UNIT V

Sorting: Bubble, Selection, Insertion sort, Quick sort, Merge sort, Heap sort, shell sort.

Searching Techniques: Linear Search, Binary Search

5.1 SORTINGS

5.1.1 Bubble Sort

Q1. Explain the working of bubble sort with an example.

Ans : **(Imp.)**

Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are not in the intended order.

Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

Algorithm

In the algorithm given below, suppose arr is an array of n elements. The assumed swap function in the algorithm will swap the values of given array elements.

```
begin BubbleSort(arr)
    for all array elements
        if arr[i] > arr[i+1]
                swap(arr[i], arr[i+1])
            end if
        end for
    return arr
end BubbleSort
```

Working of Bubble sort Algorithm

To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is $O(n^2)$.

Let the elements of array are:

13	32	26	35	10

First Pass

Sorting will start from the initial two elements. Let compare them to check which is greater.

13	32	26	35	10

Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.

13	32	26	35	10

Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -

13	26	32	35	10

Now, compare 32 and 35.

13	26	32	35	10

Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

Now, the comparison will be in between 35 and 10.

13	26	32	35	10

Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be:-

13	26	32	10	35

Now, move to the second iteration.

Second Pass

The same process will be followed for second iteration.

13	26	32	10	35

13	26	32	10	35

13	26	32	10	35

Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

13	26	10	32	35

13	26	10	32	35

Now, move to the third iteration.

Third Pass

The same process will be followed for third iteration.

13	26	10	32	35

13	26	10	32	35

Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -

13	10	26	32	35

13	10	26	32	35

13	10	26	32	35

Now, move to the fourth iteration.

Fourth pass

Similarly, after the fourth iteration, the array will be:-

10	13	26	32	35

Hence, there is no swapping required, so the array is completely sorted.

Q2. Write a program to implement bubble sort in C++ language.

Ans :

```
#include<iostream>
using namespace std;
   void print(int a[], int n)
      //function to print array elements
   {
   int i;
   for(i = 0; i < n; i++)
   {
      cout<<a[i]<<" ";
   }
   }
   void bubble(int a[], int n)
   // function to implement bubble sort
{
int i, j, temp;
   for(i = 0; i < n; i++)
   {
      for(j = i+1; j < n; j++)
            {
         if(a[j] < a[i])
         {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
         }
      }
   }
    }
```

```
int main()
{
  int i, j,temp;
  int a[5] = {45, 1, 32, 13, 26};
  int n = sizeof(a)/sizeof(a[0]);
  cout<<"Before sorting array elements are - \n";
  print(a, n);
  bubble(a, n);
  cout<<"\nAfter sorting array elements are - \n";
  print(a, n);
  return 0;
}
```

Output

Before sorting array elements are-

45 1 32 13 26

After sorting array elements are -

1 13 26 32 45

5.1.2 Selection Sort

Q3. Explain the working of selection sort with an example.

Ans : **(Imp.)**

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

Algorithm

SELECTION SORT(arr, n)

Step 1: Repeat Steps 2 and 3 for i = 0 to n-1

Step 2: CALL SMALLEST(arr, i, n, pos)

Step 3: SWAP arr[i] with arr[pos]

[END OF LOOP]

Step 4: EXIT

SMALLEST (arr, i, n, pos)

Step 1: [INITIALIZE] SET SMALL = arr[i]

Step 2: [INITIALIZE] SET pos = i

Step 3: Repeat for j = i+1 to n

if (SMALL > arr[j])

SET SMALL = arr[j]

SET pos = j

[END OF if]

[END OF LOOP]

Step 4: RETURN pos

Working of Selection sort Algorithm

To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are:

12	29	25	8	32	17	40

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

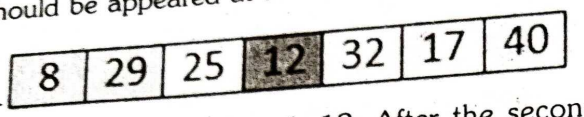
At present, 12 is stored at the first position, after searching the entire array, it is found that 8 is the smallest value.

12	29	25	8	32	17	40

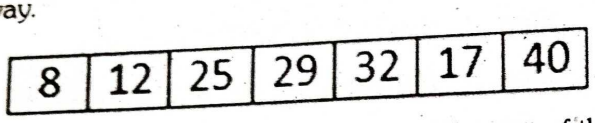
So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

8	29	25	12	32	17	40

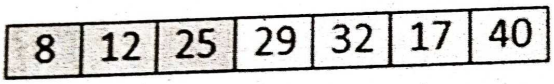
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

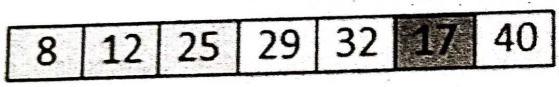


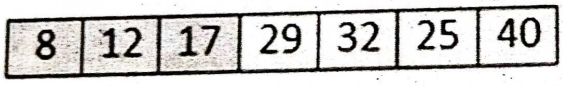
Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.

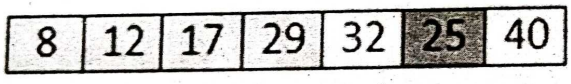


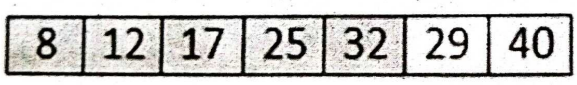
The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.

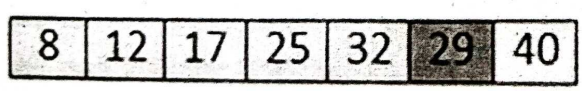


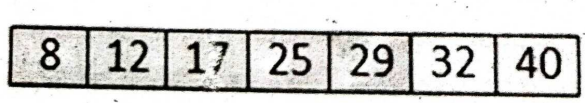












Now, the array is completely sorted.

Q4. Write a program to implement selection sort in C++ language. (Imp.)

