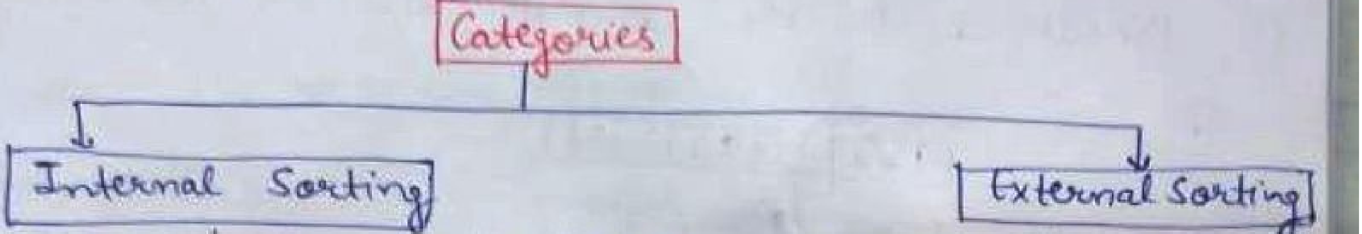
Sorting :→ The arrangement of data in a preferred order is called sorting. A sorting algorithm is used to rearrange a given array or list of elements acc. to a comparison operator on the elements.

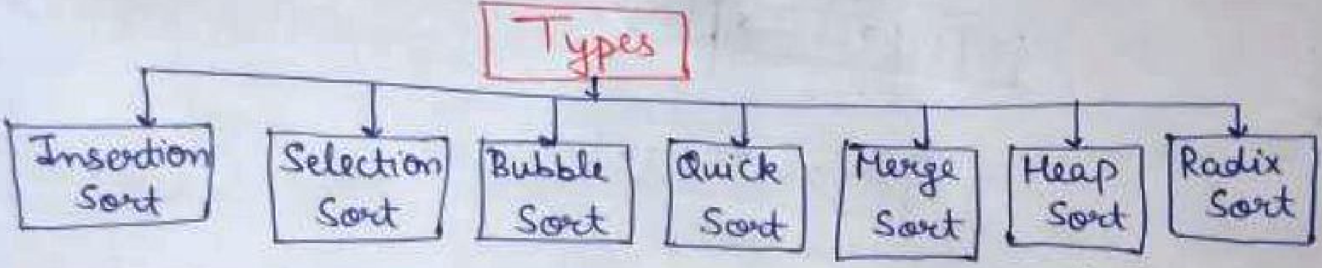


Categories

Internal Sorting

External Sorting

- If the input data is adjusted in main memory, then it is internal sorting.
- If the input data is not adjusted in main memory & it stored in other storage device, then it is external sorting.



Types

Insertion Sort

Selection Sort

Bubble Sort

Quick Sort

Merge Sort

Heap Sort

Radix Sort

1. Insertion Sort :→ It is a sorting algorithm where the array is sorted by taking one element at a time. The principle behind insertion sort is to take one element, iterate through the sorted array & find its correct position in the sorted array.

Algorithm :-

a) If the element is the first one, consider it as already sorted.

b) Move to next element.

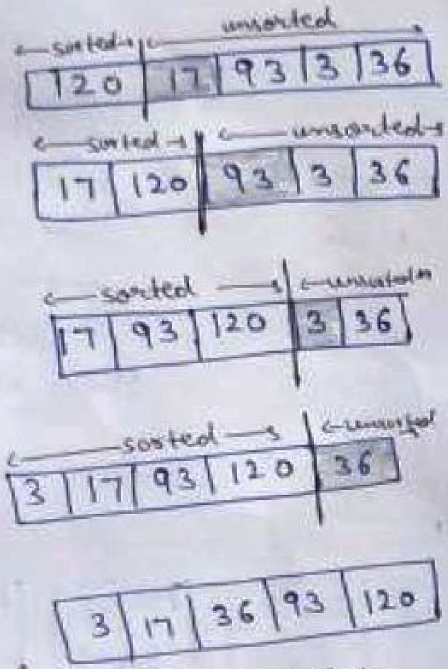
c) Compare the current element with all elements in the sorted array.

d) If the element in the sorted array is smaller than the current element, iterate to the next element. Otherwise Shift all the greater element in the array by one position towards the right.

e) Insert the value at the correct position.

f) Repeat until the complete list is sorted.

