibo(n-1) + Fibo(n-2);
  }else
  { return -1;
  }}
```

→ Tail Recursion :- In this method recursive function is called at the least. So It's more efficient than linear recursion method

```
int FibNum (int n, int x, int y)
{
  if (1 == n)
  {
    return y;
  }
  else // Recursive call by Tail method
  {
    return FibNum (n-1, x, x+y);
  }
}
```

→ Linear Search :-

* linear Search also called as Sequential Search is a very Simple method used for Searching an array for a particular value
* It works by comparing every element of the array one by one in Sequence until match is found
* linear Search is mostly used to Search an unordered list of elements

Example :- array A[ ]

A[1] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5} Here Val = 7

* Val = 7 then Searching means to find out whether the value = 7 is present in the array or not
* If Yes then Search is Succesful and it returns the position of occurence of val

Here pos = 3

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
10	8	2	7	3	4	9	1	6	5

A[0] != val

0	1	2	3	4	5	6	7	8	9
10	8	2	7	3	4	9	1	6	5

A[1] != val

0	1	2	3	4	5	6	7	8	9
10	8	2	7	3	4	9	1	6	5

A[2] != val

0	1	2	3	4	5	6	7	8	9
10	8	2	7	3	4	9	1	6	5

A[3] != val

```
int linearsearch(int a[], int n, int key)
{
    if(n<0)
    {
      return -1;
    }
    if (key == a[n-1])
    {
      return n-1;
    }
    linear search (a, n-1, key);
}
```

Time complexity
O(n)

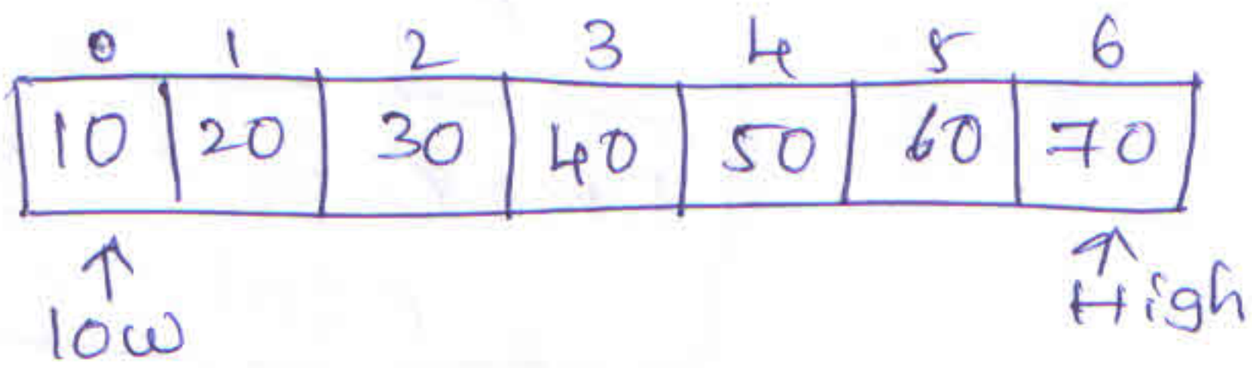
→ Binary Search

* Binary search is an efficient searching method while searching the elements using this method the most essential thing is that the elements in the array should be sorted one.
* An element which is to be searched from the list of elements stored in array A[0---n-1]

* A[m] be mid element of array A

There are 3 conditions

1. If key = A[m] then desired element is present in the list
2. Otherwise if key < A[m] then search the left sublist
3. Otherwise if key > A[m] then search the Right sublist



Key element is 60

$m = (low + high)/2$

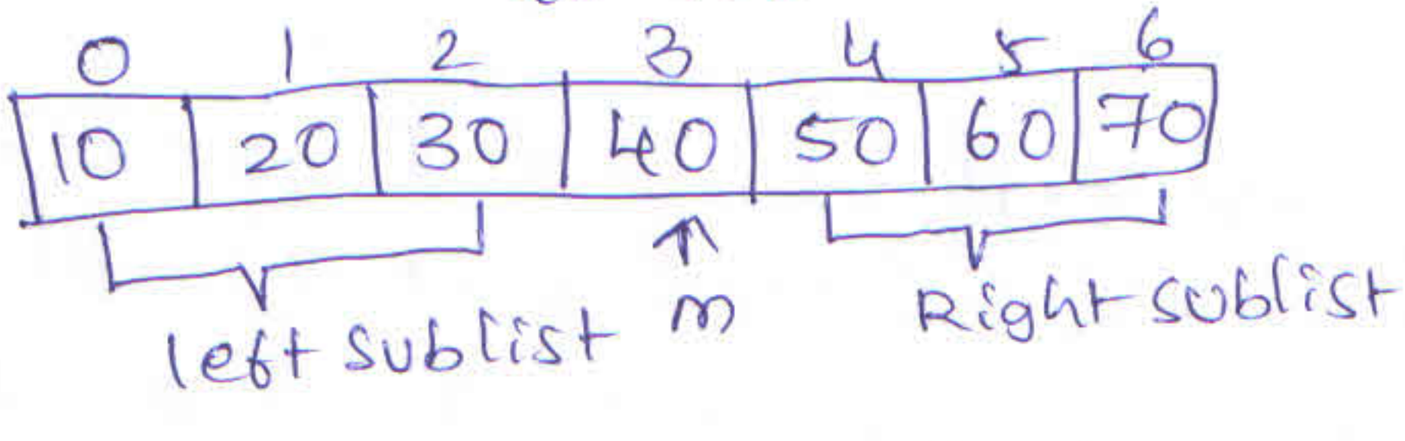
$m = (0 + 6)/2$

$m = 3$

Check A[m] = key

A[3] = 60

40 < 60 then search for right list



* Right list is

* Again divide the list and check mid.

4	5	6
50	60	70

m

$m = \frac{4+6}{2} = \frac{10}{2} = 5$

A[m] = key

∴ The number is present in the list

~~BinSearch(A[0---n-1])~~

```
Bin search (A[0---n-1], key)
    low := 0
    high := n-1
    while (low < high) do
    {
        m := (low + high)/2
        if (key = A[m]) then
            return m
        else if (key < A[m]) then
            high := m-1
        else
            low := m+1
    }
    return -1
```

Time complexity
O(n log n)

→ Fibonacci Search :-

* Fibonacci search is used to search an elemt of a sorted array with the help of Fibonacci numbers
* It studies the locations whose addresses have lower dispersion.
* Fibonacci number is subtracted from the index thereby reducing the size of the list.
* Fibonacci Series is

0 1 1 2 3 5 8 13 21 - - - -

* Example :-

1	2	3	4	5	6	7
10	20	30	40	50	60	70

n = 7

* we known the values for a, b and f

f = n = 7

* a and b values will be consida by using Fibonacci Series.

0 1 1 2 3 5 8

a b

Here a = 3, b = 5 because b is only elemnt in Fibonacci Series with n < 7

* So a = 3, b = 5, f = 7

* By using those values start searching key element from the list

* Each element will be compared with key element

arr[f]

If (key < arr[f])

f = f - a

b = a

a = b - a

If (key > arr[f])

f = f + a

b = b - a

a = a - b

* Key = 20

key < arr[f]

20 < 70

f = f - a = 7 - 3 = 4

b = a = 3

a = b - a = 2

* Compare key < arr[f]

∴ f = 4 = 40 20 < 40 → Yes.

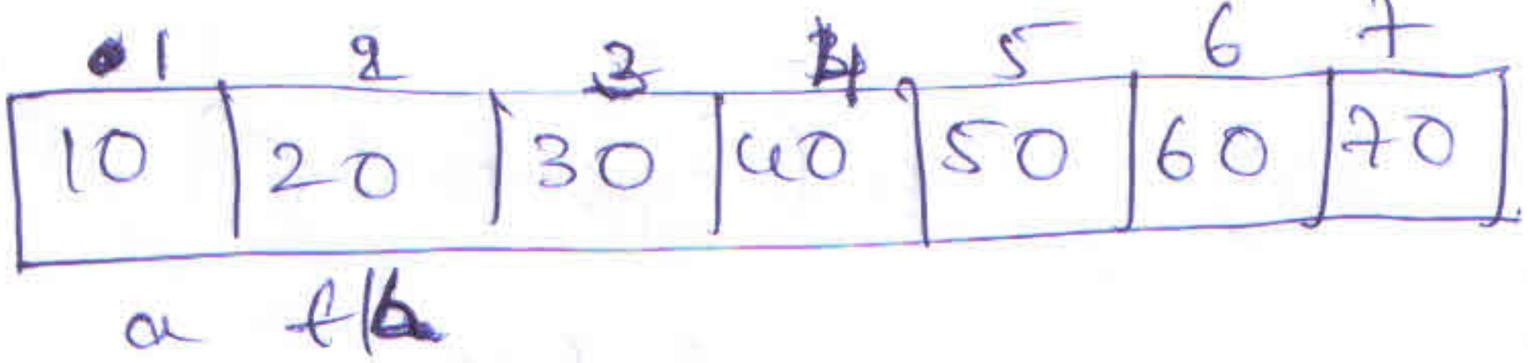
$f = 4, b = 3, a = 2$

$f = f - a = 4 - 2 = 2$

$b = a = 2$

$a = b - a = 3 - 2 = 1$

$f = 2, b = 2, a = 1$



* Element is present on location 2 = f

Sorting Techniques

Introduction

Sorting a set of items in a list is a task that occurs often in a Computer programming often, a human can perform this task intuitively. however a computer program has to follow a sequence of exact instructions to accomplish this. This sequence of instructions is called analgorithm. A sorting algorithm is a method that can be used to place a list of unordered items into an ordered sequence. the sequence of ordering is determine by a key. Various sorting algorithms exist and they differ in terms of their efficiency and performance some important and well known sorting algorithms are Bubble sort Selection sort etc.

→ **Sort order :-**

Sort order is nothing but the arrangement of the elements in some specific manner there 2 types of sortings are there.

1, Ascending order

2, Descending order

1. Ascending order :- It is the sorting order in which the elements are arranged from low value to high value. (or) increasing order

Ex :- 10, 50, 40, 20, 30 Before sorting order

After sorting 10, 20, 30, 40, 50

2. Descending order :- It is the sorting order in which the elements are arranged from high value to low value. (or) decreasing order.

Ex :- 10, 50, 40, 20, 30 Before sorting order

After sorting 50, 40, 30, 20, 10

→ Sort Stability :- The sorting stability means Comparing the records of same value and expecting them in the same order even after sorting them

Ex :- Radix Sort, Merge Sort

Heap Sort, Bubble Sort

Quick sort, Insertion Sort

above list is sort according to the first alphabet

After sorting by alphabetic order as follows

1. Bubble sort 2. Heap sort
3. Insertion sort 4. merge sort 5. Quick sort
6. Radix sort &, so

→ Efficiency and pases :

* The major issue in the sorting algorithms is its efficiency
* If we can efficiently sort the records then that adds value to the sorting algorithm
* usually efficiency can be denoted by efficiency of sorting algorithms in terms of time complexity
* time complexity can be denoted by big-oh notations ie : $O(n^2)$, $O(n \log n)$ etc.

ie for phases is as shown in below

10, 30, 20, 50 40

pas 1: 10, 20, 30, 50, 40

pas 2: 10, 20, 30, 40, 50

* There are various sorting techniques are there
  1. Insertion sort 2. Heap sort
  3. Bubble sort 4. Quick sort
  5. Radix sort 6. Merge sort

→ Sorting by Insertion :-

Insertion sort :- In this method the elements are inserted at their appropriate place. Hence is the name insertion sort

The best example for insertion sort is playing cards. The card player picks up the cards and inserts them in to the required position

Array 'a' with 'n' elements is indicated as below a[0], a[1] --- a[n-1]

pass 1 :- a[0] by itself is trivially sorted

pass 2 :- a[1] is inserted either before or after a[0] so that a[0], a[1] are sorted order.

pass 3 :- a[2] is inserted in to its proper place in a[0], a[1] that is before a[0], between a[0] and a[1] or after a[1] so that a[0], a[1] a[2] are sorted

⋮

pass n :- a[n-1] is inserted into its proper place in a[0], a[1] --- a[n-2] so that a[0], a[1] --- a[n-1] are sorted.

→ Sorting by Insertion :-

Insertion sort :- In this method the elements are inserted at their appropriate place. Hence is the name insertion sort

the best example for insertion sort is playing cards. the card player picks up the cards and inserts them in to the required position

Array 'a' with 'n' elements is indicated as below a[0], a[1] --- a[n-1]

pass 1 :- a[0] by itself is trivially sorted

pass 2 :- a[1] is inserted either before or after a[0] so that a[0], a[1] are sorted order.

pass 3 :- a[2] is inserted in to its proper place in a[0], a[1] that is before a[0], between a[0] and a[1] or after a[1] so that a[0], a[1] a[2] are sorted

⋮

pass n :- a[n-1] is inserted into its proper place in a[0], a[1] --- a[n-2] so that a[0], a[1] --- a[n-1] are sorted.

Example for insertion sort is as follows

Arrange the following elements in the sorted order using insertion sort. 30, 20, 35, 14, 90, 25, 22

Here n=7, 0 ---- n-1 positions 0 --to-- 6

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
30	20	35	14	90	25	22

pass 1 :- a[1] < a[0] Interchange the positions of elements

20	30	35	14	90	25	22
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

pass 2 :- a[2] > a[1] position of elements remain same position

20	30	35	14	90	25	22
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

pass 3 :- a[3] < a[0], a[1], a[2] so insert a[3] before a[0] we get

14	20	30	35	90	25	22
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

pass 4 :- a[4] > a[3] so position of elements remains same position

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
14	20	30	35	90	25	22

Pass 5 :- a[5] < a[2], a[3], a[4] so insert a[5]

before a[2]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
14	20	25	30	35	90	32

Pass 6 :- a[6] < a[4], a[5] so insert a[6] before a[4] position.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
14	20	25	30	32	35	90

After 6 passes the list of elements are placed in array in sorting order.

Algorithm :-

