Unit 2: **CHAPTER-2**

Sorting Techniques

2.0 OBJECTIVE :

After studying this unit, you will be able to:

- To study basic concepts of sorting.
- To study different sorting methods and their algorithms.
- Study different sorting methods and compare with their time complexity

2.1 INTRODUCTION TO SORTING

Arranging the data in ascending or descending order is known as sorting.
Sorting is very important from the point of view of our practical life.
The best example of sorting can be phone numbers in our phones. If, they are not maintained in an alphabetical order we would not be able to search any number effectively.
There are two types of sorting:
Internal Sorting:
If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.
External Sorting:

When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc, then external sorting methods are performed.

Application of sorting

1. The sorting is useful in database applications for arranging the data in desired ORDER.
2. In the dictionary like applications the data is arranged in sorted order.
3. For searching the element from the list of elements the sorting is required
4. For checking the uniqueness of the element the sorting is required.
5. For finding the closest pair from the list of elements the sorting is required.

2.2 SORTING TECHNIQUES

1) Bubble sort
2) Insertion sort
3) Radix Sort
4) Quick sort
5) Merge sort
6) Heap sort
7) Selection sort
8) shell Sort

2.2.1 BUBBLE SORT

In *bubble sorting*, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because elements ‘bubble’ to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

Note : If the elements are to be sorted in descending order, then in first pass the smallest element is movedto the highest index of the array.

Example To discuss bubble sort in detail, let us consider an arrayA[]that has the followingelements:

A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:

Compare 30 and 52. Since 30 < 52, no swapping is done.
Compare 52 and 29. Since 52 > 29, swapping is
done. 30, **29, 52**, 87, 63, 27, 19, 54

Compare 52 and 87. Since 52 < 87, no swapping is done.
Compare 87 and 63. Since 87 > 63, swapping is
done. 30, 29, 52, **63, 87**, 27, 19, 54

Compare 87 and 27. Since 87 > 27, swapping is
done. 30, 29, 52, 63, **27, 87**, 19, 54

Compare 87 and 19. Since 87 > 19, swapping is
done. 30, 29, 52, 63, 27, **19**, **87**, 54

Compare 87 and 54. Since 87 > 54, swapping is
done. 30, 29, 52, 63, 27, 19, **54, 87**

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

Pass 2:

Compare 30 and 29. Since 30 > 29, swapping is
done. **29, 30**, 52, 63, 27, 19, 54, 87

Compare 30 and 52. Since 30 < 52, no swapping is done.

Compare 52 and 63. Since 52 < 63, no swapping is done.

Compare 63 and 27. Since 63 > 27, swapping is
done. 29, 30, 52, **27, 63**, 19, 54, 87

Compare 63 and 19. Since 63 > 19, swapping is done.
29, 30, 52, 27, **19, 63**, 54, 87

Compare 63 and 54. Since 63 > 54, swapping
is done. 29, 30, 52, 27, 19, **54, 63**, 87

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

Pass 3:

Compare 29 and 30. Since 29 < 30, no swapping is done.

Compare 30 and 52. Since 30 < 52, no swapping is done.

Compare 52 and 27. Since 52 > 27, swapping
is done. 29, 30, **27, 52**, 19, 54, 63, 87

Compare 52 and 19. Since 52 > 19, swapping
is done. 29, 30, 27**, 19, 52**, 54, 63, 87

Compare 52 and 54. Since 52 < 54, no swapping is done.

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

Pass 4:

Compare 29 and 30. Since 29 < 30, no swapping is done.

Compare 30 and 27. Since 30 > 27, swapping
is done. 29, **27, 30**, 19, 52, 54, 63, 87

Compare 30 and 19. Since 30 > 19, swapping
is done. 29, 27, **19, 30**, 52, 54, 63, 87

Compare 30 and 52. Since 30 < 52, no swapping is done.

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

Pass 5:

Compare 29 and 27. Since 29 > 27, swapping
is done. **27, 29,** 19, 30, 52, 54, 63, 87

Compare 29 and 19. Since 29 > 19, swapping
is done. 27, **19, 29**, 30, 52, 54, 63, 87

Compare 29 and 30. Since 29 < 30, no swapping is done.

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

Pass 6:

Compare 27 and 19. Since 27 > 19, swapping is done. **19, 27,** 29, 30, 52, 54, 63, 87

Compare 27 and 29. Since 27 < 29, no swapping is done.

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

Pass 7:

(a) Compare 19 and 27. Since 19 < 27, no swapping is done.

Observe that the entire list is sorted now.

Algorithm for bubble sort

BUBBLE_SORT(A, N)

```
Step 1: Repeat Step 2 For 1 = 0 to N-1
Step 2: Repeat For J =  to N – I
Step 3:      IF A[J] >A[J+ 1]
                    SWAP A[J] and A[J+1]
             [END OF INNER LOOP]
           [END OF OUTER LOOP]
 Step 4: EXIT
```

Advantages :

- Simple and easy to implement
- In this sort, elements are swapped in place without using additional temporary storage, so the space requirement is at a minimum.

Disadvantages :

- It is slowest method . O(n^2)
- Inefficient for large sorting lists.

Program

```
#include<stdio.h>
void main ()
```

```
    {
        int i, j,temp;
        int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
        for(i = 0; i<10; i++)
        {
            for(j = i+1; j<10; j++)
            {
                if(a[j] > a[i])
                {
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }
        printf("Printing Sorted Element List ...\n");
        for(i = 0; i<10; i++)
        {
            printf("%d\n",a[i]);
        }
    }
```

Output:

```
Printing Sorted Element List . . .
7
9
10
12
23
34
34
44
78
101
```

Complexity of Bubble Sort

The complexity of any sorting algorithm depends upon the number of comparisons. In bubble sort, we have seen that there are N–1 passes in total. In the first pass, N–1comparisons are made to place the highest element in its correct position. Then, in Pass 2, there are N–2 comparisons and the second highest element is placed in its position. Therefore, to compute the complexity of bubble sort, we need to calculate the total number of comparisons. It can be given as:

$$f(n) = (n - 1) + (n - 2) + (n - 3) + \ldots\ldots + 3 + 2 + 1$$

$$f(n) = n\,(n - 1)/2$$

$$f(n) = n2/2 + O(n) = O(n2)$$

Therefore, the complexity of bubble sort algorithm is O(n2). It means the time required to execute bubble sort is proportional to n2, where n is the total number of elements in the array.