Eg:-



sorted array

```
for( i=1 ; i<n ; i++ )
{
        temp = a[i];
          j = i-1;
    while( j >= 0 && a[j] > temp )
    {
            a[j+1] = a[j];
              j-- ;
    }
          a[j+1] = temp;
}
```

Advantages :-

⊙ Simple.

⊙ Efficient.

Disadvantages :→

- Inefficient for large data set.
- It doesnot perform well for unsorted data.

2. **Selection Sort** :→ It sorts an array by repeatedly finding the minimum element from the unsorted part and putting it at the beginning.

Algorithm :-

a) Set min to the first location.
b) Search the minimum element in the array.
c) Swap the first location with the minimum value in the array.
d) Assign the second element as min.
e) Repeat the process until we get a sorted array.

Eg:- n = 7

12	29	25	8	32	17	40

Pass I :- Find the min. element i.e. 8 swap 12 with 8.

←sorted→ | ← unsorted →

8	29	25	12	32	17	40

Pass II :- Swap 29 with 12.

←sorted→ | ← unsorted →

8	12	25	29	32	17	40

Pass III :- Swap 25 with 17.

← sorted → | ← unsorted →

8	12	17	29	32	25	40

Pass IV :- Swap 29 with 25.

← sorted → | ← unsorted →

8	12	17	25	32	29	40

Pass V :-

← sorted → ← unsorted

8	12	17	25	29	32	40

Pass VI :-

8	12	17	25	29	32	40

sorted array

Pass = n-1

```
for (i=0; i<n-1; i++)
{
        pos = i;
        for(j= i+1; j<n; j++)
        {
            if(a[pos] > a[j])
                pos = j;
        }
        if (pos ! = i)
        {
            c = a[i];
            a[i] = a[pos];
            a [pos] = c;
        }
}
```

Advantages :-

- Performs well on small list.
- No additional storage is required.

Disadvantages :-

- Less efficient on large data.
- It requires n^2 number of steps for sorting n elements.

3. Bubble Sort :- It simply compares the current element with the next element and swaps it,

if it is greater or less depending on the condition. Each time an element is compared with all other elements till its final place is found is called a pass.

Algorithm :-

a) Run a nested for loop to traverse the input array using two variables i and j, such that $0 \le i \le n-1$ and $0 \le j < n-i-1$.

b) If arr[j] is greater than arr[j+1] then swap these adjacent elements, else move on.

c) Print the sorted array.

For eg:-

13	32	26	35	10

First Pass

- Compare 13 with 32. (32>13). already sorted.

13	32	26	35	10

- Compare 32 with 26. (32>26). Swap 32 with 26.

13	26	32	35	10

- Compare 32 with 35. No swapping.