```
Insertionsort (A [0 ... n-1])
    For i ← 1 to n-1 do
  {
     temp ← A[i]
      j ← i-1
     while ((j >= 0) && (A[j] > temp)) do
  {
      A [j+1] ← A[j]
      j ← j-1
  }
    A[j+1] ← temp
}
```

Pass 5 :- a[5] < a[2], a[3], a[4] so insert a[5]

before a[2]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
14	20	25	30	35	90	32

Pass 6 :- a[6] < a[4], a[5] so insert a[6] before a[4] position.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
14	20	25	30	32	35	90

After 6 passes the list of elements are placed in array in sorting order.

Algorithm :-

```
Insetionsort (A [0 ... n-1])
    For i ← 1 to n-1 do
{
    temp ← A [i]
    j ← i-1
    while ((j >= 0) && (A [j] > temp)) do
{
    A [j+1] ← A [j]
    j ← j-1
}
    A [j+1] ← temp
}
```

Note :- Best case Time Complexity is $O(n)$

Average case Time Complexity is $O(n^2)$

worst case Time Complexity is $O(n^2)$

Advantages & Disadvantages

1, Simple to Implement

2, This method is efficient when we want to sort small no. of elements

3, more efficient than most other

4, This method is implemented for small list of elements only.

→ Sorting by Selection :-

Heap Sort :- Heap data structure is an array of elemeneuts that can be viewed as an almost complete binary tree.

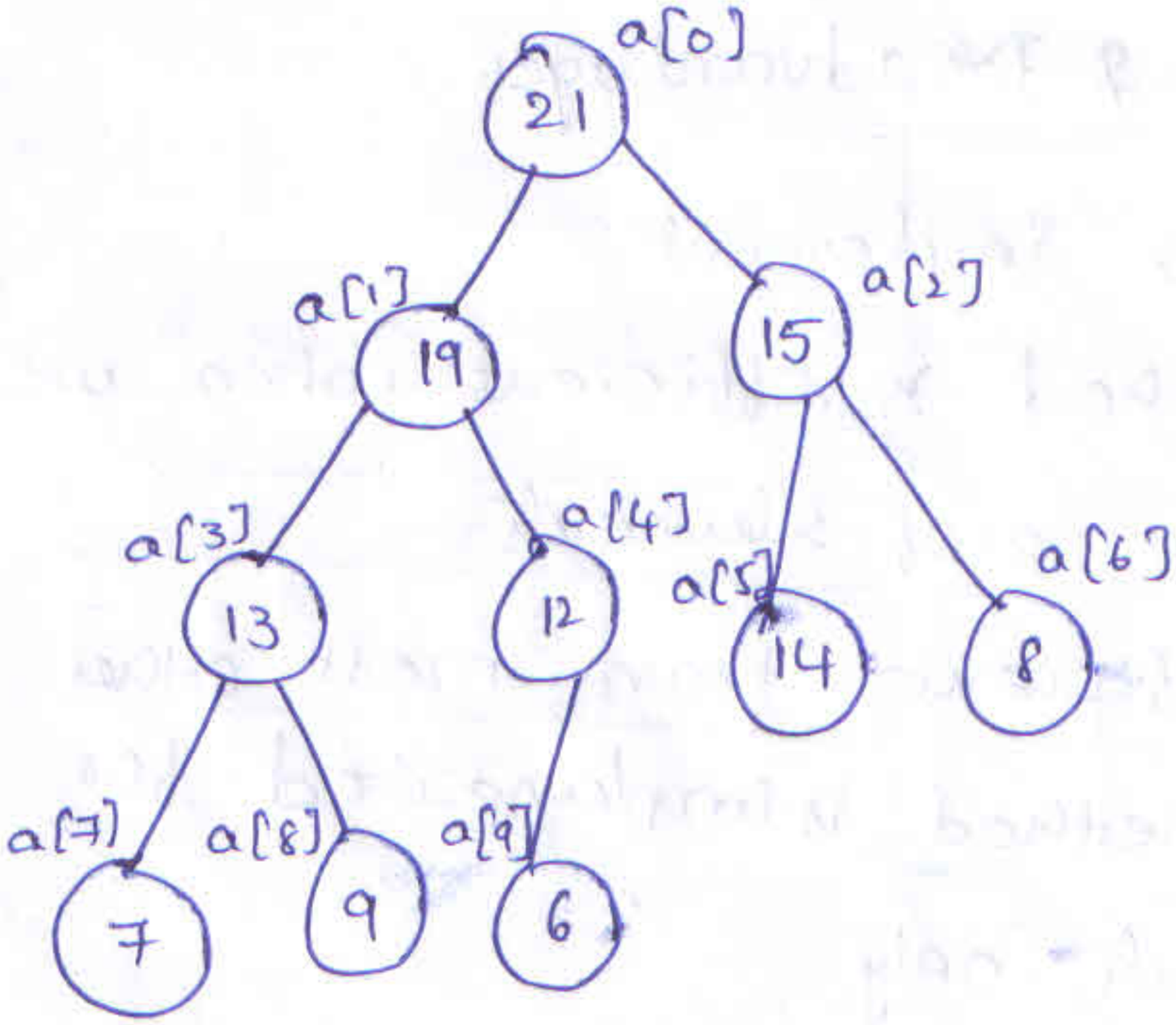
Each node vake of the tree corresponding to an element of the array

* length [a] → no. of elements in the array
* heapsize [a] → no. of elements in the heap sorted within array a.