Ans :

```
#include <iostream>
using namespace std;
    void selection(int arr[], int n)
{
    int i, j, small;
    for (i = 0; i < n-1; i++)
    // One by one move boundary of unsorted
      subarray
    {
    small = i; //minimum element in unsorted
                 array
        for (j = i+1; j < n; j++)
    if (arr[j] < arr[small])
          small = j;
    // Swap the minimum element with the first
      element
    int temp = arr[small];
     arr[small] = arr[i];
      arr[i] = temp;
    }
}
    void printArr(int a[], int n) /* function to
print the array */
{
    int i;
    for (i = 0; i < n; i++)
        cout<< a[i] <<" ";
}
  int main()
{
    int a[] = {80, 10, 29, 11, 8, 30, 15};
    int n = sizeof(a) / sizeof(a[0]);
    cout<<"Before sorting array elements are -
  "<<endl;
```

```
    printArr(a, n);
    selection(a, n);
    cout<<  "\nAfter sorting array elements
            are - "<<endl;
    printArr(a, n);
    return 0;
}
```

Output

After the execution of above code, the output will be :

Before sorting array elements are

80 10 29 11 8 30 15

After sorting array elements are

8 10 11 15 29 30 80

1.3 Insertion Sort

Q5. Explain the working of insertion sort with an example.

(Imp.)

Ans :

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put at their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array.

Algorithm

The simple steps of achieving the insertion sort are listed as follows:-

Step 1

If the element is the first element, assume that it is already sorted. Return 1.

Step 2

Pick the next element, and store it separately in a key.

Step 3

Now, compare the **key** with all elements in the sorted array.

Step 4

If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5

Insert the value.

Step 6

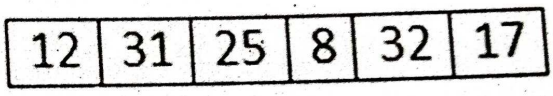
Repeat until the array is sorted.

Working of Insertion sort Algorithm

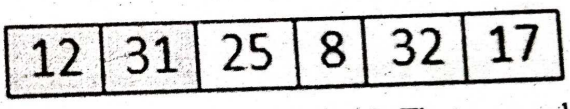
Now, let's see the working of the insertion sort Algorithm.

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

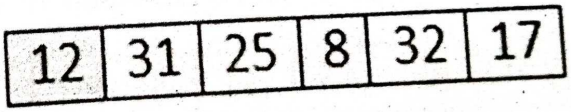
Let the elements of array are:



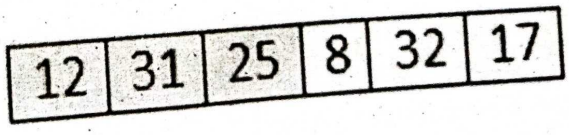
Initially, the first two elements are compared in insertion sort.

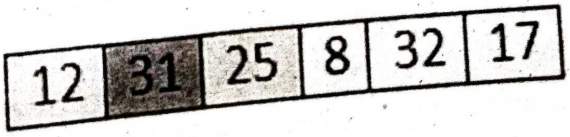


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



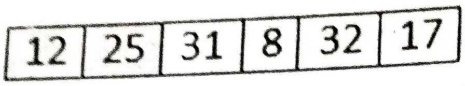
Now, move to the next two elements and compare them.



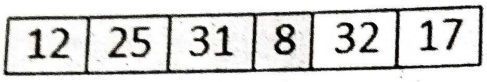


Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

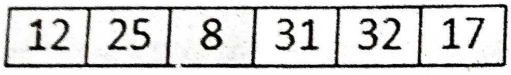
For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.



Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

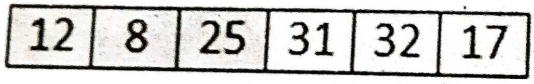


Both 31 and 8 are not sorted. So, swap them.

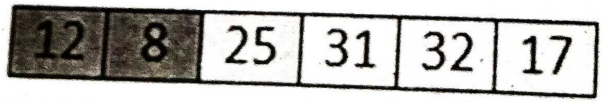


After swapping, elements 25 and 8 are unsorted.

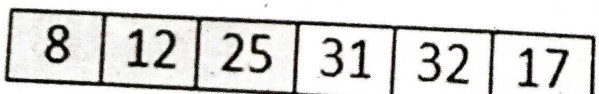
So, swap them.



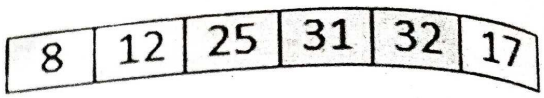
Now, elements 12 and 8 are unsorted.



So, swap them too.

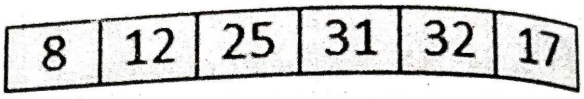


Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

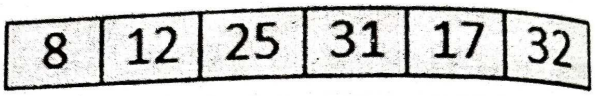


Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

Move to the next elements that are 32 and 17.



17 is smaller than 32. So, swap them.



Swapping makes 31 and 17 unsorted. So, swap them too.

Now, swapping makes 25 and 17 unsorted. So, perform swapping again.

Now, the array is completely sorted.

Q6. Write a program to implement insertion sort in C++ language.

Ans :

```
#include <iostream>
using namespace std;
void insert(int a[], int n) /* function to sort an aay with insertion sort */
```

```
int i, j, temp;
for (i = 1; i < n; i++) {
    temp = a[i];
    j = i - 1;
    while(j>=0 && temp <= a[j]) /* Move the elements greater than temp to one position ahead from their current position*/
    {
        a[j+1] = a[j];
        j = j-1;
    }
    a[j+1] = temp;
}

void printArr(int a[], int n) /* function to print the array */

int i;
for (i = 0; i < n; i++)
    cout << a[i] <<" ";

int main()

int a[] = { 89, 45, 35, 8, 12, 2 };
int n = sizeof(a) / sizeof(a[0]);
cout<<"Before sorting array elements are "<<endl;
printArr(a, n);
insert(a, n);
cout<<"\nAfter sorting array elements are "<<endl;
printArr(a, n);
return 0;
}
```

Output:

```
Before sorting array elements are-
89  45  35  9  12  2
After sorting array elements are
2  0  12  35  45  09
```

5.1.4 Quick Sort

Q7. Explain the working of quick sort with an example.