13	26	32	35	10

- Compare 35 with 10. Swap

13	26	32	10	35

Second pass :- • Compare 13 with 26. No swapping

13	26	32	10	35

- Compare 26 with 32 . No Swapping

13	26	32	10	35

- Compare 32 with 10 . Swap

13	26	10	32	35

- Compare 32 with 35. No Swapping

13	26	10	32	35

Third Pass :-

- Compare 13 with 26. No Swapping

13	26	10	32	35

- Compare 26 with 10. Swap

13	10	26	32	35

- Compare 26 with 32. No swapping

13	10	26	32	35

- Compare 32 with 35. No Swapping

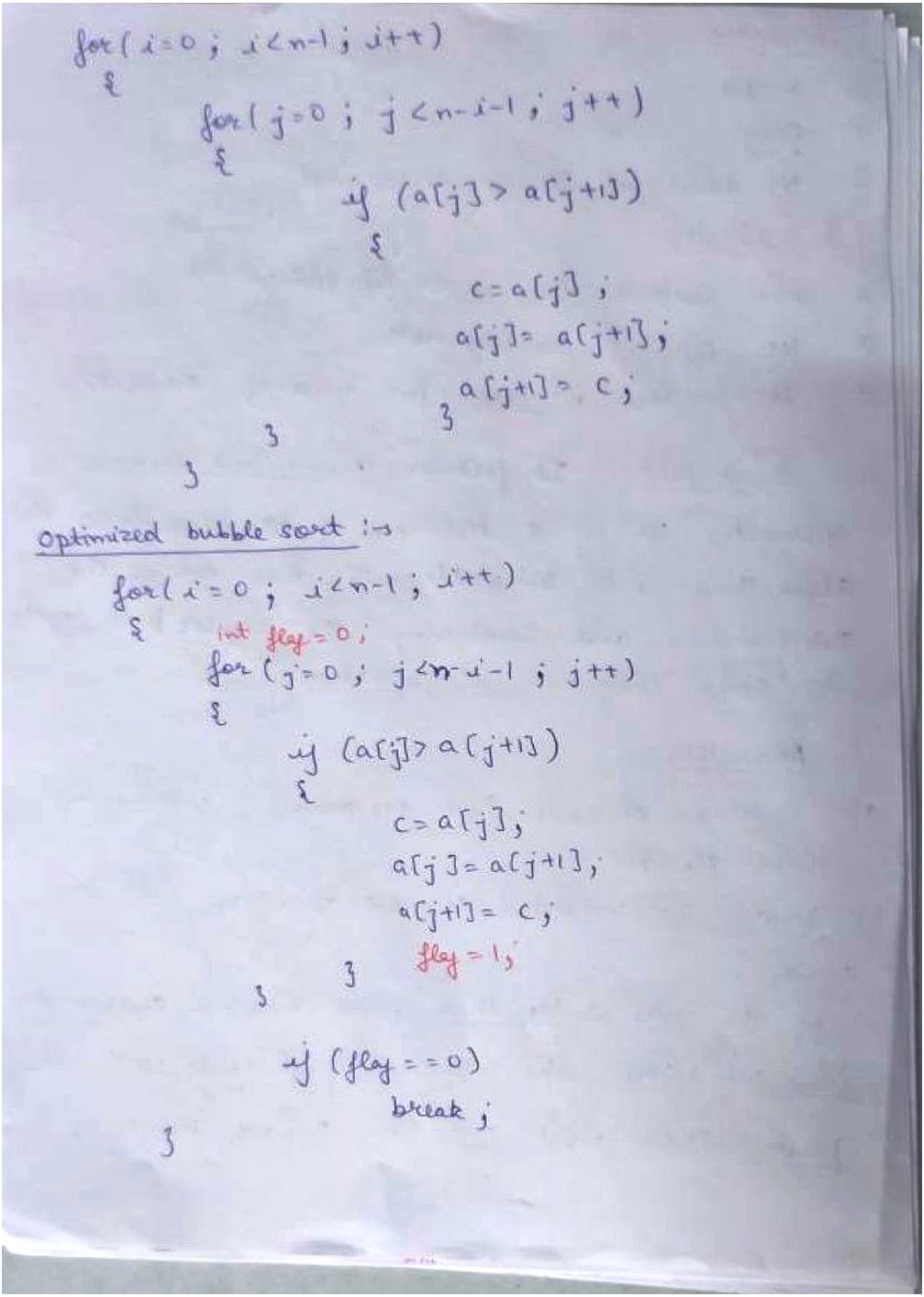
13	10	26	32	35

Fourth Pass :-

- Compare 13 with 10. Swap

10	13	26	32	35

There is no swapping required, so the array is completely sorted.



```
for( i=0 ; i<n-1 ; i++)
 {
        for( j=0 ; j<n-i-1 ; j++)
         {
               if (a[j] > a[j+1])
                {
                      c=a[j] ;
                     a[j]= a[j+1];
                     a[j+1]= c;
                }
         }
 }
```

Optimized bubble sort :->

```
for( i=0 ; i<n-1 ; i++)
 {      int flag=0;
      for (j=0 ; j<n-i-1 ; j++)
       {
            if (a[j]> a[j+1])
             {
                   c=a[j];
                   a[j]=a[j+1];
                   a[j+1]= c;
                    flag=1;
             }
       }

            if (flag==0)
                  break;
 }
```

Advantages :-

⊙ Simple .

⊙ Easy .

⊙ No additional storage is required.

Disadvantages :-

⊙ Not suitable for real-life applications.

⊙ less efficient on large data .

⊙ It requires n^2 steps for n no. of elements .

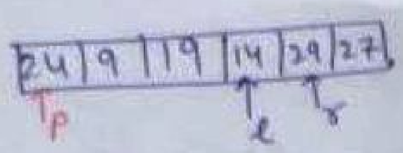
4. Quick Sort :- It follows divide and conquer approach. It is a technique of breaking down the algorithms into sub problems , then solving the sub-problems and combining the results back together to solve the original problem.

Algorithm :-

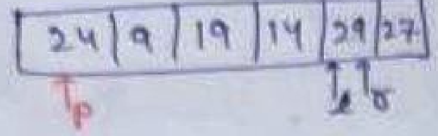
a) Pick an element from an array , call it as pivot element .

b) Divide an unsorted element of array into two arrays .

c) If the value is less than pivot element come under first sub-array , the remaining elements with value greater than pivot come in second subarray .