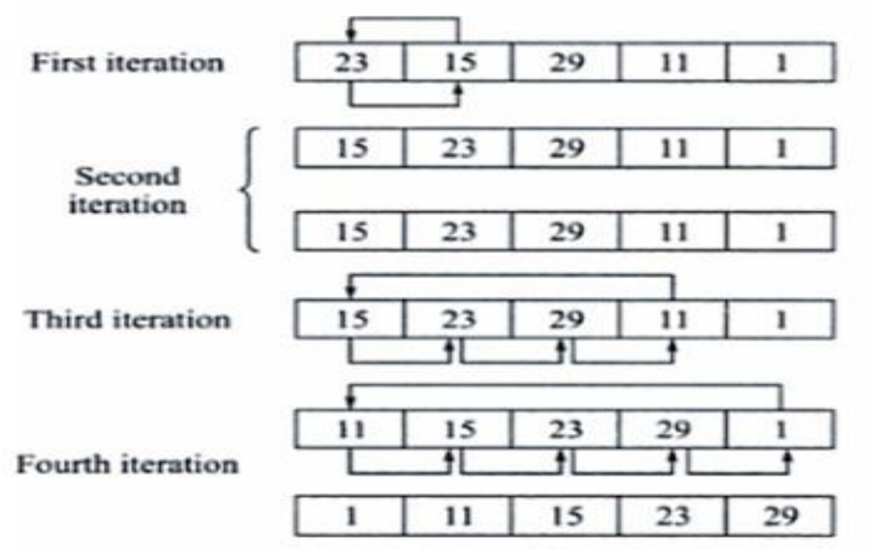
2.2.2 INSERTION SORT

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.

Insertion sort inserts each item into its proper place in the final list. In insertion sort , the first iteration starts with comparison of 1st element with 0th element. In the second iteration 2nd element is compared with the 0th and 1st element and so on. In every iteration an element is compared with all elements. The main idea is to insert the ith pass the ith element in A[1], A[2]…A[i] in its proper place.

Example Consider an array of integers given below. We will sort the values in the arrayusing insertion sort

23	15	29	11	1



0

Algorithm for insertion sort

```
    INSERTION-SORT (ARR, N)
  Step 1: Repeat Steps 2 to 5 for K = 1 to N – 1
  Step 2: SET TEMP = ARR[K]
  Step 3: SET J = K - 1
  Step 4: Repeat while TEMP <= ARR[J]
              SET ARR[J + 1] = ARR[J]
              SET J = J - 1
                      [END OF INNER LOOP]
  Step 5: SET ARR[J + 1] = TEMP
          [END OF LOOP]
  Step 6: EXIT
```

Advantages:

The advantages of this sorting algorithm are as follows:

- Relatively simple and Easy to implement.
- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- The insertion sort is an in-place sorting algorithm so the space requirement is minimal.

Disadvantages:

- Inefficient for large list O (n2).

Program

```
#include<stdio.h>
void main ()
{
   int i,j, k,temp;
   int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   printf("\nprinting sorted elements...\n");
   for(k=1; k<10; k++)
   {
      temp = a[k];
      j= k-1;
      while(j>=0 && temp <= a[j])
      {
         a[j+1] = a[j];
         j = j-1;
      }
      a[j+1] = temp;
   }
   for(i=0;i<10;i++)
   {
      printf("\n%d\n",a[i]);
   }
}
```

Output:

```
Printing Sorted Elements . . .
7
9
10
12
23
23
34
```

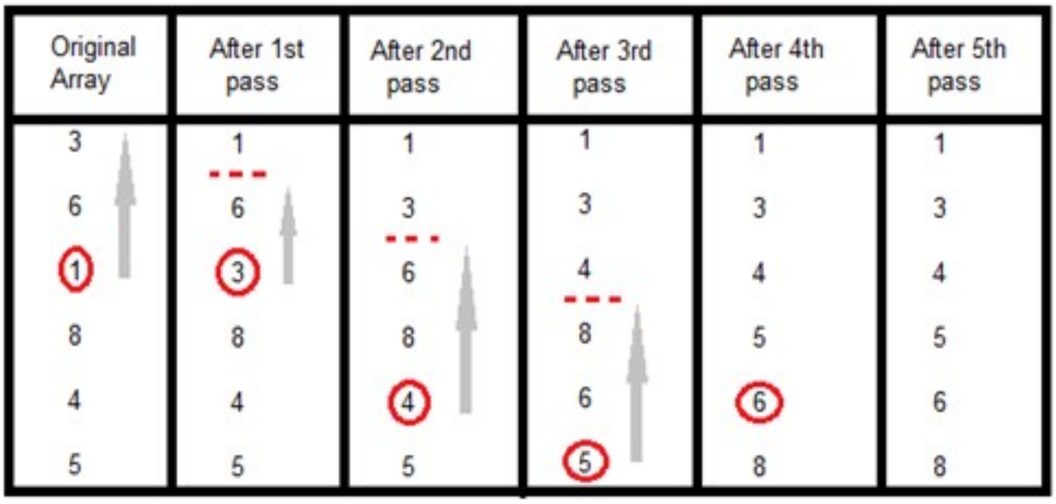
```
44
78
101
```

Complexity of Insertion Sort
If the initial tile is sorted, only one comparison is made on each iteration, so that the sort is O(n). If the file is initially sorted in the reverse order the worst case complexity is O(n^2). Since the total number of comparisons is:
(n-1)+ (n-2) +...+3+2+1= (n-1) * n/2, which is O (n^2)
The average case or the average number of comparisons in the simple insertion sort is O (n^2).