Ans : **(Imp.)**

Quicksort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide

In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer

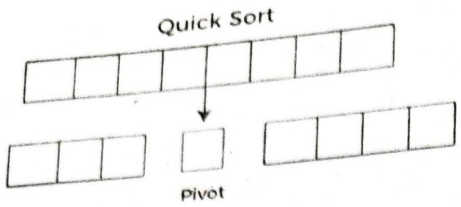
Recursively, sort two subarrays with Quicksort.

Combine

Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows:

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Algorithm:

```
QUICKSORT (array A, start, end)
{
 if (start < end)
 {
 p = partition(A, start, end)
 QUICKSORT (A, start, p - 1)
 QUICKSORT (A, p + 1, end)
 }
}
```

Partition Algorithm:

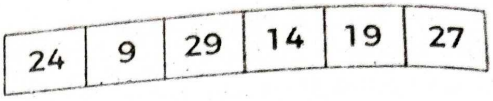
The partition algorithm rearranges the sub-arrays in a place.

```
PARTITION (array A, start, end)
{
  pivot ? A[end]
  i ? start-1
 for j ? start to end -1 {
  do if (A[j] < pivot) {
 then i ? i + 1
 swap A[i] with A[j]
  }}
  swap A[i+1] with A[end]
  return i+1
}
```

Working of Quick Sort Algorithm

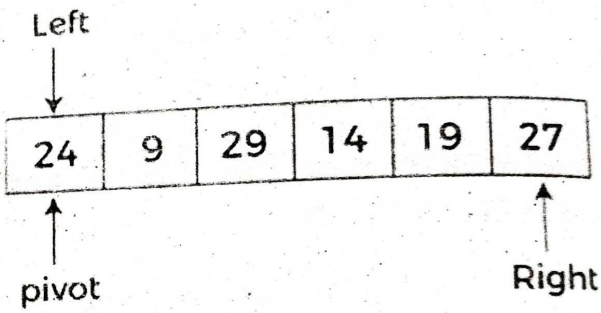
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are:

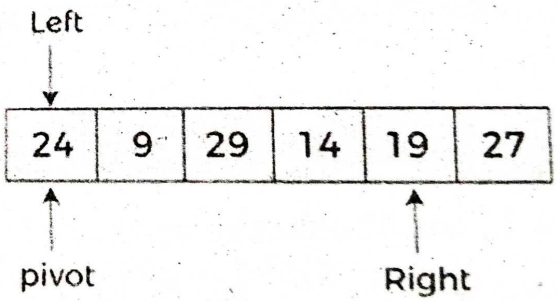


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

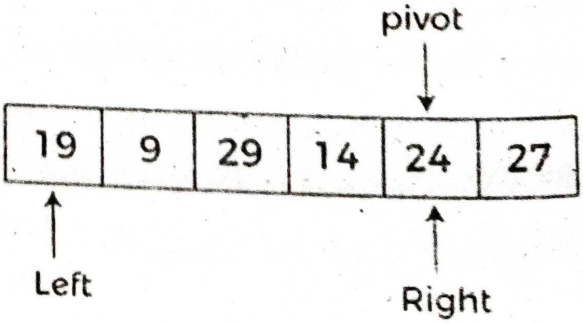


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e



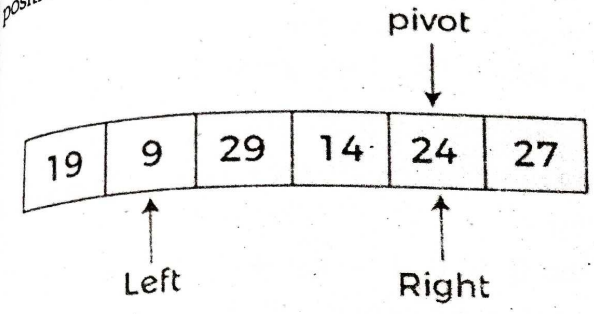
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

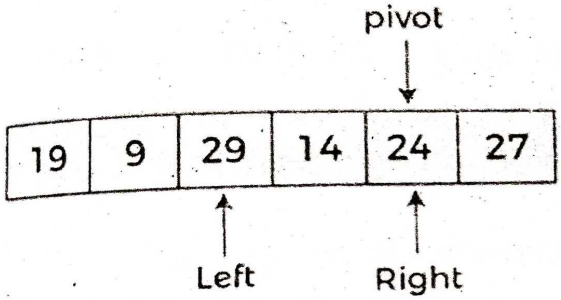


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

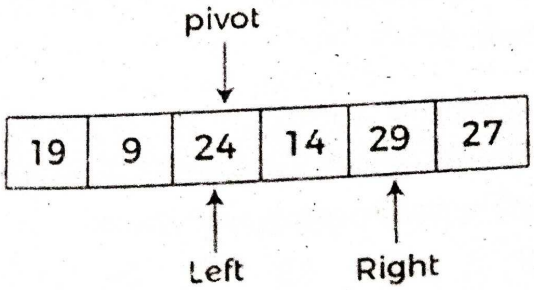
As a[pivot] > a[left], so algorithm moves one position to right as:



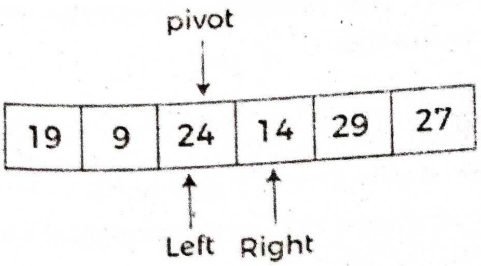
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as,



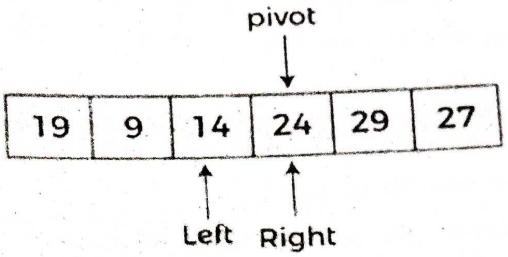
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e..