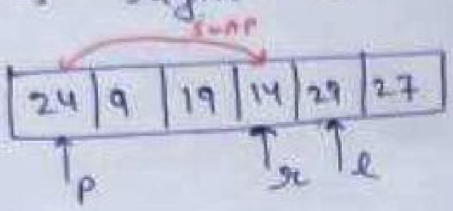


⊙ 24>14 → left ++

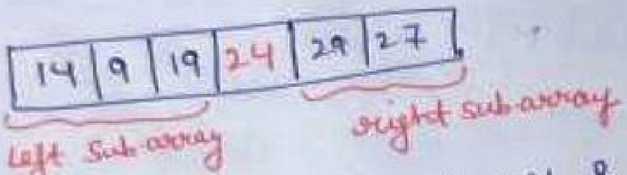


⊙ 24 ≯29 → stop, check for right value.

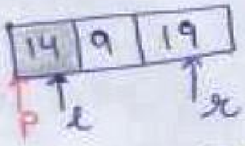
24<29 → right --



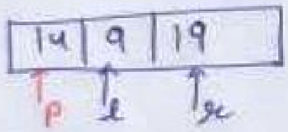
24>14 && l>r ⇒ swap right with pivot.



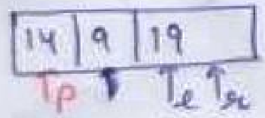
Apply quick sort again at left & right sub array.



⊙ 14 ≥14 → left ++

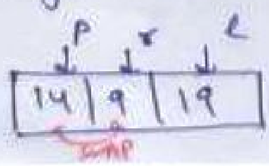


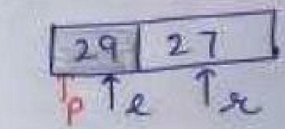
⊙ 14>9 → l++



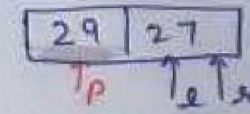
⊙ 14≯19 → check for right value

14<19 → right --



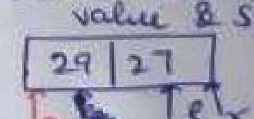


⊙ 29≥29 → l++



⊙ 29>27 → No value
check for right value

29>27 → ~~check~~ for left value & swap with pivot



*

→ a[pivot] ≥ a[left] → left ++

→ a[pivot] < a[left] → stop, Check for right values.

*

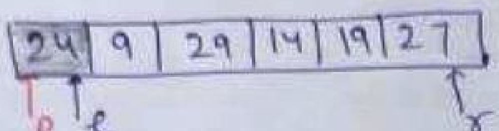
→ a[pivot] < a[right] → right --

→ a[pivot] > a[right] → stop, check for left values & Swap

*

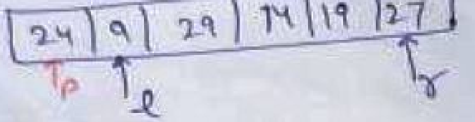
If left > right && → swap right with pivot value.
a[pivot] > a[right]

Eg:-



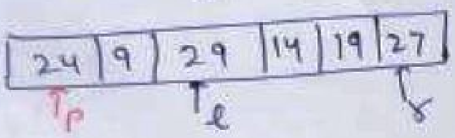
24	9	29	14	19	27

⊙ 24 ≥ 24 → left ++



24	9	29	14	19	27

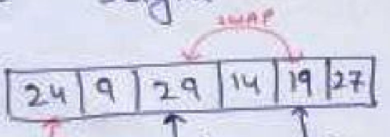
⊙ 24 > 9 → left ++



24	9	29	14	19	27

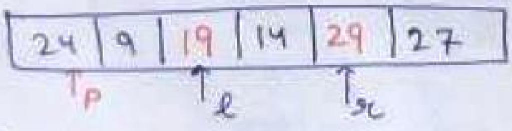
⊙ 24 ≯ 29 → Stop & check for right value. i.e. 27

24 < 27 → right --



24	9	29	14	19	27

⊙ 24 > 19 → stop, check for left value & Swap.
i.e. Swap 19 with 29.



24	9	19	14	29	27

⊙ 24 > 19 → left ++

⊙ 14>9 → check for left value (l>r)
Now, swap 14 with 9

9	14	19

⊙ 29>27 → && (l>r)
Now, swap 29 with 27.