* Index values can be represented as follows a[0], a[1], a[2] ----

Given elements are as shown in below

21, 19, 15, 13, 12, 14, 8, 7, 9, 6

* from the above group of elements we can construct the heap tree as follows



dig:- Binary tree

*By using above tree we can implement an array as follows

0	1	2	3	4	5	6	7	8	9
21	19	15	13	12	14	8	7	9	6

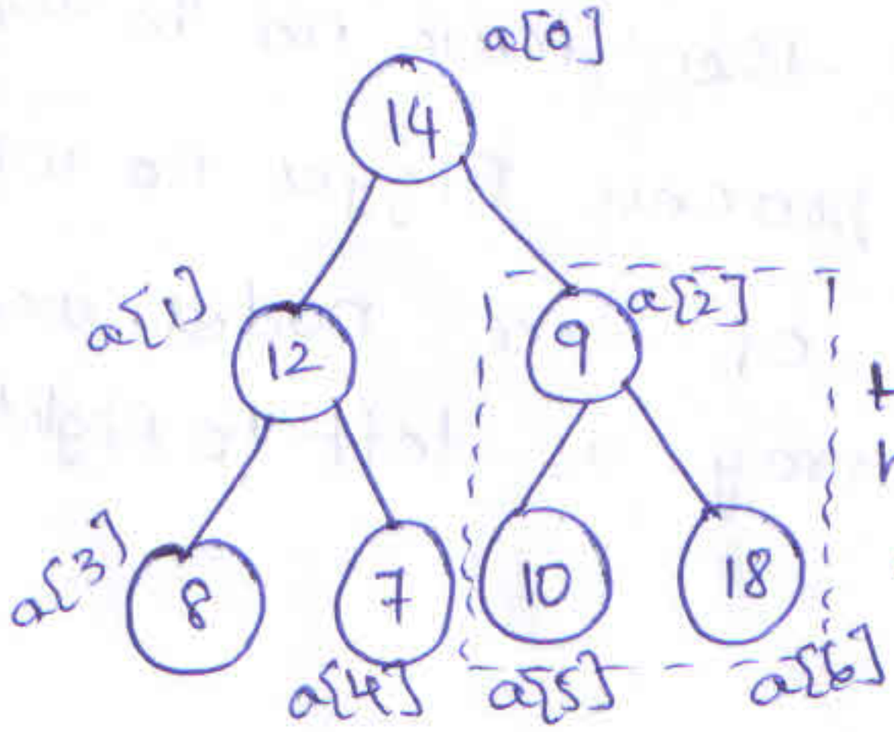
* From the above Binary tree there are 3 levels are there
  1, Root Node (or) parent
  2, left Node
  3, Right Node.

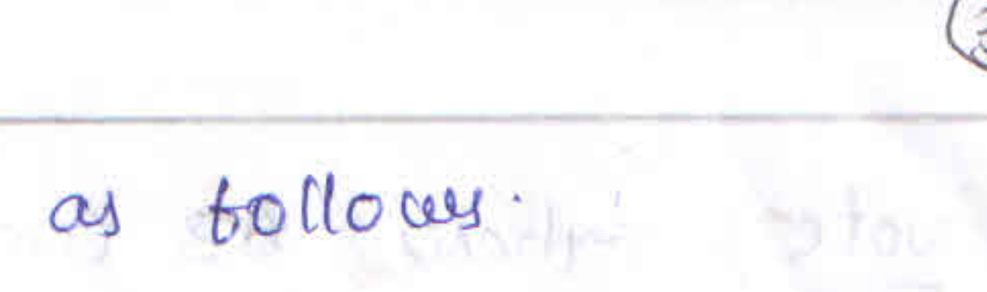
* Here every node 'i' is represented as a node in a tree

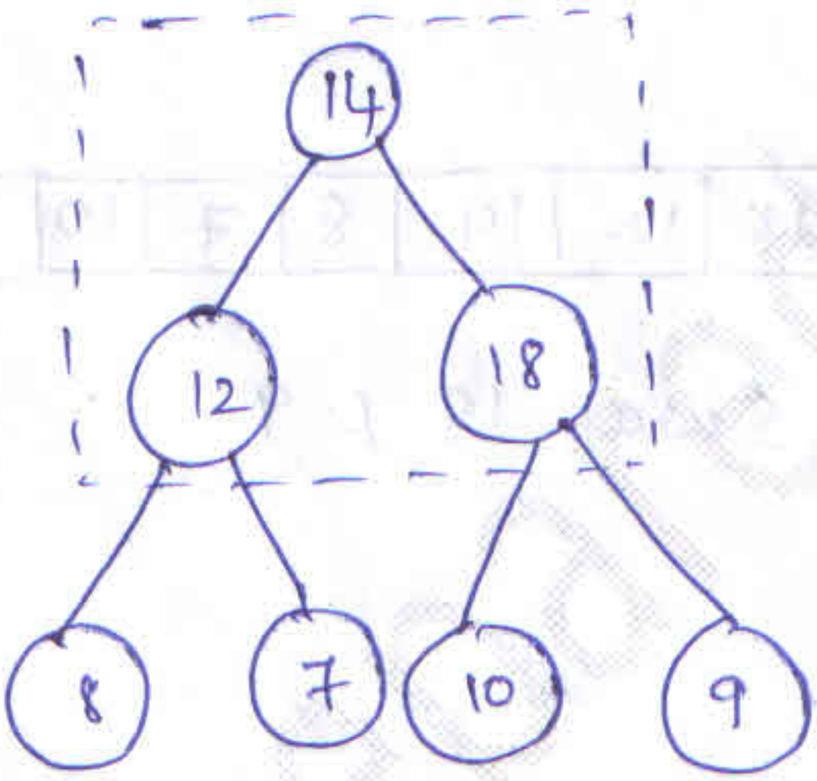
Example for heapsort is as follows.

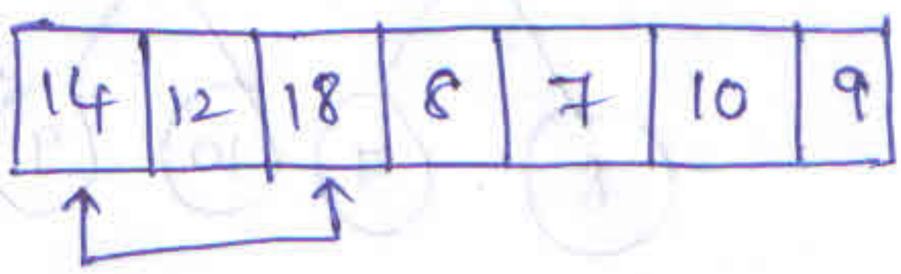
14, 12, 9, 8, 7, 10, 18

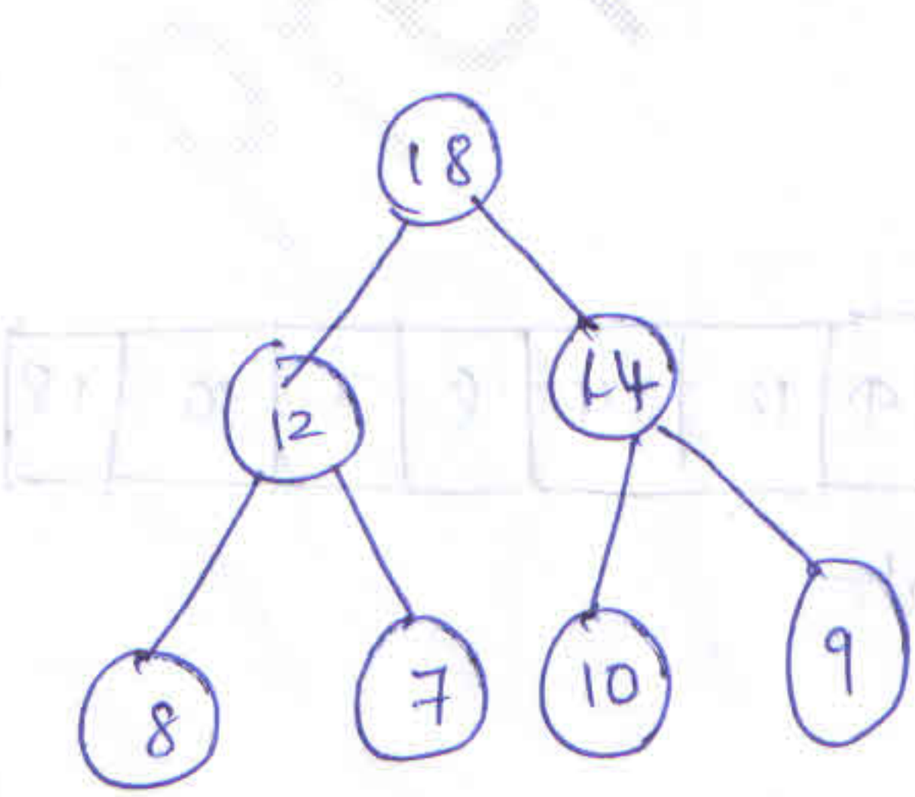
Stage 1 :- Heap Construction

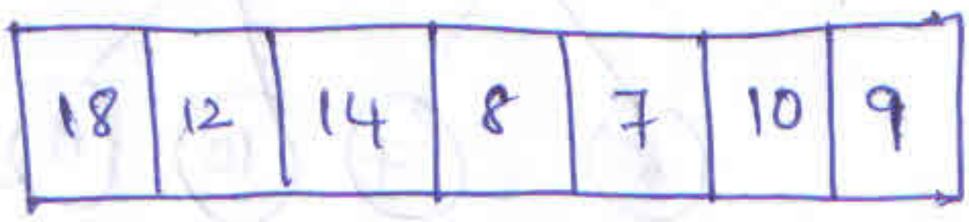








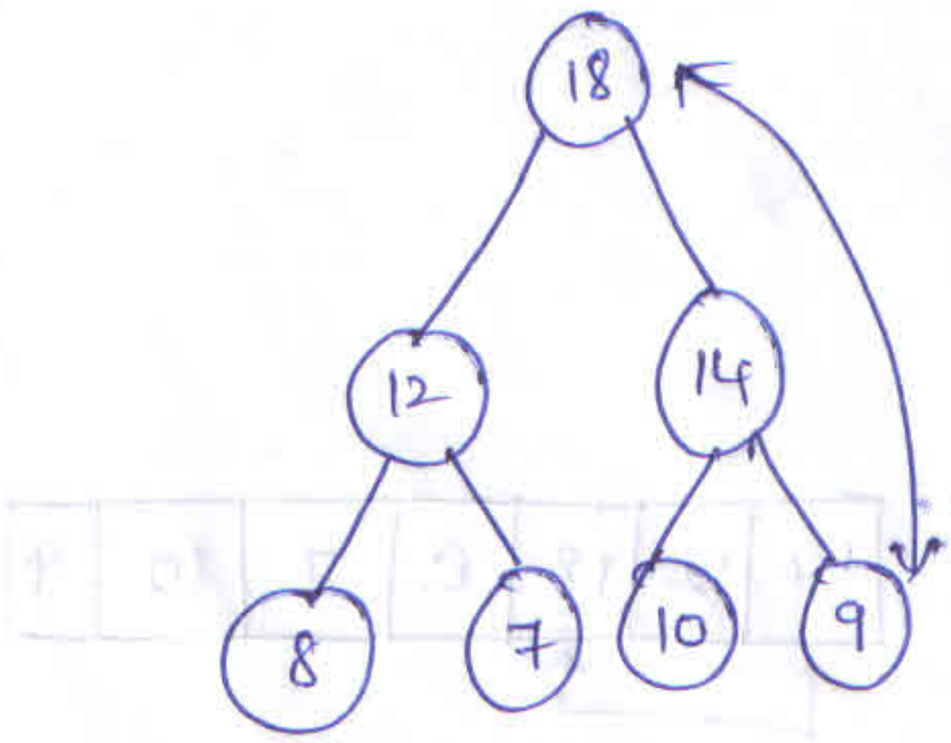




Note :- there are 3 conditions for Heap sort

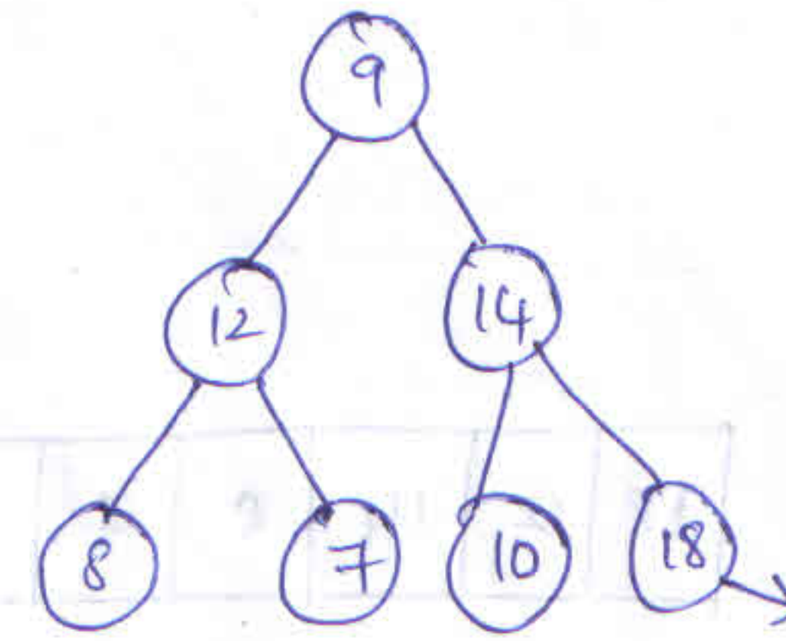
1, Construct the tree by using given data
2, Keep the elements in tree as order.
3, after that delete the max no in that tree. for deletion process Bigger no will be sents at End of tree nodes. and keep that no's in array as left to right

Delection process :-



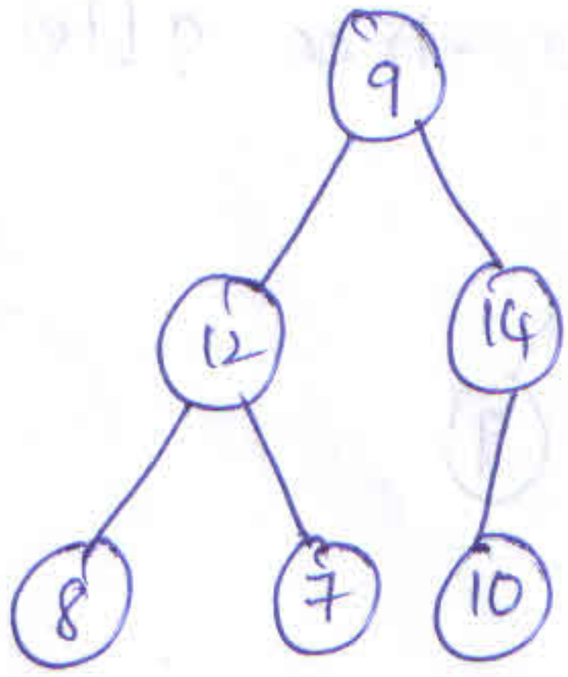
18	12	14	8	7	10	9

Swap 18 & 9



9	12	14	8	7	10	18

After deleting tree should be like below



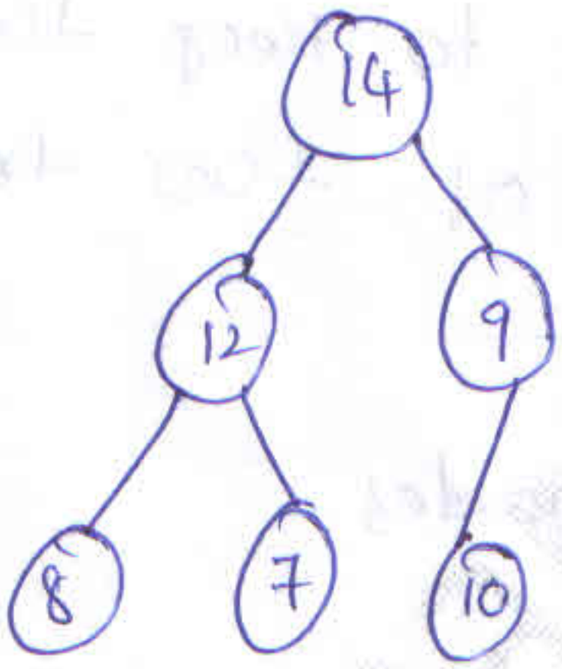