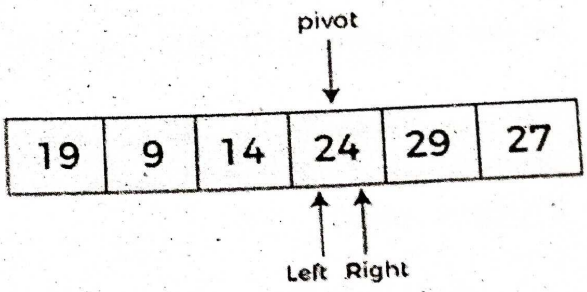
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as,



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



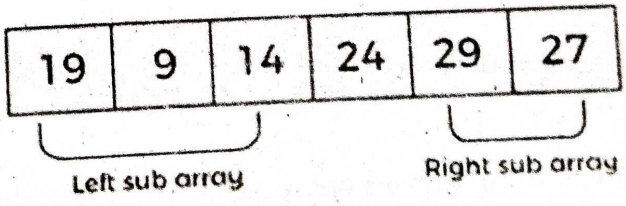
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



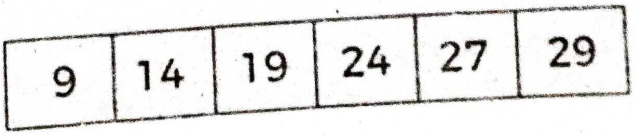
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



Q8. Write a program to implement quick sort in C++ language.

Ans :

```
#include <iostream>
  using namespace std;
    /* function that consider last element as pivot,
place the pivot at its exact position, and place
smaller elements to left of pivot and greater
elements to right of pivot. */
int partition (int a[], int start, int end)
{
    int pivot = a[end]; // pivot element
    int i = (start - 1);

    for (int j = start; j <= end - 1; j++)
    {
    // If current element is smaller than the
    pivot
    if (a[j] < pivot)
    {
    i++; // increment index of smaller element
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    int t = a[i+1];
    a[i+1] = a[end];
    a[end] = t;
    return (i + 1);
}
/* function to implement quick sort */
void quick(int a[], int start, int end) /* a[] = array
to be sorted, start = Starting index, end =
Ending index */
{
    if (start < end)
    {
        int p = partition(a, start, end);
        //p is the partitioning index
        quick(a, start, p - 1);
        quick(a, p + 1, end);
    }
}
/* function to print an array */
void printArr(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        cout<<a[i]<< " ";
}
int main()
{
    int a[] = { 23, 8, 28, 13, 18, 26 };
    int n = sizeof(a) / sizeof(a[0]);
    cout<<"Before sorting array elements are -
\n";
    printArr(a, n);
    quick(a, 0, n - 1);
    cout<<"\nAfter sorting array elements are -
\n";
    printArr(a, n);
    return 0;
}
```

Output:

Before sorting array elements are-

23 8 28 13 18 26

After sorting array elements are-

8 13 18 23 26 28

5.1.5 Merge Sort

Q9. Explain the working of merge sort with an example.

Ans : **(Imp.)**

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort

the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Algorithm

In the following algorithm, arr is the given array, beg is the starting element, and end is the last element of the array.

```
MERGE_SORT(arr, beg, end)
if beg < end
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if
END MERGE_SORT
```

The important part of the merge sort is the MERGE function. This function performs the merging of two sorted sub-arrays that are A[beg...mid] and A[mid+1...end], to build one sorted array A[beg...end]. So, the inputs of the MERGE function are A[], beg, mid, and end.

The implementation of the MERGE function is given as follows -

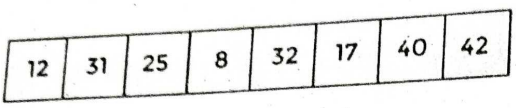
```
/* Function to merge the subarrays of a[] */
void merge(int a[], int beg, int mid, int end)
{
    int i, j, k;
    int n1 = mid - beg + 1;
    int n2 = end - mid;
    int LeftArray[n1], RightArray[n2]; // temporary arrays
/* copy data to temp arrays */
for (int i = 0; i < n1; i++)
LeftArray[i] = a[beg + i];
for (int j = 0; j < n2; j++)
RightArray[j] = a[mid + 1 + j];
i = 0, /* initial index of first sub-array */
j = 0; /*initial index of second sub-
        array */
k = beg; /* initial index of merged sub-
        array */
while (i < n1 && j < n2)
{
    if(LeftArray[i] <= RightArray[j])
    {
        a[k] = LeftArray[i];
        i++;
    }
    else
    {
        a[k] = RightArray[j];
        j++;
    }
    k++;
}
while (i<n1)
{
    a[k] = LeftArray[i];
    i++;
    k++;
}

while (j<n2)
{
    a[k] = RightArray[j];
    j++;
    k++;
}
}
```

Working of Merge sort Algorithm

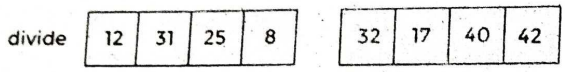
To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are,

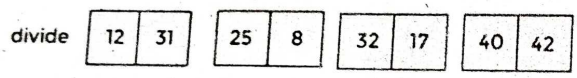


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

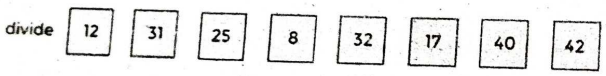
As there are eight elements in the given array, so it is divided into two arrays of size 4.



Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.



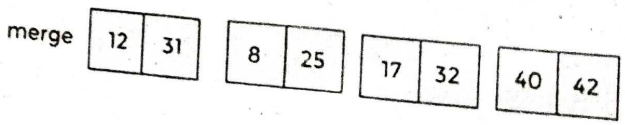
Now, again divide these arrays to get the atomic value that cannot be further divided.



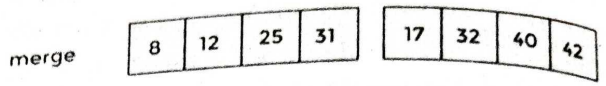
Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

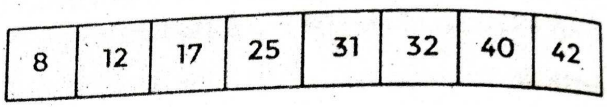
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

Q10. Write a program to implement merge sort in C++ language.