2.2.3 SELECTION SORT

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then finds the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

Example 1 : 3, 6, 1, 8, 4, 5



Original Array	After 1st pass	After 2nd pass	After 3rd pass	After 4th pass	After 5th pass
3	1	1	1	1	1
6	6	3	3	3	3
1	3	6	4	4	4
8	8	8	8	5	5
4	4	4	6	6	6
5	5	5	5	8	8

Algorithm for selection sort

```
SELECTION SORT(ARR, N)
    Step 1: Repeat Steps 2 and 3 for K = 1 to N-1
   Step 2: CALL SMALLEST(ARR, K, N, POS)
   Step 3: SWAP A[K] with ARR[POS]
```

```
            [END OF LOOP]
   Step 4: EXIT

SMALLEST (ARR, K, N, POS)
    Step 1: [INITIALIZE] SET SMALL = ARR[K]
    Step 2: [INITIALIZE] SET POS = K
    Step 3: Repeat for J = K+1 to N
                IF SMALL > ARR[J]
                    SET SMALL = ARR[J]
                    SET POS = J
                [END OF IF]
            [END OF LOOP]
   Step 4: RETURN POS
```

Advantages:

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort.

Disadvantages:

- Running time of Selection sort algorithm is very poor of 0 (n2).
- However, in case of large data sets, the efficiency of selection sort drops as compared to insertion sort.

Program

```
#include<stdio.h>
int smallest(int[],int,int);
void main ()
{
   int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   int i,j,k,pos,temp;
   for(i=0;i<10;i++)
   {
      pos = smallest(a,10,i);
      temp = a[i];
      a[i]=a[pos];
      a[pos] = temp;
```

```
        }
        printf("\nprinting sorted elements...\n");
        for(i=0;i<10;i++)
        {
            printf("%d\n",a[i]);
        }
    }
    int smallest(int a[], int n, int i)
    {
        int small,pos,j;
        small = a[i];
        pos = i;
        for(j=i+1;j<10;j++)
        {
            if(a[j]<small)
            {
                small = a[j];
                pos=j;
            }
        }
        return pos;
    }
```

Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

Complexity of Selection Sort

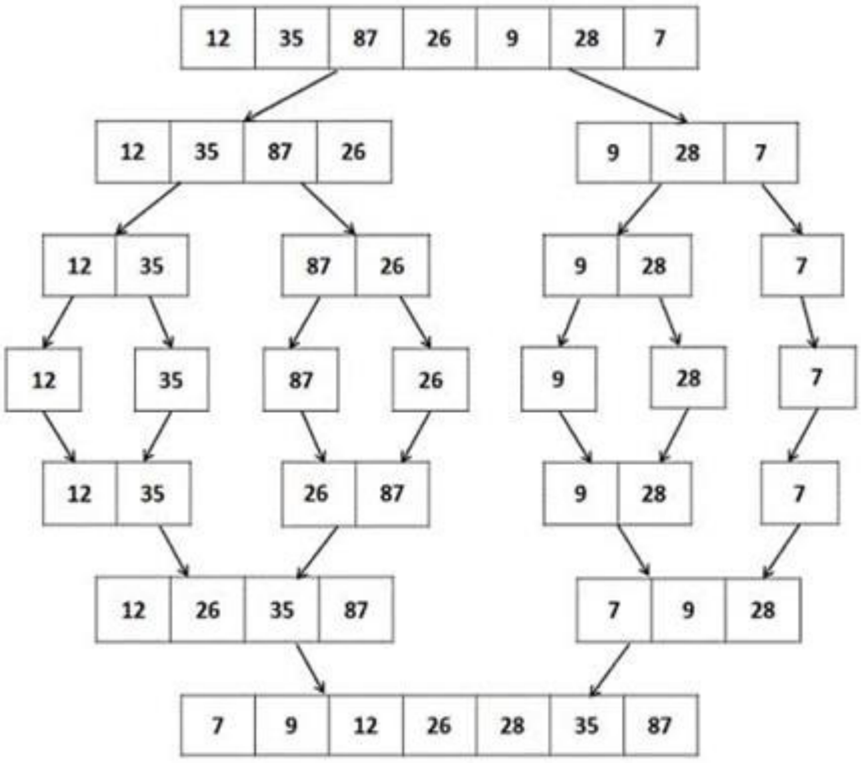
The first element is compared with the remaining n-1 elements in pass 1. Then n-2 elements are taken in pass 2, this process is repeated until the last element is encountered. The mathematical expression for these iterations will be equal to: (n-1)+(n-2)+….,+(n-(n-1)).Thus the expression become n*(n-1)/2. Thus, the number of comparisons is proportional to (n^2). The time complexity of selection sort is O(n^2).

2.2.4 MERGE SORT

Merging means combining two sorted lists into one-sorted list. The merge sort splits the array to be sorted into two equal halves and each array is recursively sorted, then merged back together to form the final sorted array. The logic is to split the array into two sub arrays each sub array is individually sorted and the resulting sequence is then combined to produce a single sorted sequence of n elements.

The merge sort recursively follows the steps:
1) Divide the given array into equal parts.
2) Recursively sorts the elements on the left side of the partitions.
3) Recursively sorts the elements on the right side of the partitions.
4) Combine the sorted left and right partitions into a single sorted array.

Example Sort the array given below using merge sort.
12 , 35 ,87, 26, 9, 28,7



Merge Sort

Algorithm for merge sort

```
MERGE_SORT(ARR, BEG, END)
Step 1: IF BEG < END
            SET MID = (BEG + END)/2
            CALL MERGE_SORT (ARR, BEG, MID)

            CALL MERGE_SORT (ARR, MID + 1, END)
            MERGE (ARR, BEG, MID, END)
        [END OF IF]
Step 2: END



                  MERGE (ARR, BEG, MID, END)

                  Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
                  Step 2: Repeat while (I <= MID) AND (J<=END)
                              IF ARR[I] < ARR[J]
```

```
                SET TEMP[INDEX] = ARR[I]
                SETI=I+1
            ELSE
                SET TEMP[INDEX] = ARR[J]
                SETJ=J+1
            [END OF IF]
            SET INDEX = INDEX + 1
        [END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
            IFI>MID
              Repeat while J <= END
                SET TEMP[INDEX] = ARR[J]
                SET INDEX = INDEX + 1, SET J = J + 1
              [END OF LOOP]
        [Copy the remaining elements of left sub-array, if any]
            ELSE
              Repeat while I <= MID
                SET TEMP[INDEX] = ARR[I]
                SET INDEX = INDEX + 1, SET I = I + 1
              [END OF LOOP]
            [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K=0
Step 5: Repeat while K < INDEX
              SET ARR[K] = TEMP[K]
              SETK=K+1
        [END OF LOOP]
Step 6: END
```

Advantages :

- Merge sort algorithm is best case for sorting slow-access data e.g) tape drive.
- Merge sort algorithm is better at handling sequential - accessed lists.