Time Complexity for Heapsort

Average case :- O(n log n)

Best case :- O(n log n)

worst case :- O(n log n)

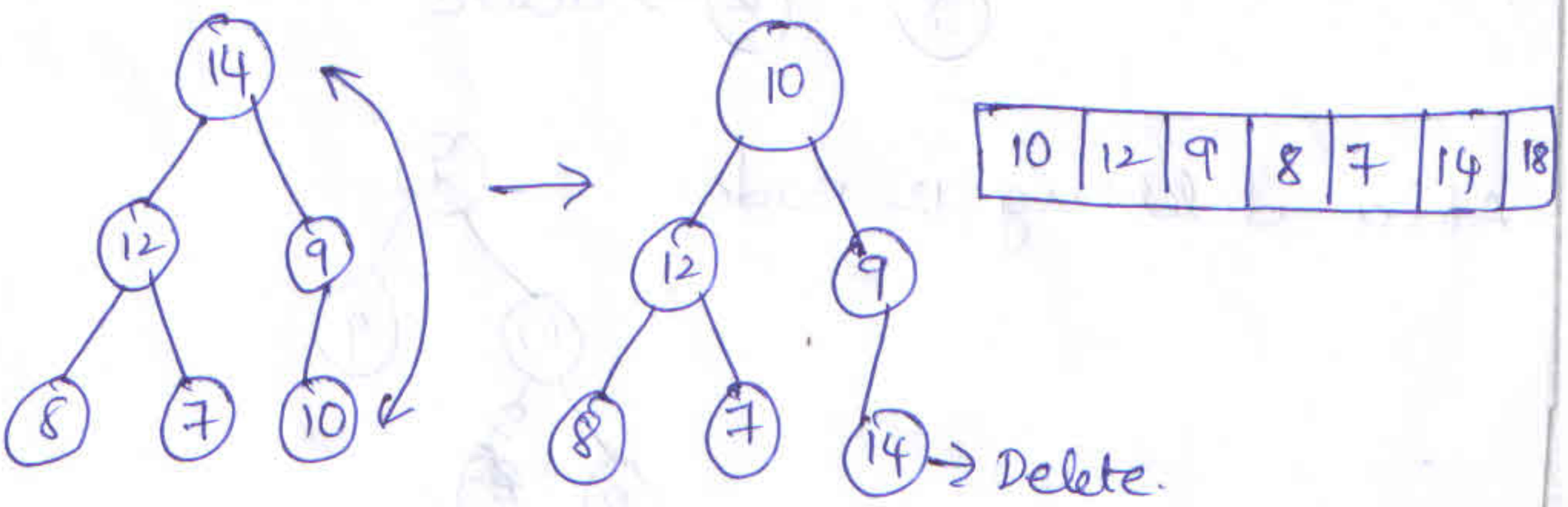
Heapify the above tree (order tree)



→ Heapified.

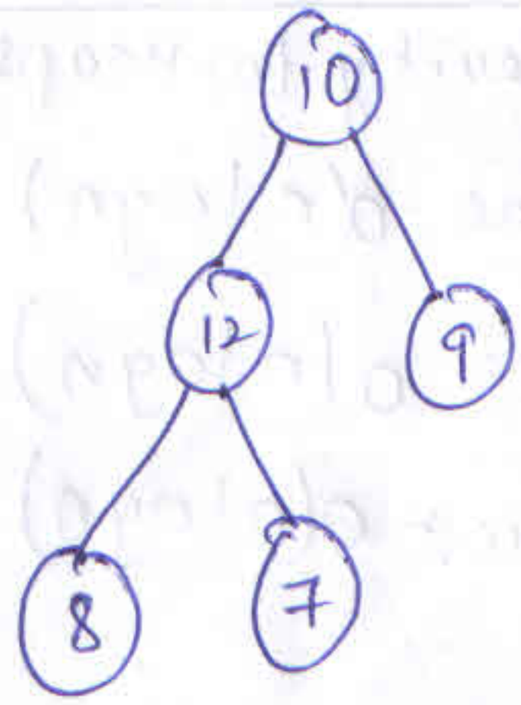
14	12	9	8	7	10	18

again deletion process continous. keep the bigger node at the end of the tree node. for that we have to swap the tree nodes.

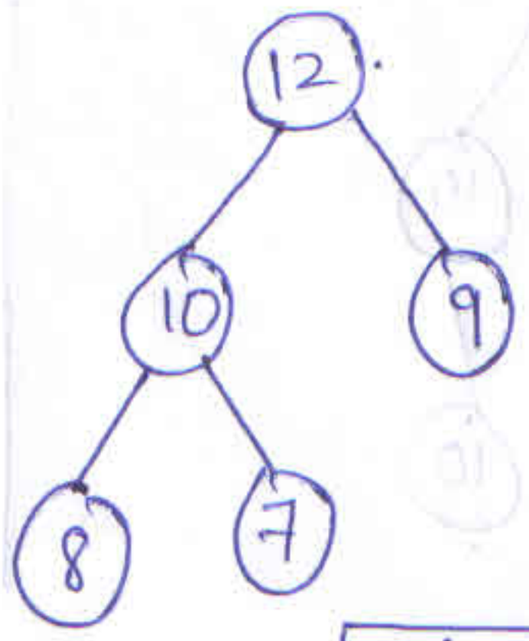


10	12	9	8	7	14	18

After deleting node 14 tree can as shown in below

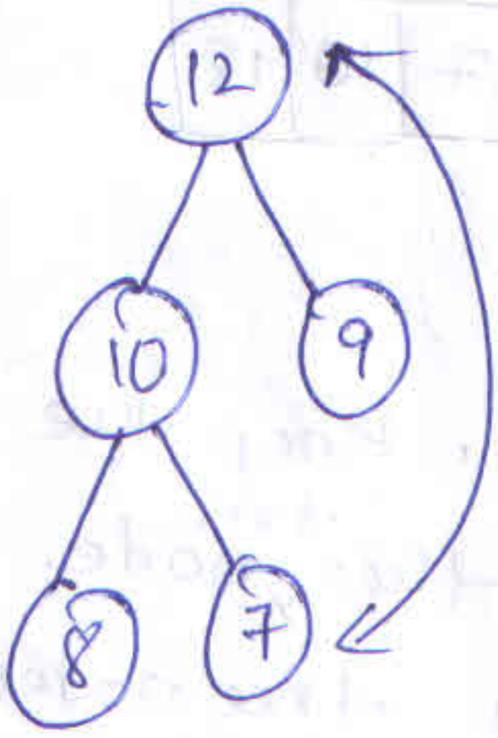


Heapify the tree after deletion

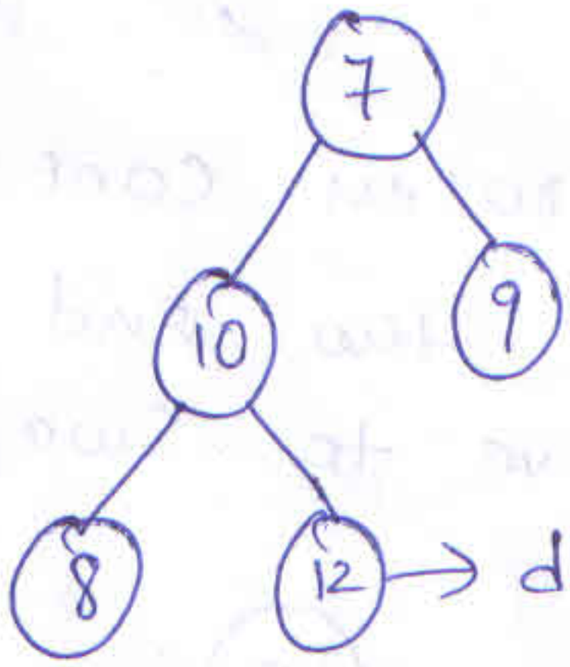


12	10	9	8	7	14	18

For deletion process we have to keep the Big node at the bottom of the tree.

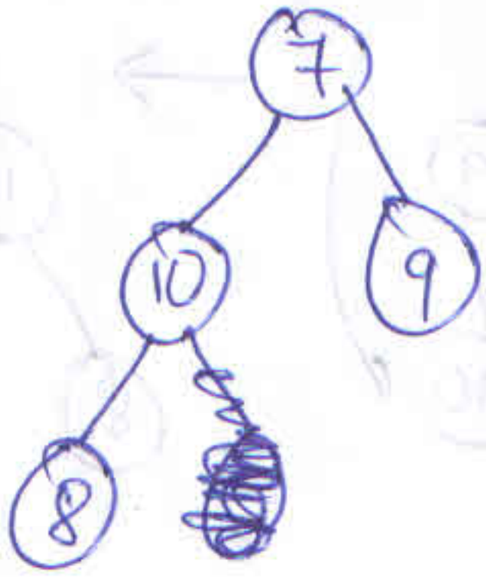


Swap 12 & 7 nodes

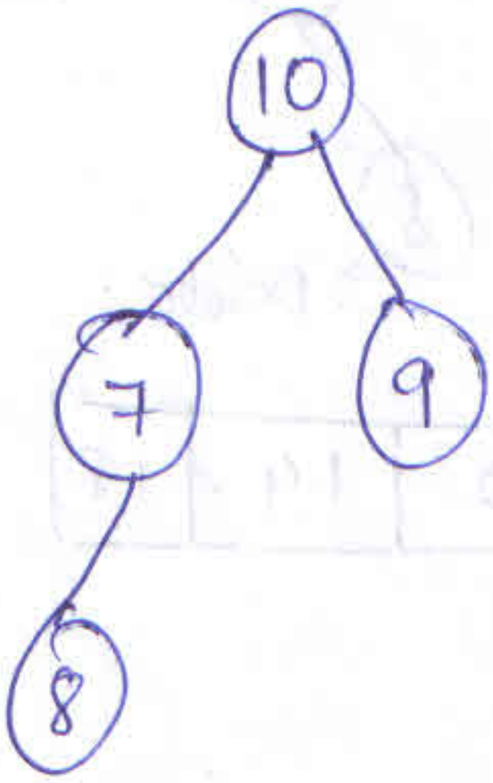


7	10	9	8	12	14	18

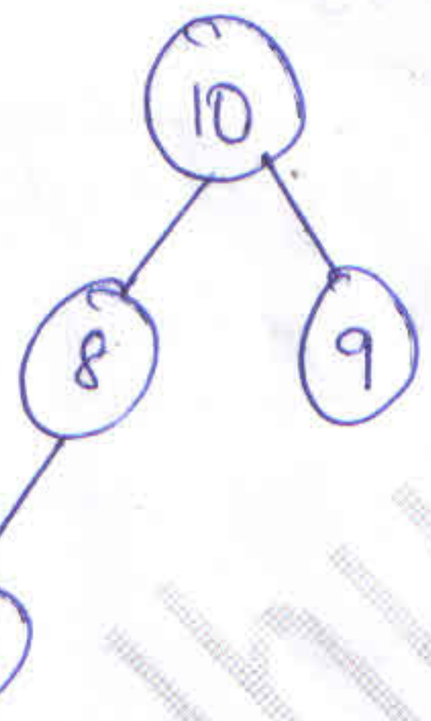
After deleting 12 node



Heapify the tree

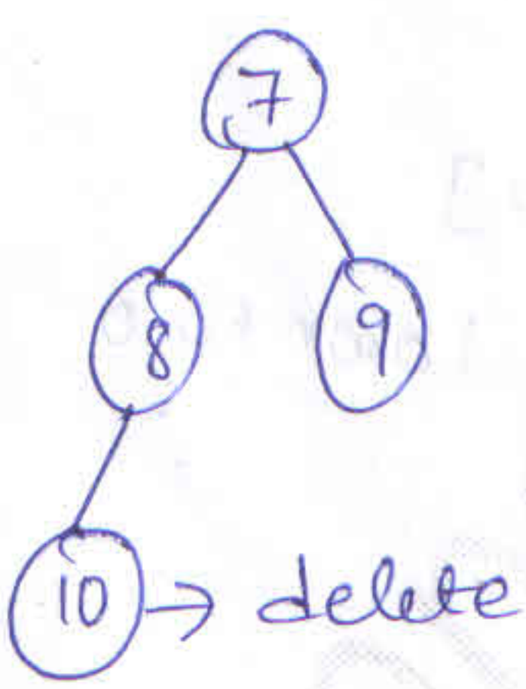


again Heapify →

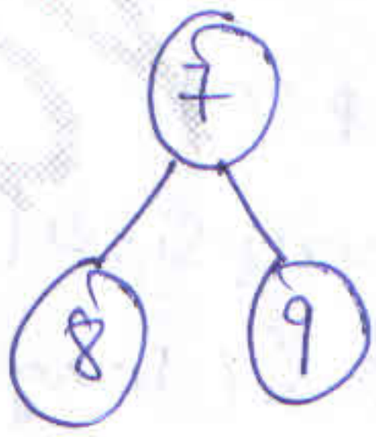


10	8	9	7	12	14	18

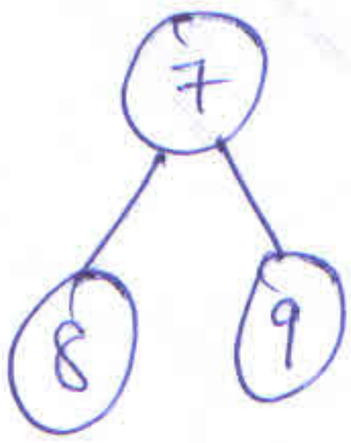
Swap 10 & 7 nodes



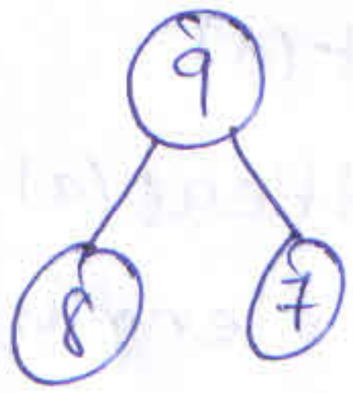
→



7	8	9	10	12	14	18



Heapify the tree →

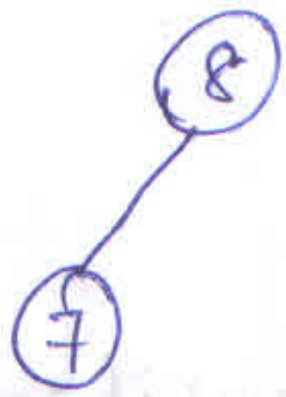


9	8	7	10	12	14	18

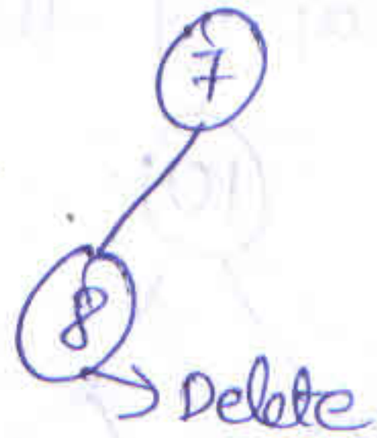
Swap 9 & 7 nodes

7	8	9	10	12	14	18

Heapify the tree



Swap 8 & 7



7	8	9	10	12	14	18

(7) → | 7 | 8 | 9 | 10 | 12 | 14 | 18 |

Algorithm for Build Heap :