Ans :

```
#include <iostream>
using namespace std;
/* Function to merge the subarrays of a[] */
void merge(int a[], int beg, int mid, int end)
{
    int i, j, k;
    int n1 = mid - beg + 1;
    int n2 = end - mid;
    int LeftArray[n1], RightArray[n2];
        //temporary arrays
        /* copy data to temp arrays */
    for (int i = 0; i < n1; i++)
    LeftArray[i] = a[beg + i];
    for (int j = 0; j < n2; j++)
    RightArray[j] = a[mid + 1 + j];
    i = 0; /*initial index of first sub-array */
    j = 0; /*initial index of second sub-
            array */
    k = beg; /* initial index of merged sub-
            array */
    while (i< n1 && j < n2)
    {
        if(LeftArray[i] <= RightArray[j])
        {
```

```
            a[k] = LeftArray[i];
            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}
void mergeSort(int a[], int beg, int end)
{
    if (beg < end)
    {
        int mid = (beg + end) / 2;
        mergeSort(a, beg, mid);
        mergeSort(a, mid + 1, end);
        merge(a, beg, mid, end);
    }
}
 /* Function to print the array */
void printArray(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        cout<<a[i]<<" ";
}
int main()
{
    int a[] = {11, 30, 24, 7, 31, 16, 39, 41};
    int n = sizeof(a) / sizeof(a[0]);
    cout<<"Before sorting array elements are -
        \n";
    printArray(a, n);
    mergeSort(a, 0, n - 1);
    cout<<"\nAfter sorting array elements are -
            \n";
    printArray(a, n);
    return 0;
}
```

Output:

```
Before sorting array elements are-
11 30 24 7 31 16 39 41
After sorting array elements are-
7 11 16 24 30 31 39 41
```

5.1.6 Heap Sort

Q11. What is heap sort?Explain the working of heap sort with an example.

(Imp.)

Ans :

Heapsort is a popular and efficient sorting algorithm. The concept of 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.

Heapsort is the in-place sorting algorithm.

Algorithm

```
HeapSort(arr)
BuildMaxHeap(arr)
for i = length(arr) to 2
    swap arr[1] with arr[i]
    heap_size[arr] = heap_size[arr] ? 1
    MaxHeapify(arr,1)
End
```

```
BuildMaxHeap(arr)
    BuildMaxHeap(arr)
        heap_size(arr) = length(arr)
        for i = length(arr)/2 to 1
    MaxHeapify(arr,i)
    End
MaxHeapify(arr,i)
    MaxHeapify(arr,i)
    L = left(i)
    R = right(i)
    if L ? heap_size[arr] and arr[L] > arr[i]
    largest = L
    else
    largest = i
    if R ? heap_size[arr] and arr[R] >
        arr[largest]
    largest = R
    if largest != i
    swap arr[i] with arr[largest]
    MaxHeapify(arr,largest)
    End
```

Working of Heap sort Algorithm

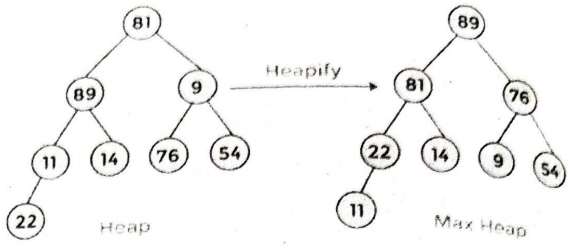
In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows.

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

Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.

81	89	9	11	14	76	54	22

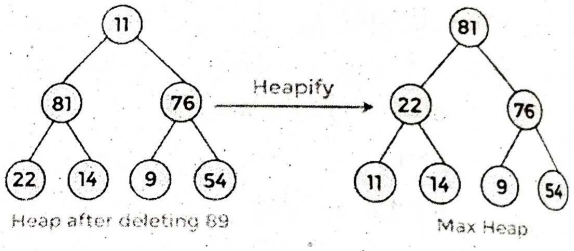
First, we have to construct a heap from the given array and convert it into max heap.



After converting the given heap into max heap, the array elements are -

89	81	76	22	14	9	54	11

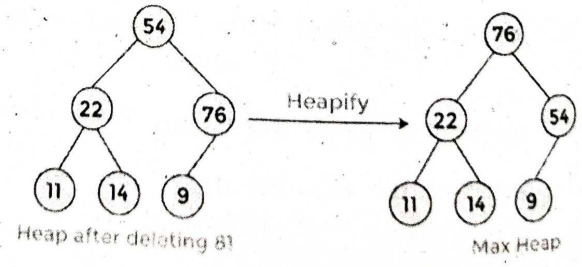
Next, we have to delete the root element (89) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 89 with 11, and converting the heap into max-heap, the elements of array are,

81	22	76	11	14	9	54	89

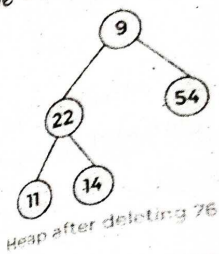
In the next step, again, we have to delete the root element (81) from the max heap. To delete this node, we have to swap it with the last node, i.e. (54). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping t
and converting the
elements of array are

76	22	54	1

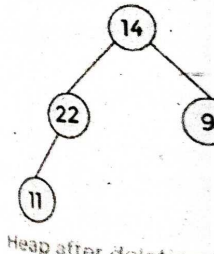
In the next step
element (76) from th
this node, we have t
i.e. (9). After deletin
have to heapify it to



After swapp
with 9 and conver
the elements of arra

54	22	9

In the next st
root element (54)
this node, we have
i.e. (14). After del
have to heapify it t



After swapp
and converting
elements of array

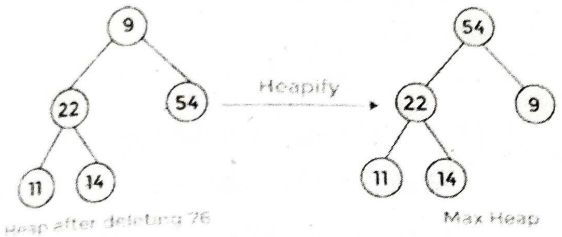
22	14	9

In the next
root element (22

After swapping the array element 81 with 54 and converting the heap into max-heap, the elements of array are

76	22	54	11	14	9	81	89

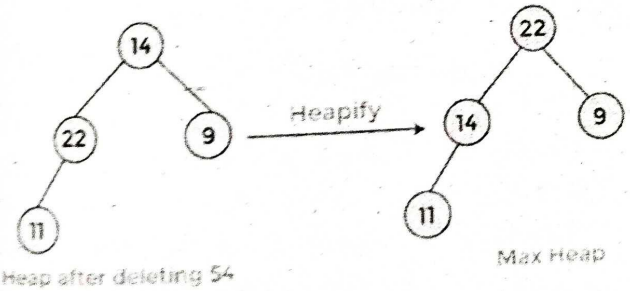
In the next step, we have to delete the root element (76) from the max heap again. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 76 with 9 and converting the heap into max-heap, the elements of array are,

54	22	9	11	14	76	81	89

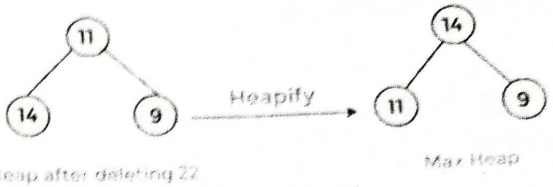
In the next step, again we have to delete the root element (54) from the max heap. To delete this node, we have to swap it with the last node, i.e. (14). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 54 with 14 and converting the heap into max-heap, the elements of array are.