Disadvantages:

- Slower comparative to the other sort algorithms for smaller tasks.
- Merge sort algorithm requires additional memory space of 0(n) for the temporary array .
- It goes through the whole process even if the array is sorted.

Program

```
#include<stdio.h>
void mergeSort(int[],int,int);
void  merge(int[],int,int,int);
void main ()
{
   int a[10]= {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   int i;
   mergeSort(a,0,9);
   printf("printing the sorted elements");
   for(i=0;i<10;i++)
   {
      printf("\n%d\n",a[i]);
   }


}
void mergeSort(int a[], int beg, int end)
{
   int mid;
   if(beg<end)
   {
      mid = (beg+end)/2;
      mergeSort(a,beg,mid);
      mergeSort(a,mid+1,end);
```

```
        merge(a,beg,mid,end);
    }
}
void merge(int a[], int beg, int mid, int end)
{
    int i=beg,j=mid+1,k,index = beg;
    int temp[10];
    while(i<=mid && j<=end)
    {
        if(a[i]<a[j])
        {
            temp[index] = a[i];
            i = i+1;
        }
        else
        {
            temp[index] = a[j];
            j = j+1;
        }
        index++;
    }
    if(i>mid)
    {
        while(j<=end)
        {
            temp[index] = a[j];
            index++;
            j++;
        }
    }
    else
    {
        while(i<=mid)
```

```
                {
                    temp[index] = a[i];
                    index++;
                    i++;
                }
            }
            k = beg;
            while(k<index)
            {
                a[k]=temp[k];
                k++;
            }
        }
```

Output:

```
printing the sorted elements

7
9
10
12
23
23
34
44
78
101
```

Complexity of Merge Sort

The running time of merge sort in the average case and the worst case can be given as O(n logn). Although merge sort has an optimal timecomplexity, it needs an additional space of O(n) for the temporary array TEMP.

Applications

- Merge Sort is useful for sorting linked lists in O(nLogn) time.
- Inversion Count Problem
- Used in External Sorting

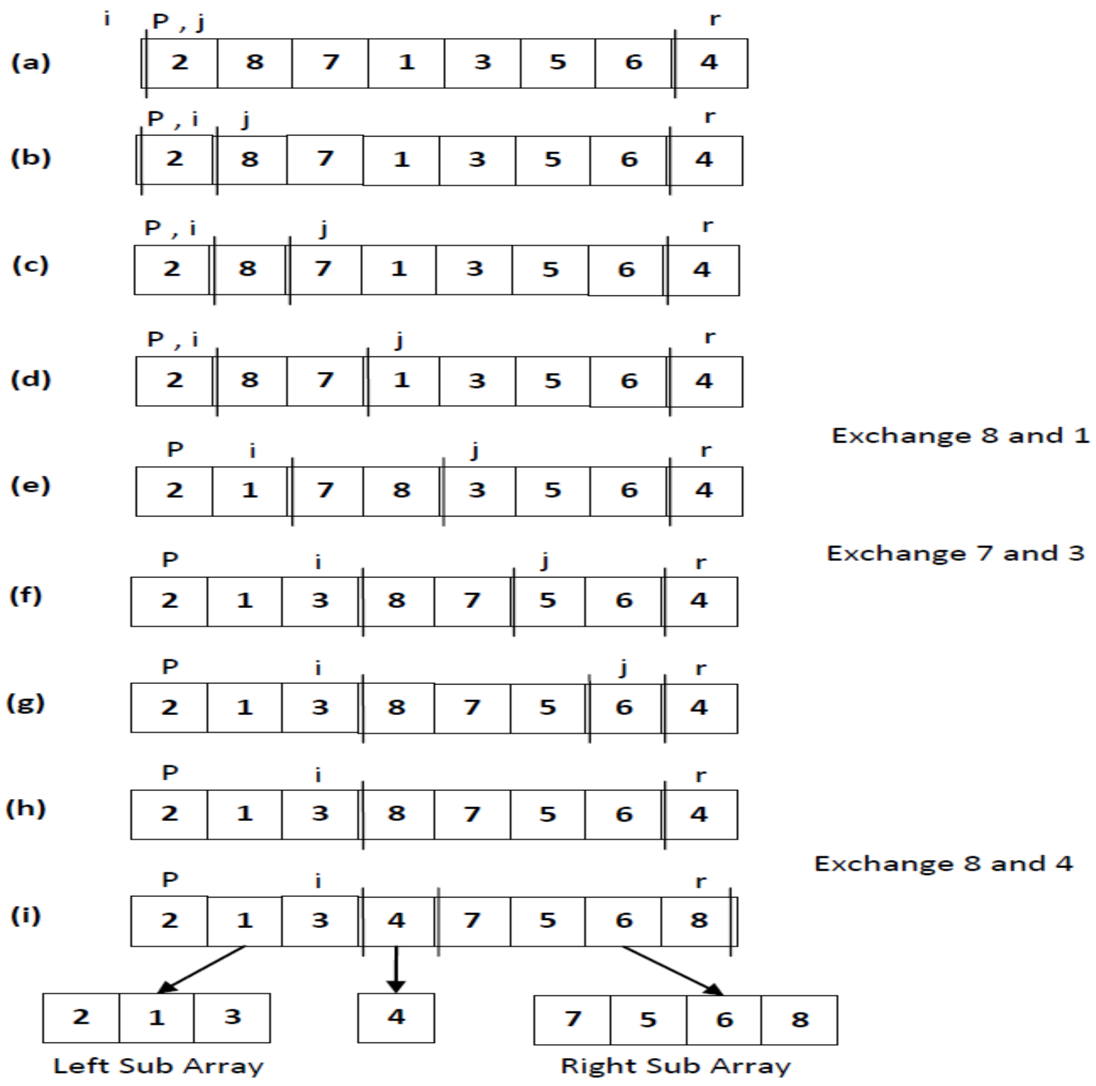
2.2.5 QUICK SORT

Quick sort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.
The steps are:
1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so
that all elements greater than the pivot come after it (equal values can go either way). After
this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

Example

Sort given array using Quick Sort:

2	8	7	1	3	5	6	4



Apply same method for left and right sub array finally we will get sorted

Algorithm for Quick Sort

```
                QUICK_SORT (ARR, BEG, END)

                Step 1: IF (BEG < END)
                            CALL PARTITION (ARR, BEG, END, LOC)
                            CALL QUICKSORT(ARR, BEG, LOC - 1)
                            CALL QUICKSORT(ARR, LOC + 1, END)
                        [END OF IF]
                Step 2: END


PARTITION (ARR, BEG, END, LOC)

    Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG =
    Step 2: Repeat Steps 3 to 6 while FLAG =

    Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND
LOC!= RIGHT SET RIGHT = RIGHT - 1
            [END OF LOOP]
    Step 4: IF LOC = RIGHT
                 SET FLAG = 1
            ELSE IF ARR[LOC] > ARR[RIGHT]
                 SWAP ARR[LOC]  with ARR[RIGHT]
                 SET LOC = RIGHT
             [END OF IF]
    Step 5: IF FLAG = 0
               Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
               SET LEFT = LEFT + 1
            [END OF LOOP]
    Step 6:IF LOC = LEFT
                 SET FLAG = 1
            ELSE IF ARR[LOC] < ARR[LEFT]
                 SWAP ARR[LOC]  withARR[LEFT]
                 SET LOC = LEFT
            [END OF IF]
            [END OF IF]
   Step 7: [END OF LOOP]
   Step 8: END
```

Advantages:

- Extremely fast O(n $\log_2$ n)
- Gives good results when an array is in random order.

- Quick sort can be used to sort arrays of small size, medium size, or large size.

Disadvantages:

- Algorithm is very complex
- In worst case of quick sort algorithm, the time efficieny is very poor which is very much likely to selection sort algorithm i.e) n(log 2 n).

Program

```
#include <stdio.h>
int partition(int a[], int beg, int end);
void quickSort(int a[], int beg, int end);
void main()
{
   int i;
   int arr[10]={90,23,101,45,65,23,67,89,34,23};
   quickSort(arr, 0, 9);
   printf("\n The sorted array is: \n");
   for(i=0;i<10;i++)
   printf(" %d\t", arr[i]);
}
int partition(int a[], int beg, int end)
{

   int left, right, temp, loc, flag;
   loc = left = beg;
   right = end;
   flag = 0;
   while(flag != 1)
   {
      while((a[loc] <= a[right]) && (loc!=right))
      right--;
      if(loc==right)
      flag =1;
      else if(a[loc]>a[right])
      {
            temp = a[loc];
            a[loc] = a[right];
            a[right] = temp;
            loc = right;
        }
        if(flag!=1)
        {
            while((a[loc] >= a[left]) && (loc!=left))
            left++;
            if(loc==left)
            flag =1;
            else if(a[loc] <a[left])
            {
                temp = a[loc];
                a[loc] = a[left];
                a[left] = temp;
                loc = left;
            }
        }
    }
    return loc;
}
void quickSort(int a[], int beg, int end)
{

    int loc;
    if(beg<end)
    {
        loc = partition(a, beg, end);
        quickSort(a, beg, loc-1);
        quickSort(a, loc+1, end);
    }
}
```

Output:

```
The sorted array is:
23
23
23
34
45
65
67
89
90
101
```

Complexity of Quick Sort

Pass 1 will have n comparisons. Pass 2 will have 2*(n/ 2) comparisons. In the subsequent passes will have 4*(n/4), 8*(n/8) comparisons and so on. The total comparisons involved in this case would be O(n)+O(n)+O(n)+···+s. The value of expression will be O (n log n).Thus time complexity of quick sort is O(n log n).Space required by quick sort is very less, only O(n log n) additional space is required.

2.2.6 RADIX SORT

Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order. In radix sort algorithm, a list of integer numbers will be sorted based on the digits of individual numbers. Sorting is performed from **least significant digit to the most significant digit**.

Radix sort algorithm requires the number of passes which are equal to the number of digits present in the largest number among the list of numbers. For example, if the largest number is a 3 digit number then that list is sorted with 3 passes.

Step by Step Process

The Radix sort algorithm is performed using the following steps...

- **Step 1 -** Define 10 queues each representing a bucket for each digit from 0 to 9.
- **Step 2 -** Consider the least significant digit of each number in the list which is to be sorted.
- **Step 3 -** Insert each number into their respective queue based on the least significant digit.
- **Step 4 -** Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
- **Step 5 -** Repeat from step 3 based on the next least significant digit.
- **Step 6 -** Repeat from step 2 until all the numbers are grouped based on the most significant digit.

Algorithm for Radix Sort

Algorithm for RadixSort (ARR, N)

```
Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:SET I = 0 and INITIALIZE buckets
Step 6:Repeat Steps 7 to 9 while I<N-1
Step 7:SET DIGIT    = digit at PASSth place in A[I]
Step 8:Add A[I] to the bucket numbered DIGIT
Step 9:INCEREMENT bucket count for bucket numbered DIGIT
            [END OF LOOP]
Step 1 :Collect the numbers in the bucket
             [END OF LOOP]
Step 11: END
```

Example Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result. After the third pass, the list can be given as

123, 345, 472, 555, 567, 654, 808, 911, 924.

Advantages:

- Radix sort algorithm is well known for its fastest sorting algorithm for numbers and even for strings of letters.
- Radix sort algorithm is the most efficient algorithm for elements which are arranged in descending order in an array.

Disadvantages:

- Radix sort takes more space than other sorting algorithms.
- Poor efficieny for most elements which are already arranged in ascending order in an array.
- When Radix sort algorithm is applied on very small sets of data(numbers or strings), then algorithm runs in 0(n) asymptotic time.