27	29

9	14	19	24	27	29

sorted array

```
quicksort ( int a[10], l, r )
                 int i, j, pivot, c, temp;
{
    if ( l < r )
    {
        pivot = l;
         i = l ;
         j = r ;
        while (i<j)
        {
           while ( a[i] <= a[pivot] && i<r)
                      i++ ;
           while ( a[j] > a[pivot])
                      j-- ;
              if (i<j)                          // Swapping of left
              {                                       & right
                   c = a[i];
                  a[i] = a[j];
                   a[j] = C ;
              }
         }
             temp = a[pivot];                   // Swapping of
             a[pivot] = a[j] ;                     right with
              a[j]   = temp ;                        pivot
           quicksort (a , l , j-1) ;
            quicksort (a , j+1 , r);
     }
}
```

Advantages :-

- ⊙ It works rapidly and effectively.
- ⊙ It doesnot repuire any additional memory.
- ⊙ It is widely used method of sorting.

Disadvantages :-

- ⊙ Complex.
- ⊙ In worst case, time efficiency is very poor.
- ⊙ Unstable (won't preserve the element order).

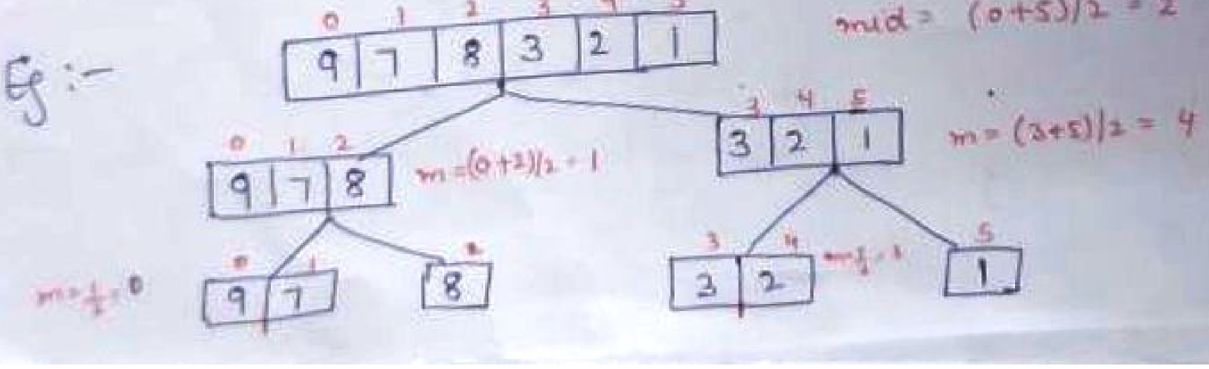
5. Merge Sort :→

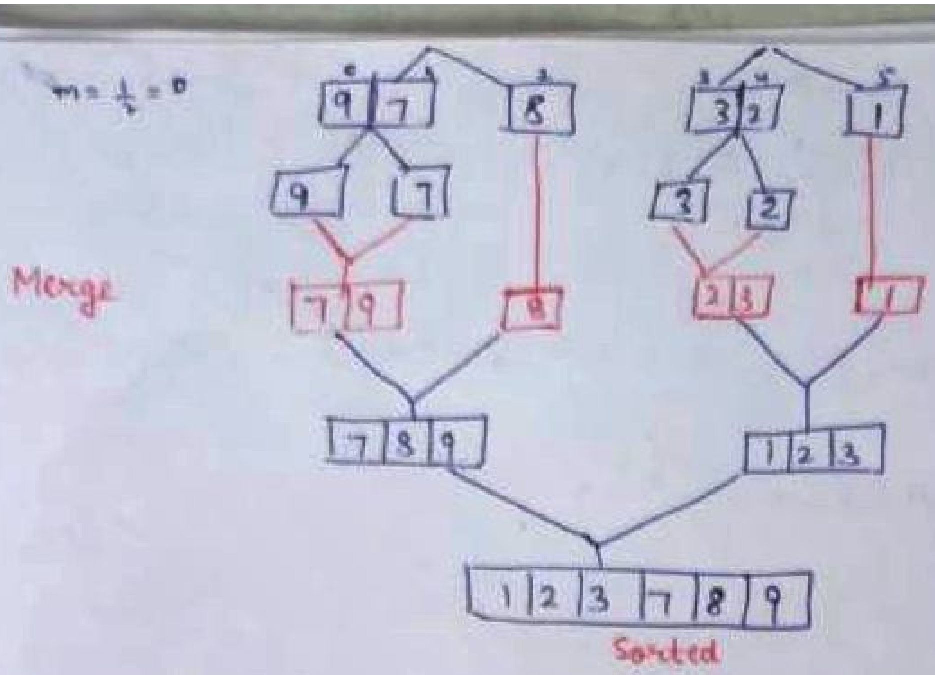
This follows divide and conquer approach to sort the element. It divides the given list into two halves until the list cannot be divided further and then merge them into sorted way.