22	14	9	11	54	76	81	89

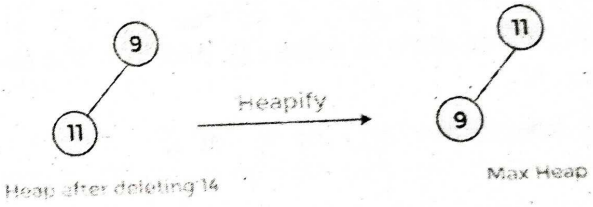
In the next step, again we have to delete the root element (22) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 22 with 11 and converting the heap into max-heap, the elements of array are,

14	11	9	22	54	76	81	89

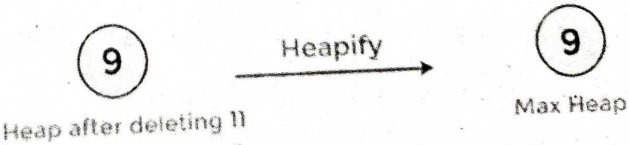
In the next step, again we have to delete the root element (14) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 14 with 9 and converting the heap into max-heap, the elements of array are

11	9	14	22	54	76	81	89

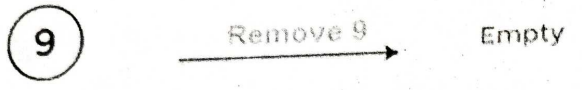
In the next step, again we have to delete the root element (11) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 11 with 9, the elements of array are,

9	11	14	22	54	76	81	89

Now, heap has only one element left. After deleting it, heap will be empty.



After completion of sorting, the array elements are

9	11	14	22	54	76	81	89

Now, the array is completely sorted.

Q12. Write a program to implement heap sort in C++.

Ans : **(Imp.)**

```
#include <iostream>
using namespace std;
/* function to heapify a subtree. Here 'i' is the index of root node in array a[], and 'n' is the size of heap. */
void heapify(int a[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && a[left] > a[largest])
        largest = left;
    // If right child is larger than root
    if (right < n && a[right] > a[largest])
        largest = right;
    // If root is not largest
    if (largest != i) {
        // swap a[i] with a[largest]
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;
        heapify(a, n, largest);
    }
}
/*Function to implement the heap sort*/
void heapSort(int a[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i—)
        heapify(a, n, i);
    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i—) {
        /* Move current root element to end*/
        // swap a[0] with a[i]
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;
        heapify(a, i, 0);
    }
}
/* function to print the array elements */
void printArr(int a[], int n)
{
    for (int i = 0; i < n; ++i)
    {
        cout<<a[i]<<" ";
    }

}
int main()
{
    int a[] = {47, 9, 22, 42, 27, 25, 0};
    int n = sizeof(a) / sizeof(a[0]);
    cout<<"Before sorting array elements are - \n";
    printArr(a, n);
```

```
heapSort(a, n);
cout<<"\nAfter sorting array elements are \n";
printArr(a, n);
return 0;
```

Output

```
Before sorting array elements are-
47 9 22 42 27 25 0
After sorting array elements are-
0 9 22 25 27 42 47
```

3.1.7 Shell Sort

Q13. Explain the working of shell sort with an example.

Ans : **(Imp.)**

It is a sorting algorithm that is an extended version of insertion sort. Shell sort has improved the average time complexity of insertion sort. As similar to insertion sort, it is a comparison-based and in-place sorting algorithm. Shell sort is efficient for medium-sized data sets.

In insertion sort, at a time, elements can be moved ahead by one position only. To move an element to a far-away position, many movements are required that increase the algorithm's execution time. But shell sort overcomes this drawback of insertion sort. It allows the movement and swapping of far-away elements as well.

This algorithm first sorts the elements that are far away from each other, then it subsequently reduces the gap between them. This gap is called the interval. This interval can be calculated by using the Knuth's formula given below.

hh = h * 3 + 1 where, 'h' is the interval having initial value 1.

Algorithm

```
ShellSort(a, n) // 'a' is the given array, 'n' is the size of array
for (interval = n/2; interval > 0; interval / 2)
for ( i = interval; i < n; i += 1)
temp      = a[i];
for (j = i; j >= interval &&
a[j - interval] > temp; j -= interval)
a[j] = a[j - interval];
a[j] = temp;
End ShellSort
```

Working of Shell Sort Algorithm

Now, let's see the working of the shell sort Algorithm.

To understand the working of the shell sort algorithm, let's take an unsorted array. It will be easier to understand the shell sort via an example.

Let the elements of array are,

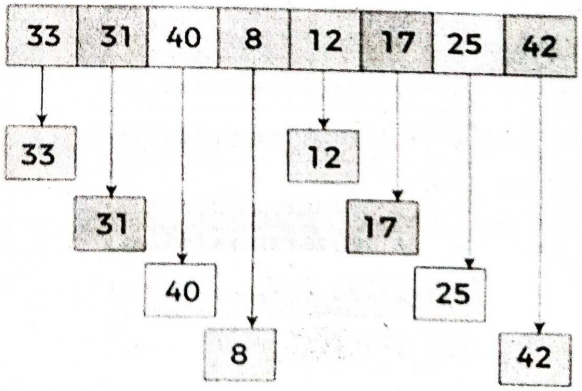
33	31	40	8	12	17	25	42

We will use the original sequence of shell sort, i.e., N/2, N/4,....,1 as the intervals.

In the first loop, n is equal to 8 (size of the array), so the elements are lying at the interval of 4 (n/2 = 4). Elements will be compared and swapped if they are not in order.