Program

```
#include <stdio.h>
int largest(int  a[]);
void radix_sort(int a[]);
void main()
{
   int i;
   int a[10]={90,23,101,45,65,23,67,89,34,23};
   radix_sort(a);
   printf("\n The sorted array is: \n");
   for(i=0;i<10;i++)
      printf(" %d\t", a[i]);
}

int largest(int a[])
{
   int larger=a[0], i;
   for(i=1;i<10;i++)
   {
      if(a[i]>larger)
      larger = a[i];
   }
   return larger;
}
void radix_sort(int a[])
{
   int bucket[10][10], bucket_count[10];
   int i, j, k, remainder, NOP=0, divisor=1, larger, pass;
   larger = largest(a);
   while(larger>0)
   {
      NOP++;
      larger/=10;
   }
```

```
        for(pass=0;pass<NOP;pass++) // Initialize the buckets
        {
          for(i=0;i<10;i++)
          bucket_count[i]=0;
          for(i=0;i<10;i++)
          {
            // sort the numbers according to the digit at passth place
            remainder = (a[i]/divisor)%10;
            bucket[remainder][bucket_count[remainder]] = a[i];
            bucket_count[remainder] += 1;
          }
          // collect the numbers after PASS pass
          i=0;
          for(k=0;k<10;k++)
          {
            for(j=0;j<bucket_count[k];j++)
            {
              a[i] = bucket[k][j];
              i++;
            }
          }
          divisor *= 10;

      }
      }
```

Output:

```
The sorted array is:
23
23
23
34
45
65
67
89
```

```
90
101
```

Complexity of Radix Sort
To sort an unsorted list with **'n'** number of elements, Radix sort algorithm needs the following complexities...

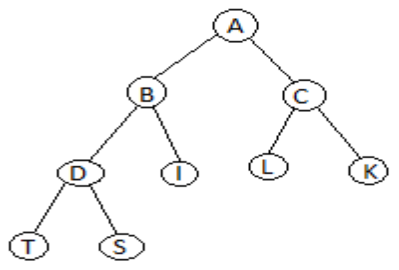
Worst Case : O(n)
Best Case : O(n)
Average Case : O(n)

2.2.7 HEAP SORT

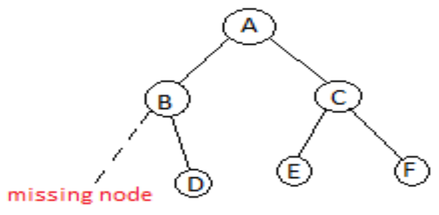
A) Terms in Heap:

a) Heap:
Heap is a special tree-based data structure that satisfies the following special heap properties

b) Shape Property:
Heap data structure is always a complete Binary Tree, which means all levels of the tree are fully filled.

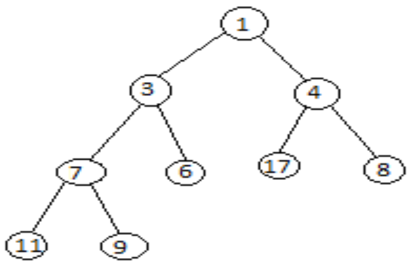


Complete Binary Tree



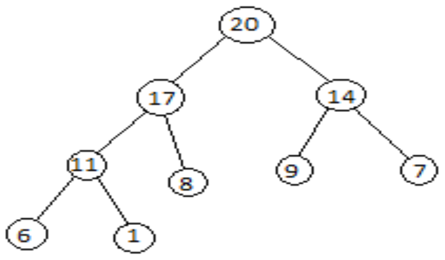
In-Complete Binary Tree

c) Heap Property:
All nodes are either (greater than or equal to) or (less than or equal to) each of its children. If the parent nodes are greater than their children, heap is called a Max-Heap, and if the parent nodes are small than their child nodes, heap is called Min-Heap.



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.

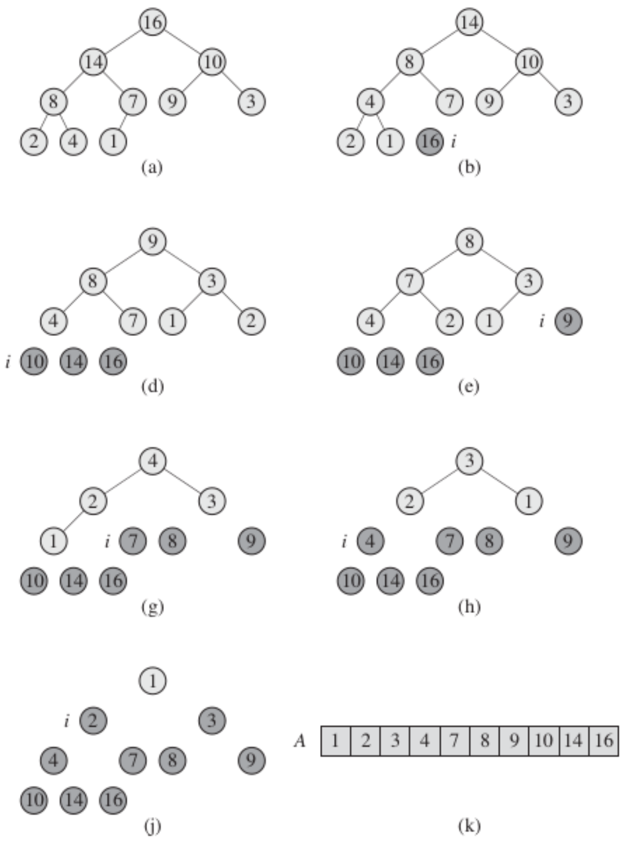


Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

B) Working of Heap Sort:
Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap). Once heap is built, the first element of the Heap I either largest or smallest(depending upon Max-Heap or min-Heap), so put the first element of the heap in array. Then again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. Keep on doing the same repeatedly until we have the complete sorted list in array.

Example 2 : 4 ,1,3 ,2 ,16,9,10,14, 8,7



Operations on the heap :

1. Inserting An Element into Heap :

The elements are always inserted at the bottom of the original heap. After insertion, the heap remains complete but the order property is not followed so we use an UPHEAP or HEAPIFY operation. This involves moving the elements upward from the last position where it satisfies the order property. If the value of last node is greater than its parent, exchange it's value with it's parent and repeat the process with each parent node until the order property is satisfied.

2. Deleting an Element from Heap :

Elements are always deleted from the root of the heap.