```
BuildHeap (a)
1. Set heapsize[a] = length[a]
2. for i = (length[a] - 1)/2 down to 0
3, Heapify (a, i)
```

Algorithm for Heapsort :

```
Heapsort (a)
1, BuildHeap (a)
2, for i = length [a] - 1 down to 1
3, Swap (a[0], a[i])
4, Set heapsize [a] = heapsize [a] - 1
5, Heapify (a, 0)
```

Algorithm for Heapify :-

Heapify (a, i)
1, Set l = left (i)
2, Set r = Right (i)
3, if l ≤ heap-size [a] - 1 and a[l] > a[i]
4, then Set largest = l
5, Else set largest = i
6, If r ≤ heap-size (a] - 1 and a[r] > a [largest]
7, then Set largest = r
8, If largest ≠ i
9, then Swap (a[i], a [largest])
10, Heapify (a, largest)

→ Sorting by Exchange :-

1. Bubble sort
2. Quick sort

1, Bubble sort :- In this type of sorting proceeds by scanning the list from left to right and whenever a pair of adjacent key found out of order then those items are swapped this process repeats till all the elements of the lists are sorted order.

Bubble sort works as follows

pass 1 :- Compare a[0] & a[1] and arrange them in the desired order a[0] ≤ a[1] then Compare a[1] & a[2] and arrange them a[1] ≤ a[2] continue the process until a[n-2] & a[n-1] and arrange them so that 'n-1' comparisons are done during this process largest elements are bubbledup to the $(n-1)^{th}$ position.

pass 2 :- Repeat the process with one less comparison i.e we stop after we compare and possible arrangments a[n-3] & a[n-2] so that 'n-2' comparisons and second largest no will place

⋮

pass n-1 :- Compare a[0] & a[1] and arrange them so that a[0] ≤ a[1]

Example :- Arrange the following list of elemements in sorted order using bobblesort

44, 55, 33, 88, 77, 22, 11, 66

place the elements in array as shown below

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
44	55	33	88	77	22	11	66

pass 1 :- 1, Compare a[0] & a[1] i.e, 44 < 55 True

2, Compare a[1] & a[2] i.e, 55 < 33 False

then (Interchange) swap the elements

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
44	33	55	88	77	22	11	66

3, Compare a[2] & a[3] i.e, 55 < 88 True

4, Compare a[3] & a[4] i.e, 88 < 77 False

then Swap 88 & 77

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
44	33	55	77	88	22	11	66

5, Compare a[4] & a[5] i.e 88 < 22 False

then Swap 88 & 22

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
44	33	55	77	22	88	11	66

6, Compare a[5] & a[6] i.e, 88 < 11 False

then Swap 88 & 11

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
44	33	55	77	22	11	88	66

7, Compare a[6] & a[7] i.e, 88<66 False

then Swap 88 & 66

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
44	33	55	77	22	11	66	88

Pass 2 :-

(33) (44) 55 77 22 11 66 88

33 44 55 77 22 11 66 88

33 44 55 (22) (77) 11 66 88

33 44 55 22 (11) (77) 66 88

33 44 55 22 11 (66) (77) 88

33 44 55 22 11 66 (77) (88)

33 44 55 22 11 66 77 88

pass 3 :-

33 44 (22) (55) 11 66 77 88

33 44 22 (11) (55) 66 77 88

pass 4 :-

33 (22) (44) 11 55 66 77 88

33 22 (11) (44) 55 66 77 88

pass 5 :- (22) (33) 11 44 55 66 77 88

22 (11) (33) 44 55 66 77 88

pass 6 :- (11) (12) 33 44 55 66 77 88

Algorithm for Bubble Sort (Ascending order)