Algorithm :-

1) Divide the unsorted list into N sublists, each containing 1 element.
2) Take adjacent pairs of two lists & merge them to form a list.
3) Repeat the process till a single sorted list is obtained.

Eg :-





```
void merge (int a[], int l, int m, int r)
 {
          int  i; j, k, b[50];
            i = l;
            j = m+1;
            k = l;
     while (i <= m  && j <= r)
      {
             if (a[i] < = a[j])
              {
                   b[k] = a[i];
                     i++;
                }
             else
               {
                   b[k] = a[j];
                     j++;
                }

              k++;
      }

      while (i <= m)
       {
             b[k] = a[i];
               i++;
               k++;
       }
```

```
    while (j <= r)
      {
              b[k] = a[j];
                j++;
        }       k++;
  for( int k= l; k<=r ; k++)
      {
            a[k]= b[k];
      }
 }
```

Advantages:-

- It is quicker for larger lists because it doesnot go through the whole list several times.
- It never changes the order of previous data.
- It is good for external sorting.

Disadvantages:-

- It uses extra space in sorting.
- Not suitable for small problems.

6. Heap Sort :→

Heap is a complete binary tree (in which the node can have utmost two Children).

Heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

Algorithm :- There are two phases involved:-

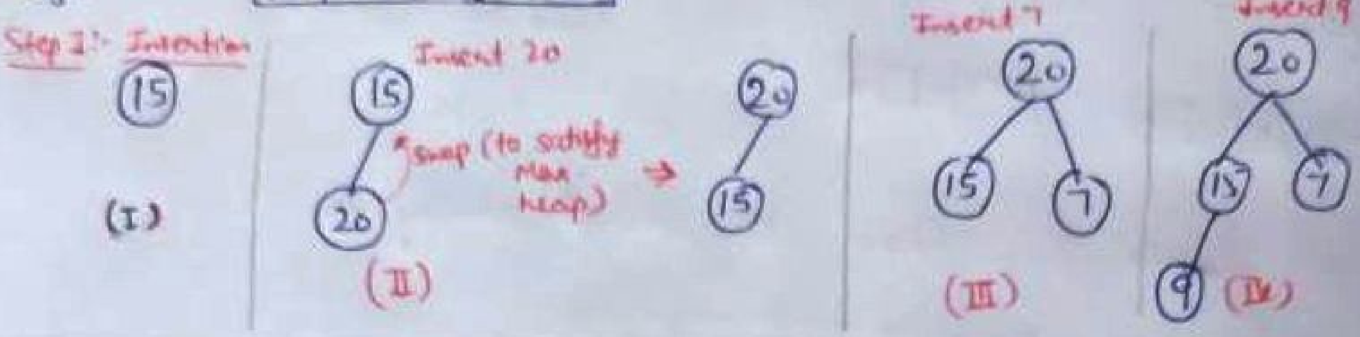
i) The first step includes the creation of a heap by adjusting the elements of the array.

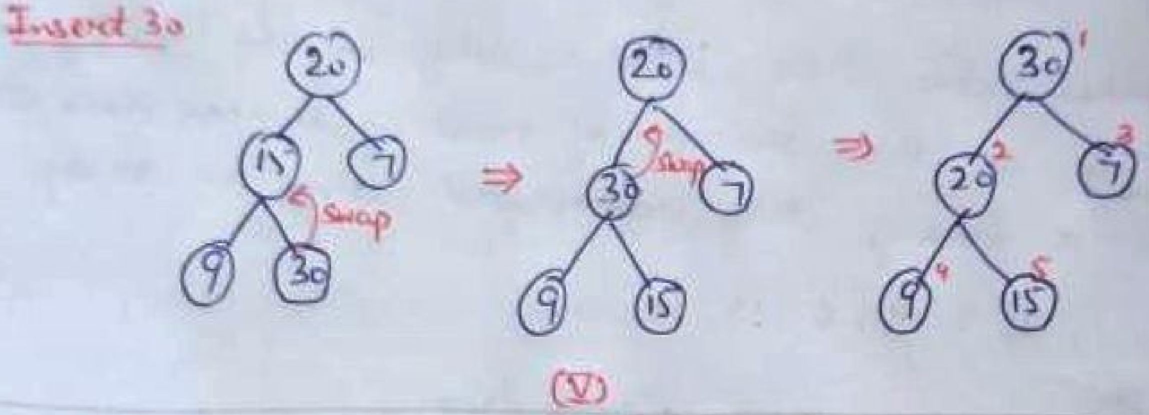
(ii.) After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then Store the heap structure with the remaining elements.