Here, in the first loop, the element at the 0^{th} position will be compared with the element at 4^{th} position. If the 0^{th} element is greater, it will be swapped with the element at 4^{th} position. Otherwise, it remains the same. This process will continue for the remaining elements.

At the interval of 4, the sublists are {33, 12}, {31, 17}, {40, 25}, {8, 42}.

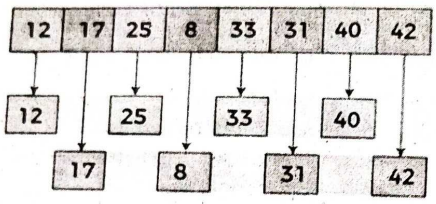


Now, we have to compare the values in every sub-list. After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows -

12	17	25	8	33	31	40	42

In the second loop, elements are lying at the interval of 2 (n/4 = 2), where n = 8.

Now, we are taking the interval of 2 to sort the rest of the array. With an interval of 2, two sublists will be generated - {12, 25, 33, 40}, and {17, 8, 31, 42}.



Now, we again have to compare the values in every sub-list. After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows.

12	8	25	17	33	31	40	42

In the third loop, elements are lying at the interval of 1 (n/8 = 1), where n = 8. At last, we use the interval of value 1 to sort the rest of the array elements. In this step, shell sort uses insertion sort to sort the array elements.

12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

Now, the array is sorted in ascending order.

Q14. Write a program to implement Shell sort in C++. (Imp.)

Ans :

```
#include <iostream>
using namespace std;
/* function to implement shellSort */
int shell(int a[], int n)
{
    /* Rearrange the array elements at n/2, n/4, ..., 1 intervals */
    for (int interval = n/2; interval > 0; interval /= 2)
    {
    for (int i = interval; i < n; i += 1)
    {
    /* store a[i] to the variable temp and make the ith position empty */
            int temp = a[i];
            int j;
    for (j = i; j >= interval &&
    a[j - interval] > temp; j -= interval)
    a[j] = a[j - interval];
     // put temp (the original a[i]) in its correct position
            a[j] = temp;
        }
    }
    return 0;
}
    void printArr(int a[], int n)
    /* function to print the array elements */
{
    int i;
    for (i = 0; i < n; i++)
        cout<<a[i]<<" ";
}
int main()
```

```
int a[] = { 32, 30, 39, 7, 11, 16, 24, 41 };
int n = sizeof(a)/sizeof(a[0]);
cout<<"Before sorting array elements are -
      \n";
printArr(a, n);
shell(a, n);
cout<<"\nAfter applying shell sort, the
      array elements are - \n";
printArr(a, n);
return 0;
```

Output

After the execution of the above code, the output will be,

Before sorting array elements are-

32 30 39 7 11 16 24 41

After sorting array elements are-

7 11 16 24 30 32 39 41

5.2 Searching Techniques

5.2.1 Linear Search

Q15. Explain the working of linear search with example.

(Imp.)

Ans :

Linear search is also called as sequential search algorithm. It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

The steps used in the implementation of Linear Search are listed as follows.

- First, we have to traverse the array elements using a for loop.
- In each iteration of for loop, compare the search element with the current array element, and
- If the element matches, then return the index of the corresponding array element.
- If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return -1.

Algorithm

```
Linear_Search(a, n, val)
// 'a' is the given array, ' n' is the size of
given array, 'val' is the value to search
Step 1: set pos = -1
Step 2: set i = 1
Step 3: repeat step 4 while i <= n
Step 4: if a[i] == val
set pos = i
print pos
go to step 6
[end of if]
set ii = i + 1
[end of loop]
Step 5: if pos = -1
print "value is not present in the array "
[end of if]
Step 6: exit
```

Working of Linear search

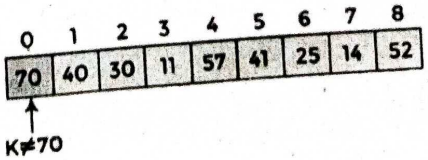
To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are,

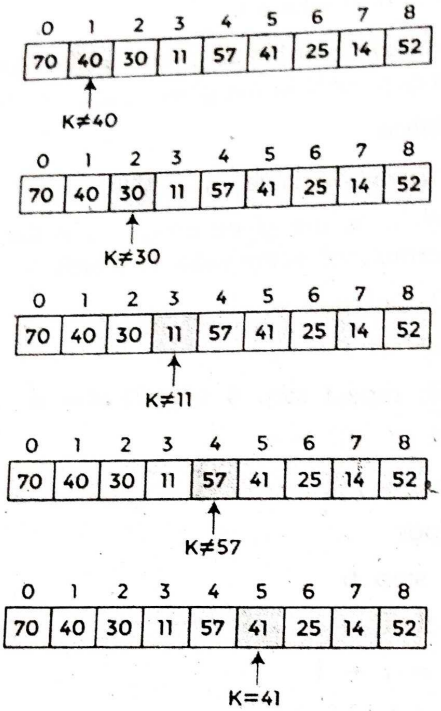
0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

Let the element to be searched is K = 41.

Now, start from the first element and compare K with each element of the array.



The value of K, i.e., 41, is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found



Now, the element to be searched is found. So algorithm will return the index of the element matched.

Q16. Write a program to implement linear search in C++.

Ans :

```
#include <iostream>
using namespace std;
int linearSearch(int a[], int n, int val) {
  // Going through array linearly
  for (int i = 0; i < n; i++)
    {
      if (a[i] == val)
      return i+1;
    }
  return -1;
}
int main() {
  int a[] = {69, 39, 29, 10, 56, 40, 24, 13, 51};
                    // given array
  int val = 56;     // value to be searched
  int n = sizeof(a) // sizeof(a[0]);
                    // size of array
  int res = linearSearch(a, n, val);
                    // Store result
    cout<<"The elements of the array are-";
    for (int i = 0; i < n; i++)
    cout<<a[i]<<" ";
    cout<<"\nElement to be searched is -
          "<<val;
    if (res == -1)
    cout<<"\nElement is not present in the
    array";
    else
    cout<<"\nElement is present at"<<
res<<"position of array";
  return 0;
}
```

Output

```
The elements of the array are
69 39 29 10 56 40 24 13 51
Element to be searched is 56
Element is present at 5 position of array
```

5.2.2 Binary Search

Q17. Explain the working of binary search with an example.