```
Void BubbleSort (int a[], int n)
{
    int temp, pass, i;
    For (pass=0; pass<n-1; pass++)
    {
        For (i=0; i<n-pass-1; i++)
        {
            if (a[i] > a[i+1])
            {
                temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
            }
        }
    }
}
```

note :- Time complexity for BubbleSort is same for all Best, worst, Avg time complexity is $O(n^2)$

2) Quicksort :- Quick sort is a sorting algorithm that uses the divide and conquel satrategy In this method division is dynamically carried out.

there are 3 steps for quicksort

1, Divide
2, conquel
3, combine

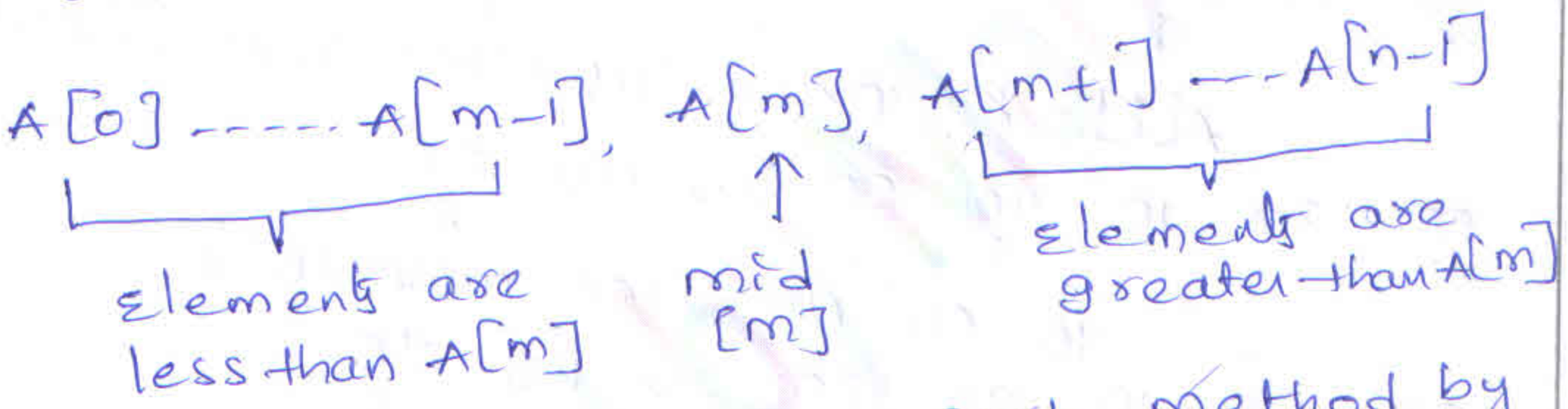
1, Divide :-

* split the array into 2 sub arrays. that each element in the left subarray is less than or equal to the middle element and each element in Right sub array is greater than the middle elemenunt
* Splitting of the array into 2 sub arrays is based on pivot elements.
* All the Elements that are less than pivot should be in left sub array and all the elements that are more than pivot should be in right subarray

2, Conquel :- Recursively sorts the 2 sub arrays.

Combine :- It combines all the sorted elements in a group to form a list of sorted elements

* merge sort divides the array based on the positions of array elements.
* Consider an array A[i], where 'i' is ranging from 0 to n-1
* division of array elements are as shown in below.



* Example :- Solve the Quicksort method by applying it to the following array of elements 50 30 10 90 80 20 40 70

Steps for solving the sort

1, Select first element of array A as pivot(*)
2, Select 'i' & 'j' to 1st & last elements of the array
3, Increment 'i', until A[i] < * and stop
4, decrement 'j', until A[j] > * and stop
5, If i < j swap A[i] & A[j]
6, Repeat above steps until i > j i.e when i and j cross each other.

7, Exchange the pivot Element with 'j' Element
8, Swapping keeps the bigger no's to the right and smaller no's to the left

50 30 10 90 80 20 40 70
low high
*, i j

50 30 10 90 80 20 40 70
* i j

A[i] < *, 30 < 50 Yes then increment 'i'

50 30 10 90 80 20 40 70
* i j

A[i] < *, 10 < 50 Yes then increment 'i'

50 30 10 90 80 20 40 70
* i j

90 < 50 NO then move to 'j'

50 30 10 90 80 20 40 70
* i j

A[j] > *, 70 > 50 Yes then decrement 'j'

50 30 10 90 80 20 40 70
* i j

A[j] > * 40 > 50 NO then swap 'i' & 'j'

50 30 10 40 80 20 90 70
* i j

A[i] < * 40 < 50 Yes then increment 'i'

50 30 10 40 80 20 90 70
* i j

A[i] < * 80 < 50 NO then move to 'j'

50 30 10 40 80 20 90 70
* i j

A[j] > * 90 > 50 Yes then decrement 'j'

50 30 10 40 80 20 90 70
* i j

A[j] > * 20 > 50 NO then Swap i & j

50 30 10 40 20 80 90 70
* i j

A[i] < *, 20 < 50 Yes then increment 'i'

50 30 10 40 20 80 90 70
* i, j

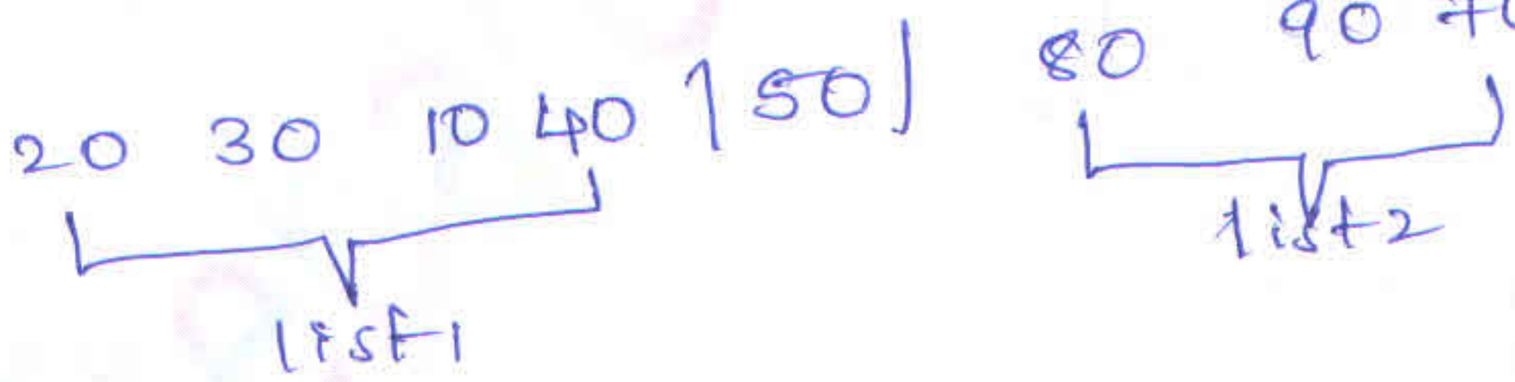
again check A[i] > *, 80 > 50 Yes decrement 'j'

50 30 10 40 20 80 90 70
* j i

when ever 'j' is crossed to 'i' then swap "j" with *

20 30 10 40 50 80 90 70

After swapping the 'j' value and * entire list will be divided to two parts and 50 is fixed for entire list. as shown in below



20 30 10 40
* i j

A[i] < *, 30 < 20 NO
then check 'j'

20 30 10 40
* i j

A[j] > *, 40 > 20 Yes
then decrement 'j'

20 30 10 40
* i j

10 > 20 NO A[j] > * 'j' can't be decrement and 'i' can't be
~~then decrement 'j'~~ increment then swap i & j

20 10 30 40
* i j

A[i] < *, 10 < 20 Yes increment 'i'

20 10 30 40
* i,j

A[j] > *, 30 > 20 Yes decrement 'j'

20 10 30 40
* j i

when ever 'j' cross 'i' then swap j with *

10 | 20 | 30 40 list is divided into two parts
↓ ↓
list1 list2

80 90 70
* i j

A[i] < *, 90 < 80 NO then check 'j' A[j] > *
70 > 80 NO then ~~decrement~~ then swap i & j

80 70 90
* i j

A[i] < *, 70 < 80 Yes increment 'i'

80 70 90
* i,j

A[j] > *, 90 > 80 Yes decrement 'j'

80 70 90
* i j

when ever 'j' cross 'i' then swap * with 'j' and entire list will be divided to two part

70 | 80 | 90

10 20 30 40 50 70 80 90

Algorithm for Quick Sort :- (Here low=i, high=j)

Quick (A[0 ---- n-1], low, high)

if (low < high) then

m ← partition (A[low --- high])

Quick (A[low --- m-1])

Quick (A[midtl --- high])

partition Algorithm for Quick Sort

partition (A[low --- high])

pivot ← A[low]

i ← low

j ← high+1

where (i <= j) do

{

while (A[i] <= pivot) do

i ← i+1