Algorithm for insertion of element :

INHEAP (TREE, N, ITEM)

A heap H with N elements is stored in the array TREE, and an item of information is given. This procedure inserts ITEM as a new element of H. PTR gives the location of ITEM as it rises in the tree, and PAR denotes the location of the parent of ITEM.

```
 1. Set N = N +1 and PTR = N                [Add new node to H and initialize PTR].
 2. Repeat Steps 3 to 6 while PTR<1         [Find location to insert ITEM].
 3. Set PAR = [PTR/2].                      [Location of parent node].
 4. If ITEM ≤ TREE[PAR], then :
        Set TREE[PTR] = ITEM, and return.
        [End of If Structure].
 5. Set TREE[PTR] = TREE[PAR].              [Moves node down]
 6. Set PTR = PAR                           [Updates PTR]
        [End of step 2 loop].
 7. Set TREE[1] = ITEM                      [Assign ITEM as a root of H].
 8. Exit
```

Algorithm for deletion of element :

DELHEAP(TREE,N,ITEM)

A heap H with N elements is stored in the array TREE. This procedure assigns the root TREE[1] of H to the variable ITEM and then reheaps the remaining elements. The variable LAST saves the value of the original last node of H. The pointers PTR, LEFT and RIGHT give the locations of LAST and its left and right children as LAST sinks in the tree.

```
 1. Set ITEM = TREE[1].                          [Removes root of H].
 2. Set LAST = TREE[N] and N = N – 1             [Removes last node of H]
 3. Set PTR = 1, LEFT = 2 and RIGHT = 3          [Initialize Pointers]
4.  Repeat Steps 5 to 7 while RIGHT ≤ N :

5.     If LAST ≥ TREE[LEFT] and LAST ≥ TREE[RIGHT], then :
    6.  If TREE[RIGHT] ≤ TREE[L


    7.  EFT] and PTR = LEFT

        Set TREE[PTR] = TREE[LEFT]
      and PTR = LEFT. Else

        Set TREE[PTR] = TREE[RIGHT] and
        PTR = RIGHT. [End of If structure]

7.  Set LEFT = 2 * PTR and RIGHT = LEFT + 1

        [End of Step 4 loop].
```

```
8.  If LEFT = N and if LAST M TREE[LEFT], then Set PTR = LEFT.

9.  Set TREE[PTR] = LAST

10. Exit
```

Advantages:

- Heap Sort is very fast and is widely used for sorting.
- Heap sort algorithm can be used to sort large sets of data.

Disadvantages:

- Heap sort is not a Stable sort, and requires a constant space for sorting a list.
- Heap sort algorithm's worst case comes with the running time of 0(n log (n)) which is more likely to merge sort algorithm.

Program

```
#include<stdio.h>
int temp;

void heapify(int arr[], int size, int i)
{
int largest = i;
int left = 2*i + 1;
int right = 2*i + 2;

if (left < size && arr[left] >arr[largest])
largest = left;

if (right < size && arr[right] > arr[largest])
largest = right;

if (largest != i)
{
temp = arr[i];
```

```
    arr[i]= arr[largest];
     arr[largest] = temp;
heapify(arr, size, largest);
}
}

void heapSort(int arr[], int size)
{
int i;
for (i = size / 2 - 1; i >= 0; i--)
heapify(arr, size, i);
for (i=size-1; i>=0; i--)
{
temp = arr[0];
    arr[0]= arr[i];
    arr[i] = temp;
heapify(arr, i, 0);
}
}

void main()
{
int arr[] = {1, 10, 2, 3, 4, 1, 2, 100,23, 2};
int i;
int size = sizeof(arr)/sizeof(arr[0]);

heapSort(arr, size);

printf("printing sorted elements\n");
for (i=0; i<size; ++i)
printf("%d\n",arr[i]);
}
```

Output:

```
printing sorted elements

1
1
2
2
2
3
4
10
23
100
```

Complexity of Heap Sort

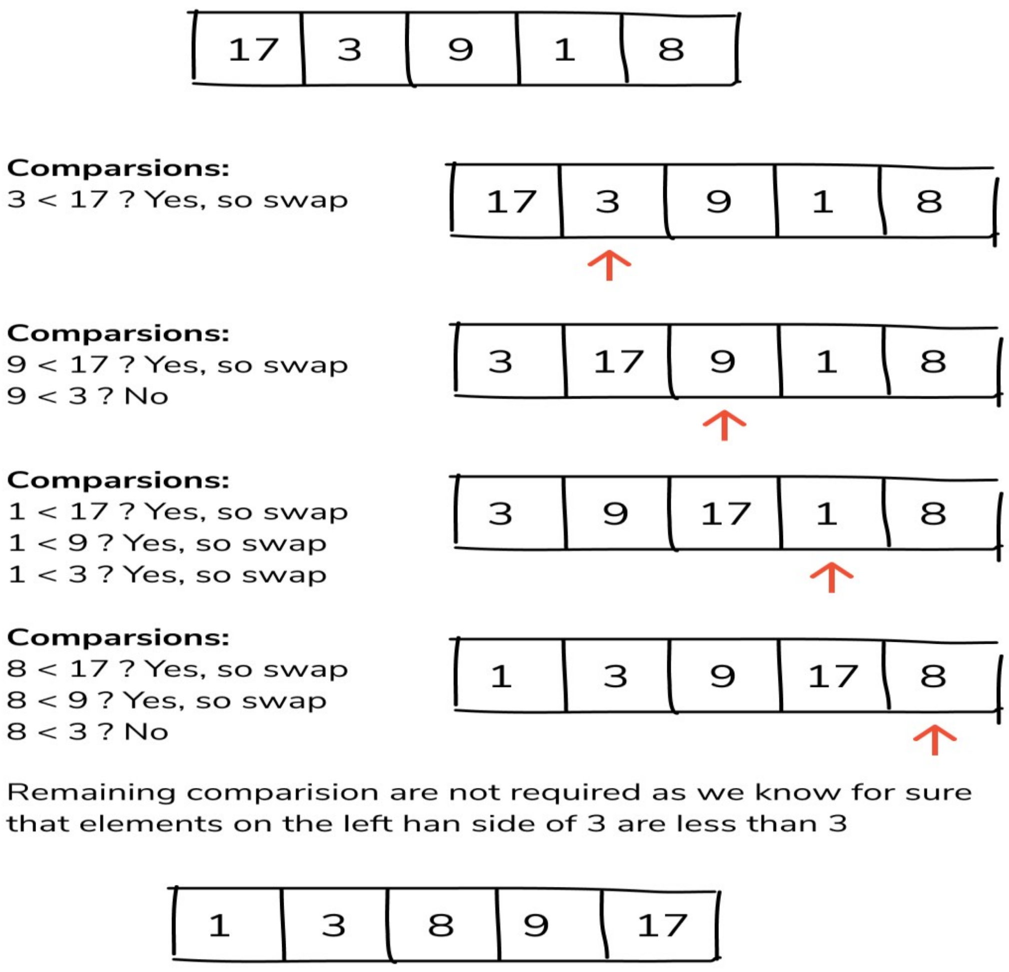
To sort an unsorted list with **'n'** number of elements, following are the complexities...