Eg:-

15	20	7	9	30

Step I:- Insertion



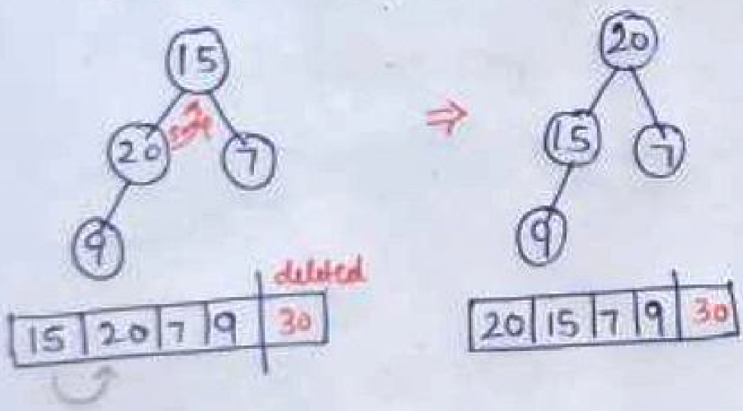


Step II:- Deletion :- delete root node with last element of array.

1	2	3	4	5
30	20	7	9	15

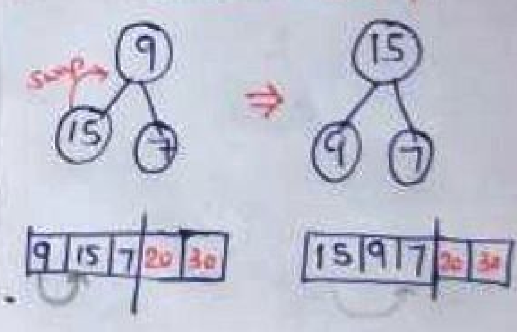
(I) delete 30 with 15.

(II) delete 20 with 9.



15	20	7	9	30

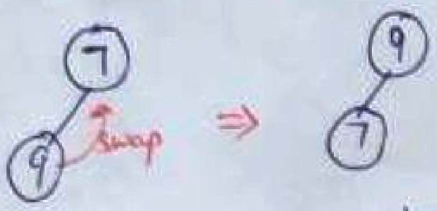
20	15	7	9	30



9	15	7	20	30

15	9	7	20	30

III) delete 15 with 7.



7	9	15	20	30

9	7	15	20	30

IV) delete 9 with 7

7	9	15	20	30

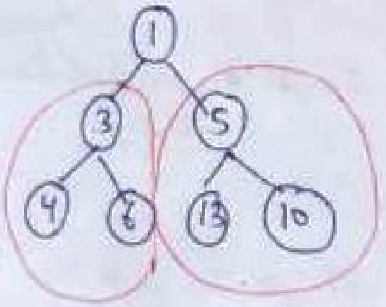
v) delete 7, then the sorted array is

7	9	15	20	30

It takes more time that's why we use Heapify method.

Heapify is the process of creating a heap data structure from a binary tree represented using an array.

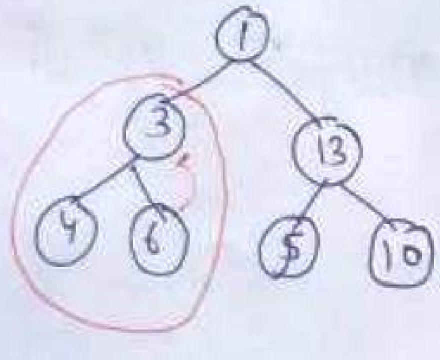
Eg:- 1, 3, 5, 4, 6, 13, 10.