```
while (A[j] >= pivot) do
        j ← j-1;
    if (i<=j) then
        swap (A[i], A[j])
}
swap (A[low], A[j])
return j
}
```

Time complexity :-

- Average case :- $n \log n$

Best case :- $n \log n$

Worst case :- n^2

Note :- Quick Sort is Faster Sorting

→ Sorting by Distribution :-

Radix Sort :- This sort is based on the values of the actual digits in the position representation of the no's being sorted.

Eg :- number 367 in decimal notation is written with a 3 in hundreds position, 6 in tens position and 7 in one's position

The larger of the 2 such integers of equal length can be determined as follows :-

Start at the MSD (most significant Digit) and LSD (Least significant Digit)

The no with the larger digit in the first position.

Example for Radix Sort :-

Consider the unsorted array of 8 elements

45, 37, 05, 09, 06, 11, 18, 27

Now sort the elements according to the last digit.

Last Digit	0	1	2	3	4	5	6	7	8	9
Elements		11				05 45	06	37 27	18	09

Now Sort this number

last digit	Element
0	
1	11
2	
3	
4	
5	05, 45
6	06
7	27, 37
8	18
9	09

Now Sort the above array with the help of second last digit

2nd last digit	Element
0	05, 06, 09
1	11, 18
2	27
3	37
4	45
5	

~~2nd last digit~~ 2nd last digit	~~Element~~ Element
6	
7	
8	
9	

Algorithm for Radix Sort :-

Radix (a, n)

1. Set large = largest Element in the array
2. Set num = total no. of digits in the large Element
3. Set digit = num
4. Set pass = 1
5. Repeat Steps 6 to 12 while pass ≤ num
6. Initialize buckets
7. Set i = 0
8. Repeat Steps 9 to 11 while i ≤ n-1
9. Set l = pass - 1 position of number a[i]
   [0th position of number 123 is 3]
10. Put the number a[i] into bucket l
11. Set i = i+1 [End of Step 8 loop]
12. Set pass = pass+1 [End of Step 5 loop]
13. write all the no's from the bucket in order
14. Exit

Time complexity for Radix Sort :-

Average case :- nlogn

Best case :- nlogn

Worst case :- nlogn

Example :- Arrange the following elements in sorted order using radix sort

151, 60, 875, 342, 12, 477, 689, 128, 15

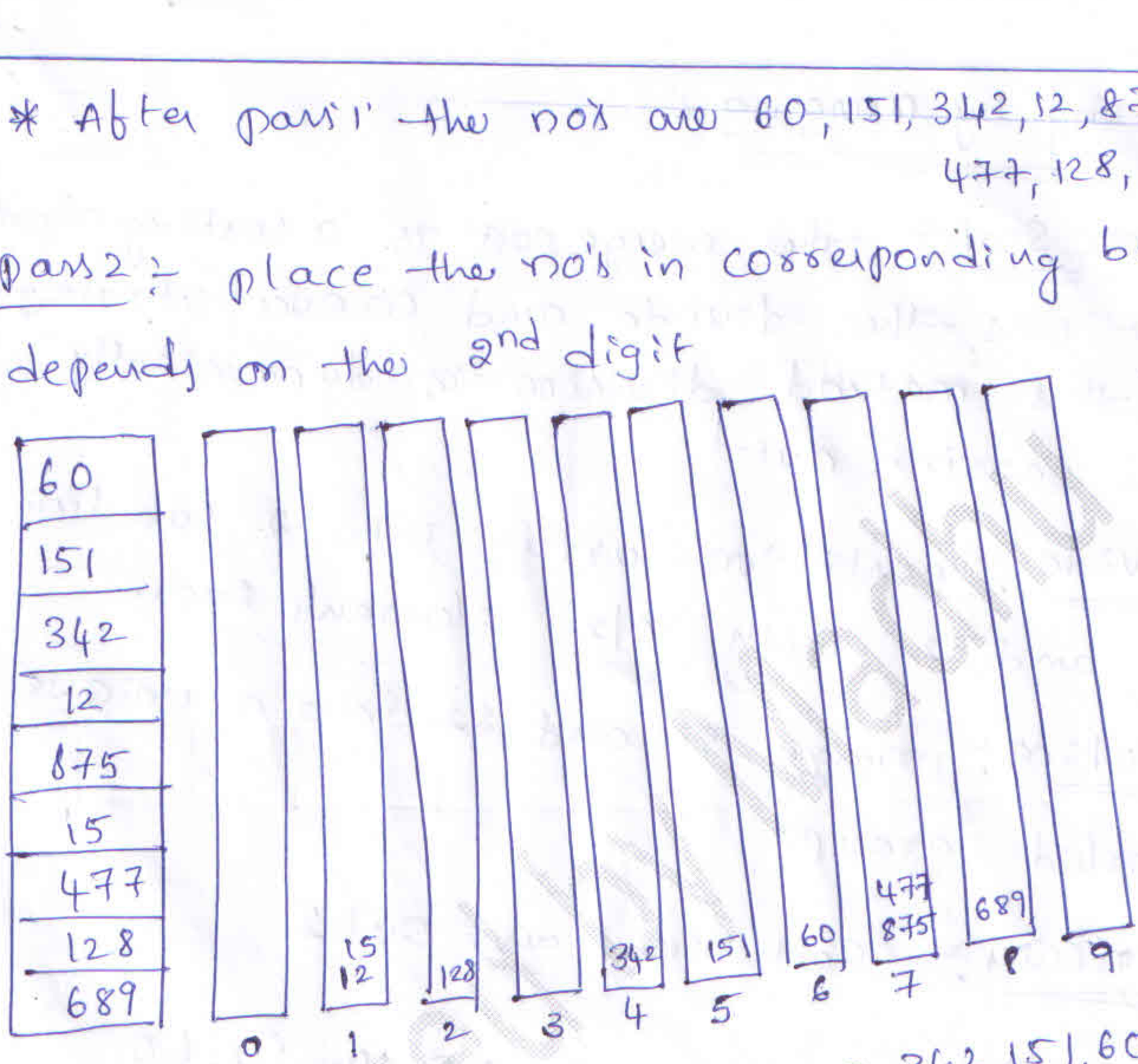
* To sort decimal no's we need ten buckets
* buckets are numbered as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
* put the above elements in array as shown in below

151	60	875	342	12	477	689	128	15

* In the above list the large no is 875 and it has 3 digits so it requires 3 passes to complete the sorting process.
* In step 1 we place the numbers in corresponding buckets depends on the LSD if LSD is '5' then put the number in pass 1: bucket '5' (there are 0 to 9 buckets are there)

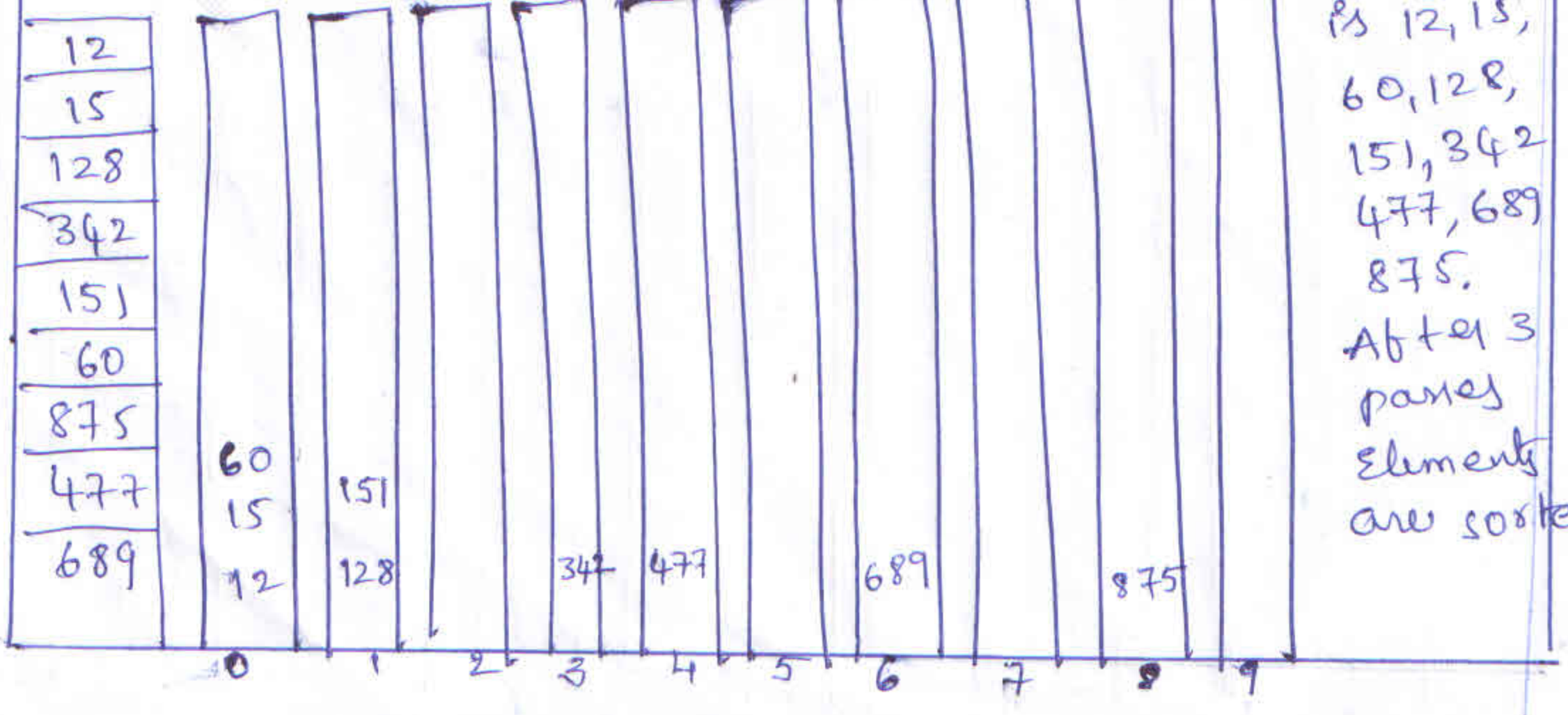
* After pass 1 the no's are 60, 151, 342, 12, 875, 15, 477, 128, 689

Pass 2 :- place the no's in corresponding buckets depends on the 2nd digit



* o/p of 2nd pass is 12, 15, 128, 342, 151, 60, 875, 477 and 689

* Pass 3:- place the number in corresponding buckets depends on the MSD



o/p of pass 3 is 12, 15, 60, 128, 151, 342, 477, 689, 875. After 3 passes Elements are sorted.

→ Sorting by manging :-

Merge sort :- the merge sort is a sorting algorithm that uses the divide and conquer strategy In this method division is dynamically carried out

Divide :- partition array into 2 sub lists S1 and S2 with n/2 elements each.

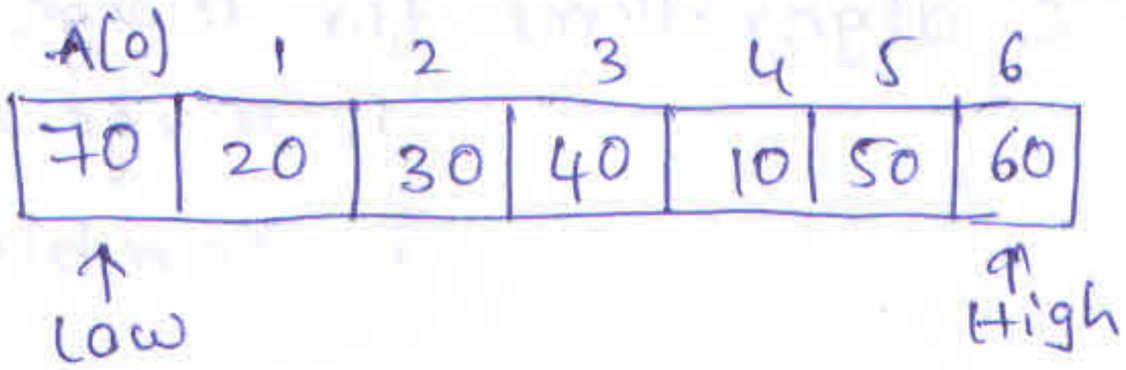
Combine :- merge S1 and S2 into a unique sorted group.

Conquer :- Sort sub1 and sub2

Examples :- 1) 70, 20, 30, 40, 10, 50, 60.

2) 10, 5, 7, 6, 1, 4, 8, 3, 2, 9.

3) E, X, A m, P, L, E

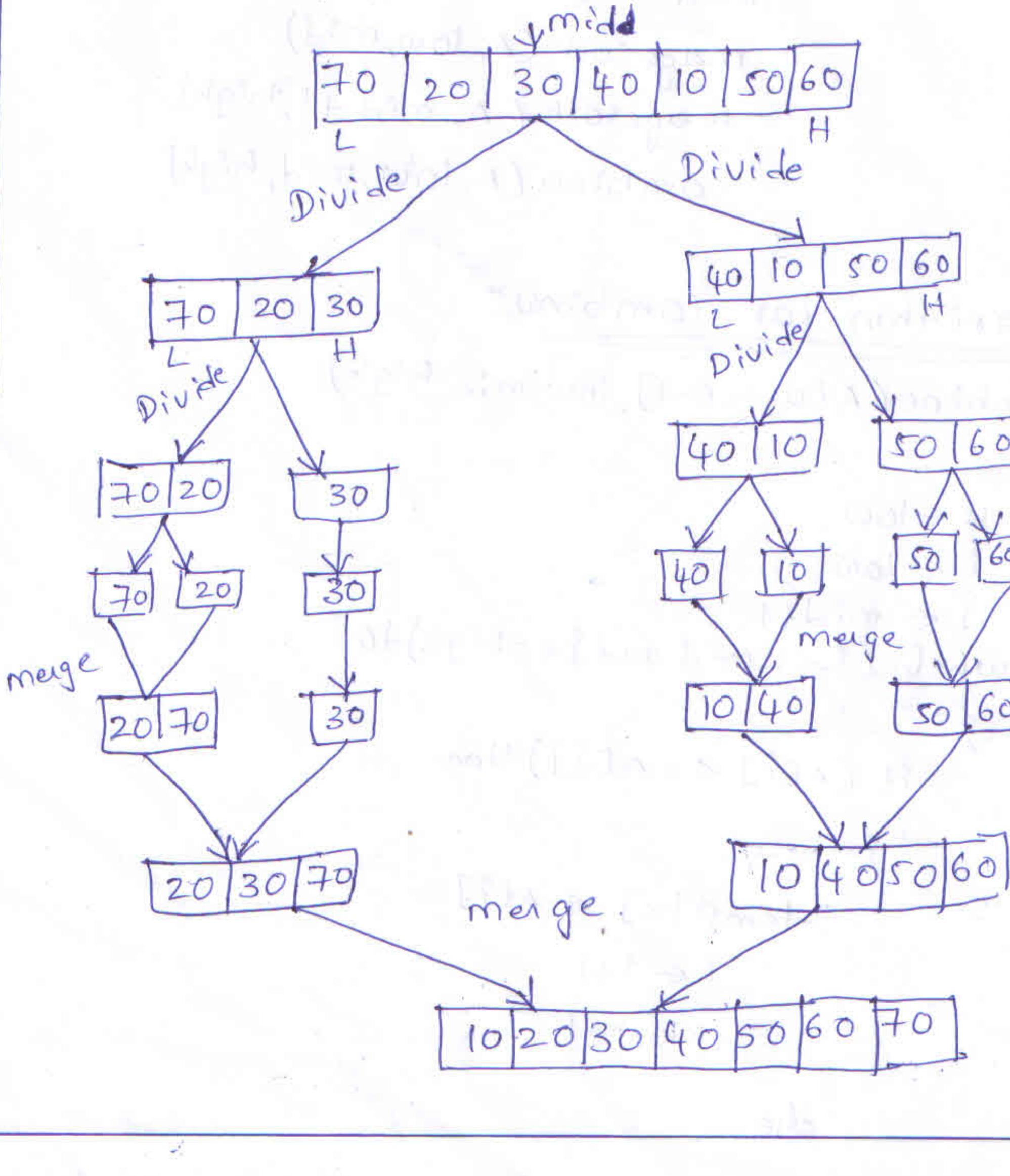


* Foo Identify the middle value then we have to find mid

$$mid = (low + High)/2$$

Here low = 70, High = 60 $m = \frac{0+6}{2} = \underline{3} \rightarrow A[3]$ mid

A[0] A[6]



* There are 2 algorithms for merge sort
  i) merge sort (sortingelem)
  ii) Combine.

Algorithm for Sorting :-