Worst Case : O(n log n)
Best Case : O(n log n)
Average Case : O(n log n)

2.2.8 SHELL SORT

Shell sort algorithm is **very similar to that of the Insertion sort algorithm**. In case of Insertion sort, we move elements one position ahead to insert an element at its correct position. Whereas here, Shell sort starts by sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared. Starting with far apart elements, it can move some out-of-place elements into the position faster than a simple nearest-neighbor exchange.

Here is an example to help you understand the working of Shell sort on array of elements name A = {17, 3, 9, 1, 8}


Comparsions:
3 < 17 ? Yes, so swap

Comparsions:
9 < 17 ? Yes, so swap
9 < 3 ? No

Comparsions:
1 < 17 ? Yes, so swap
1 < 9 ? Yes, so swap
1 < 3 ? Yes, so swap

Comparsions:
8 < 17 ? Yes, so swap
8 < 9 ? Yes, so swap
8 < 3 ? No

Remaining comparision are not required as we know for sure that elements on the left han side of 3 are less than 3

Algorithm for shell sort

```
Shell_Sort(Arr, n)

Step 1: SET FLAG = 1, GAP_SIZE = N
Step 2: Repeat Steps 3 to 6 while FLAG = 1 OR GAP_SIZE > 1
Step 3:     SET FLAG = 0
Step 4:     SET GAP_SIZE = (GAP_SIZE + 1) / 2
Step 5:     Repeat Step 6 for I = 0 to I < (N - GAP_SIZE)
Step 6:         IF Arr[I + GAP_SIZE] >Arr[I]
                    SWAP Arr[I + GAP_SIZE], Arr[I]
                    SET FLAG = 0
Step 7: END
```

Advantages:

- Shell sort algorithm is only efficient for finite number of elements in an array.
- Shell sort algorithm is 5.32 x faster than bubble sort algorithm.

Disadvantages:

- Shell sort algorithm is complex in structure and bit more difficult to understand.
- Shell sort algorithm is significantly slower than the merge sort, quick sort and heap sort algorithms.

Program

```
#include <stdio.h>
void shellsort(int arr[], int num)
{
int i, j, k, tmp;
for (i = num / 2; i > 0; i = i / 2)
{
for (j = i; j < num; j++)
{
for(k = j - i; k >= 0; k = k - i)
{
if (arr[k+i] >= arr[k])
break;
else
{
tmp = arr[k];
arr[k] = arr[k+i];
arr[k+i] = tmp;
}
}
}
}
}
int main()
{
        int arr[30];
        int k, num;
        printf("Enter total no. of elements : ");
        scanf("%d", &num);
        printf("\nEnter %d numbers: ", num);

        for (k =  0 ; k < num; k++)
        {
        scanf("%d", &arr[k]);
        }
        shellsort(arr, num);
        printf("\n Sorted array is: ");
        for (k = 0; k < num; k++)
        printf("%d ", arr[k]);
        return 0;
        }
```

Output:

```
Enter total no. of elements : 6

Enter 6 numbers: 3
2
4
10
2
1

Sorted array is: 1 2 2 3 4 10
```

Complexity of Shell Sort

Time complexity of above implementation of shellsort is $O(n^2)$. In the above implementation, gap is reduced by half in every iteration.

2.2.7 TABLE OF COMPARISON OF ALL SORTING TECHNIQUES

Time complexity of various sorting algorithms

S.No.	Algorithm	Worst Case	Average Case	Best Case
1.	**Selection Sort**	O (n^2)	O (n^2)	O (n^2)
2.	**Bubble Sort**	O (n^2)	O (n^2)	O (n^2)
3.	**Insertion Sort**	O (n^2)	O (n^2)	n – 1
4.	**Quick Sort**	O (n^2)	$\log_2 n$	$\log_2 n$
5.	**Heap Sort**	O (n log n)	O (n log n)	O (n log n)
6.	**Merge Sort**	O (n log n)	O (n log n)	O (n log n)
7.	**Radix Sort**	O (n^2)	O (n log n)	O (n log n)
8.	**Shell sort**	O(n log2 n)	–	–

2.3 SUMMERY

Sorting deals with sorting the data stored in the memory, whereas external sorting deals with sorting the data stored in files.

In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other.

Insertion sort works by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in the correct place.

Selection sort works by finding the smallest value and placing it in the first position. It then finds the second smallest value and places it in the second position. This procedure is repeated until the whole array is sorted.

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm. *Divide* means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements in each sub-array. *Conquer* means sorting the two sub-arrays recursively using merge sort. *Combine* means merging the two sorted sub-arrays of size n/2 each to produce a sorted array of n elements. The running time of merge sort in average case and worst case can be given as O(n log n).

Quick sort works by using a divide-and-conquer strategy. It selects a pivot element and rearranges the elements in such a way that all elements less than pivot appear before it and all elements greater than pivot appear after it.

Radix sort is a linear sorting algorithm that uses the concept of sorting names in alphabetical order.

Heap sort sorts an array in two phases. In the first phase, it builds a heap of the given array. In the second phase, the root element is deleted repeatedly and inserted into an array.

Shell sort is considered as an improvement over insertion sort, as it compares elements separated by a gap of several positions.

2.4 MODEL QUESTIONS

1. Define sorting. What is the importance of sorting?
2. What are the different types of sorting techniques? Which sorting technique has the least worst case?
3. Explain the difference between bubble sort and quick sort. Which one is more efficient?
4. Sort the elements 77, 49, 25, 12, 9, 33, 56, 81 using
   (a) insertion sort (b) selection sort
   (b) bubble sort (d) merge sort
   (e) quick sort (f) radix sort
   (g) shell sort
5. Compare heap sort and quick sort.

2.5 LIST OF REFERENCES

https://www.javatpoint.com/
https://www.studytonight.com
https://www.tutorialspoint.com
https://www.geeksforgeeks.org/heap-sort/
https://www.programiz.com/dsa/heap-sort
https://www.2braces.com/data-structures