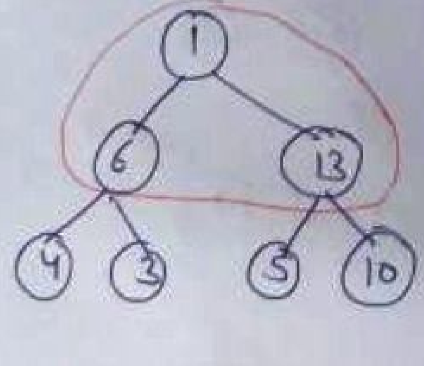
Check subtree from leaf node & then switch if max or min-heap property is not satisfiable

⇓

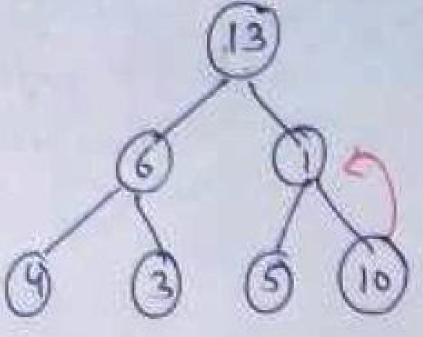
I) Swap 13 with 5



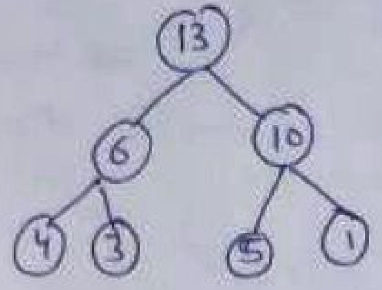
II) Swap 3 with 6.



III) Swap 1 with 13.



IV Swap 1 with 10.



```
void heapify (int A[], int n, int i)
{
        int largest = i;
        int l = 2×i;              // To calculate left &
        int r = 2×i + 1;          //   right value.

    if (l < n && A[l] > A[largest])
            largest = l;

    if (r < n && A[r] > A[largest])
            largest = r;

     if (largest ! = i)
      {     int temp = A[i];          // Swapping of A[i] &
                A[i] = A[largest];    //   A[largest].
            A[largest] = temp;
      }
         heapify (A, n, largest)
  }
```

For heapsorting

```
   void heapsort (int A[], int n)
   {
          for (int i = n/2 - 1 , i >= 0, i--)
                  heapify (A[], n, i) ;
```

```
for ( int i = n-1 ;  i >= 0 ; i -- )
    {
          int temp = A [0] ;            // Swapping & heapify for getting
              A[0] = A[i] ;                highest value at
              A[i] = temp ;                       root .
       }

          Heapify ( A, i, 0 );
  }
```

Advantages :-

1) Fast , when we have less elements.
2) Used in suffix array construction algorithms
3) Easy .

Disadvantages :-

1) Consume time .
2) Memory Management is complex.

Radix Sort :- It is the linear sorting algorithm that is used for integers. In radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.

Algorithm :-

1) Take an unsorted list .
2) Find least significant digit of the elements. Also find largest element 'max' with 'x' no. of digits.
3) After that, go through one by one each significant

place and sort the list.

Eg:- 181, 289, 390, 121, 145, 736, 514, 212

Pass I :- Check least Significant bit & put in particular bucket.

0	390
1	181,121
2	212
3	
4	514
5	145
6	736
7	
8	
9	289

Pass II :- Check other significant bit i.e. ten's place.

390, 181, 121, 212, 514, 145, 736, 289

0	
1	212, 514
2	121
3	736
4	145
5	
6	
7	
8	181, 289
9	390

212, 514, 121, 736, 145, 181, 289, 390

Pass III :- Check other significant bit i.e hundred's place.

0	
1	121, 145, 181
2	212, 289
3	390
4	
5	514
6	
7	736
8	
9	

Advantages :-

1) Fast

2) Used in algorithms like DC3.

Disadvantages :-

1) Less flexible.

2) More comparison.

The sorted array is :- 121, 145, 181, 212, 289, 390, 514, 736

```
void countsort (A [], int n, int p)
{
    int B[];
    int i, count[10] = {0};              // All value 0.
    for( i = 0 ;  i<n;  i++ )
     Count [[A [i]/p) % 10]++ ;          // Update 0 by 1,2 - so on
                                         //   as per significant
                                         //   bits
    for (i=1 ; i<10 ; i++)
      Count [i]+ = Count [i-1];
      for(i = n-1 ; i > = 0 ; i--)       // put values in
      {                                  //   B array.
          B[ count[ (A[i]/pos) % 10] -1] = A[i] ;
           Count [(A[i]/pos) % 10]-- ;
      }
      for(i = 0; i<n;  i++)
      {
             A[i] = B[i];                // Copy of B arrey into A.
       }
```