```
mergesort(int A[0---n-1], low, high)
    if (low < high) then
    {
        mid ← (low+high)/2
        mergesort (A, low, mid)
        mergesort (A, mid+1, high)
        Combine (A, low, mid, high)
    }
```

Algorithm for combine :-

```
Combine (A[0---n-1], low, mid, high)
 {
    k ← low
    i ← low
    j ← mid+1
   while (i<=mid and j<=high) do
    {
   if (A[i] <= A[j]) then
    {
      temp[k] ← A[i]
        i ← i+1
        k ← k+1
    }
   else
```

```
{
    temp[k] ← A[j]
      j ← j+1
      k ← k+1
  }
}
while (i <= mid) do
  {
    temp[k] ← A[i]
      i ← i+1
      k ← k+1
  }
while (j <= high) do
  {
    temp[k] ← A[j]
      j ← j+1
      k ← k+1
  }
```

Time complexity for merge sort :-

Avrage case :- n log n

Best case :- n log n

worst case :- n log n

Write a program for heapsort using heap construction

```
#include <stdio.h>
#include <conio.h>

int parent(int);
int left(int);
int right(int);
void heapify(int[], int, int);
void buildheap(int[], int);
void heapsort(int[], int);
void main()
{
  int x[20], n, i;
  clrscr();
  pf("\n Enter the numbers to be sorted:");
  sf("%d", &n);
  pf("\n Enter %d elements \n", n);
  for(i=0; i<n; i++)
  sf("%d", &x[i]);
  heapsort(x, n);
  pf("\n the sorted array is \n");
  for(i=0; i<n; i++)
  pf("\t %d", x[i]);
  getch();
}
```

```
int parent (int i)
{
   return (i/2);
}
int left (int i)
{
  return (2*i+1);
}
int right(int i)
{
  return (2*i+2);
}
Void heapify (int a[], int i, intn)
{
 int l, r, large, temp;
   l = left (i);
   r = right (i);
 if ((l <n-1) && (a[l] > a[i]))
 large=l;
 else
 large=i;
 if ((r<=n-1) && (a[r] > a[large]))
```

```
large = r;
if (large != i)
{
  temp = a[i];
  a[i] = a[large];
  a[large] = temp;
  heapify(a, large, n);
}
}
Void buildheap(int a[], int n)
{
 int i;
 for (i = (n-1)/2; i>=0; i--)
 heapify(a, i, n);
}
Void heapsort (int a[], int n)
{
 int i, m, temp;
 buildheap (a, n);
  m = n;
 for (i = n-1; i >= 1; i --)
 {
   temp = a[0];
```

```
a[0] = a[i];
a[i] = temp;
m = m-1
heapify (a, 0, m);
}
```

Out put :-

```
Enter two numbers to be sorted : 5
Enter 5 Elements
26
16
98
564
87
the sorted array is
16 26 87 98 564
```

write a program for sorting the elements by bubble sort method.

```
#include <stdio.h>
#include <conio.h>
void bubblesort (int[], int);
void main()
{
 int x[20], n, i;
 clrscr();
 pf ("\n Enter the numbers to be sorted:");
 sf("%d", &n);
 pf ("\n Enter %d elements \n", n);
 for (i=0; i<n; i++)
 sf ("%d", &x[i]);
 bubblesort (x, n);
 pf ("\n the sorted array is \n");
 for (i=0; i<n; i++)
 pf ("%4d", x[i]);
 getch();
}
void bubblesort (int a[], int n)
{
```

```
int temp, pass, i;
  for (pass = 0; pass < n-1; pass++)
  {
    for (i=0; i<n-pass-1; i++)
    {
      if (a[i] > a[i+1])
      {
        temp = a[i];
        a[i] = a[i+1];
        a[i+1] = temp;
      }
    }
  }
```

Output: Enter the numbers to be sorted:
5
Enter 5 elements
16
23
84
5
12
The sorted array is
5 12 16 23 84

write a program for quicksort by using ascending order.

```
#include <stdio.h>
void swap (int a[], int left, int right)
{
int temp;
clrscr();
 temp = a[left];
 a[left] = a[right];
 a[right] = temp;
} // end swap.
void Quicksort (int a[], int low, int high)
 {
   int pivot;
   // termination condition
   if (high > low)
   {
    pivot = partition (a, low, high);
    Quicksort (a, low, pivot-1);
    Quicksort (a, pivot+1, high);
   }
} // End Quicksort
```

```
int partition (int a[], int low, int high)
{
    int left, right;
    int pivot_item;
    int pivot = left = low;
    pivot_item = a[low];
    right = high;
    while (left < right)
    {
// move left while item < pivot
    while (a[left] <= pivot_item)
      left++;
// move right while item > pivot
    while (a[right] > pivot_item)
      right--;
    if (left < right)
      swap (a, left, right);
    }
//right is final position for the pivot
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
```

```
        getch();
}// End partition

//void Quicksort (int a[], int, int);
void printarray (int a[], int);

int main()
{
    int a[50], i, n;
    Pf("\n Enter no. of elements : ");
    Sf("%d", &n);
    Pf("\n Enter the elements : \n");
    for (i=0; i<n; i++)
    Sf("%d", &a[i]);
    Pf("\n Unsorted elements : \n");
    printarray (a, n);
    Quicksort(a, 0, n-1);
    Pf("\n Sorted elements : \n");
    printaarray(a, n);
} // End main

void printarray (int a[], int n)
{
    int i;
```

```
for (i=0; i<n; i++)
    pf ("%d", a[i]);
    pf ("\n");
}// end printarray
```

Output:-

```
Enter no.of elements: 5
Enter the elements: 12
                    6
                    326
                    15
                    99
unsorted elements:
    12   6   326   15   99
Sorted elements:
    6   12   15   99   326
```

a program for sorting the elements by radix sort.

```
#include <stdio.h>
#include <conio.h>
#include< math.h>
void main()
{
int a[100][100], r=0, c=0, i, sz, b[50], temp;
clrscr();
Pf("\n size of array:");
Sf("%d", &sz);
Pf("\n");
for(r=0; r<100; r++)
{
for(c=0; c<100; c++)
a[r][c] =1000;
}
for(i=0; i<sz; i++)
{
Pf("Enter elements %d:", i+1);
Sf("%d", &b[i]);
r=b[i]/100;
c=b[i]%100;
```

```
a[r][c] = b[i];
}
for (r=0; r<100; r++)
{
 for (c=0; C<100; c++)
 {
  for (i=0; i<sz; i++)
  {
   if (a[r][c] == b[i])
   {
    pf ("\n%d", a[r][c]);
   }
  }
 }
 getch();
}
```

Output :- Size of array : 5

Enter Elements 1 : 236	Enter Elements 5 : 66
Enter Elements 2 : 15	8
Enter Elements 3 : 69	15
Enter Elements 4 : 8	66
Enter Elements 5 : 66	69
	236