Ans : **(Imp.)**

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

Algorithm

Binary_Search(a, lower_bound, upper_bound, val) //

'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_ bound' is the index of the last arrayelement, 'val' is the value to search

Step 1: set beg = lower_bound, end = upper_bound, pos = - 1

Step 2: repeat steps 3 and 4 while beg <=end

Step 3: set mid = (beg + end)/2

Step 4: if a[mid] = val

set pos = mid

print pos

go to step 6

else if a[mid] > val

set end = mid - 1

else

set beg = mid + 1

[end of if]

[end of loop]

Step 5: if pos = -1

print "value is not present in the array"

[end of if]

Step 6: exit

Working of Binary Search

To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example.

There are two methods to implement the binary search algorithm.

- Iterative method
- Recursive method

The recursive method of binary search follows the divide and conquer approach.

Let the elements of array are.

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

Let the element to search is, K = 56

We have to use the below formula to calculate the mid of the array -

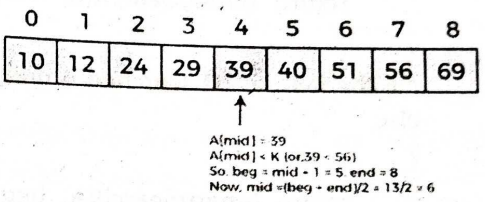
mid = (beg + end)/2

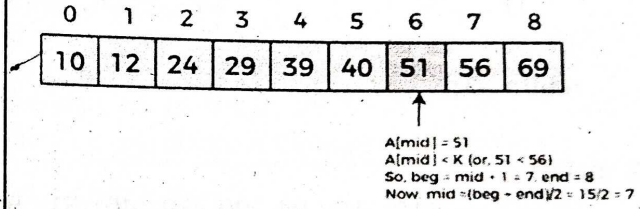
So, in the given array -

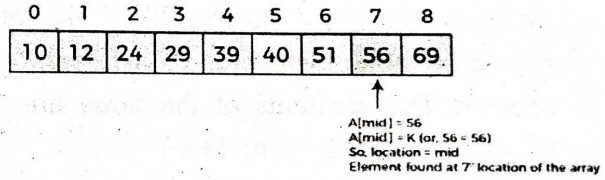
beg = 0

end = 8

mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.







Now, the element to search is found. So algorithm will return the index of the element matched.

Q18. Write a program to implement Binary search in C++.

Ans :

```
#include <iostream>
using namespace std;
int binarySearch(int a[], int beg, int end, int val)
{
   int mid;
   if(end >= beg)
   {
```

```
        mid = (beg + end)/2;
                /* if the item to be searched is present at middle */
            if(a[mid] == val)
            {
                return mid+1;
            }
            /* if the item to be searched is smaller than m iddle, then it can only be in left subarray */
            else if(a[mid] < val)
            {
                return binarySearch(a, mid+1, end, val);
            }
            /*if the item to be searched is greater than middle, then it can only be in right subarray */
            else
            {
                return binarySearch(a, beg, mid-1, val);
            }
        }
        return -1;
}
int main() {
    int a[] = {10, 12, 24, 29, 39, 40, 51, 56, 70}; // given array
    int val = 51; // value to be searched
    int n = sizeof(a) / sizeof(a[0]); // size of array
    int res = binarySearch(a, 0, n-1, val); // Store result
    cout<<"The elements of the array are - ";
    for (int i = 0; i < n; i++)
    cout<<a[i]<<" ";
    cout<<"\nElement to be searched is - "<<val;
    if (res == -1)
    cout<<"\nElement is not present in the array";
    else
    cout<<"\nElement is present at "<<res<<" position of array";
    return 0;
}
```

Output

The elements of the array are

10 12 24 29 39 40 51 56 70

Element to be searched is 51

Element is present at 7 position of array

Short Question and Answers

Bubble sort.

Ans :

Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are not in the intended order.

Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

Algorithm

In the algorithm given below, suppose arr is an array of n elements. The assumed swap function in the algorithm will swap the values of given array elements.

```
begin BubbleSort(arr)
    for all array elements
        if arr[i] > arr[i+1]
                swap(arr[i], arr[i+1])
            end if
        end for
    return arr
end BubbleSort
```

2. Selection sort

Ans :

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

3. Insertion sort

Ans :

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array.

Q4. Quick Sort

Ans :

Quicksort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide

In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer

Recursively, sort two subarrays with Quicksort.

Combine

Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

5. Merge Sort

Ans :

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

6. Heap sort.

Ans :

Heapsort is a popular and efficient sorting algorithm. The concept of 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.

Heapsort is the in-place sorting algorithm.

Algorithm

```
HeapSort(arr)
BuildMaxHeap(arr)
for i = length(arr) to 2
    swap arr[1] with arr[i]
    heap_size[arr] = heap_size[arr]?1
    MaxHeapify(arr,1)
End
```

```
BuildMaxHeap(arr)
    BuildMaxHeap(arr)
        heap_size(arr) = length(arr)
        for i = length(arr)/2 to 1
    MaxHeapify(arr,i)
    End
MaxHeapify(arr,i)
    MaxHeapify(arr,i)
    L = left(i)
    R = right(i)
    if L ? heap_size[arr] and arr[L] > arr[i]
    largest = L
    else
    largest = i if R ? heap_size[arr] and arr[R]
    > arr[largest]
    largest = R
    if largest != i
    swap arr[i] with arr[largest]
    MaxHeapify(arr,largest)
    End
```

7. Shell sort

Ans :

It is a sorting algorithm that is an extended version of insertion sort. Shell sort has improved the average time complexity of insertion sort. As similar to insertion sort, it is a comparison-based and in-place sorting algorithm. Shell sort is efficient for medium-sized data sets.

In insertion sort, at a time, elements can be moved ahead by one position only. To move an element to a far-away position, many movements are required that increase the algorithm's execution time. But shell sort overcomes this drawback of insertion sort. It allows the movement and swapping of far-away elements as well.

This algorithm first sorts the elements that are far away from each other, then it subsequently reduces the gap between them. This gap is called as interval. This interval can be calculated by using the Knuth's formula given below.

hh = h * 3 + 1 where, 'h' is the interval having initial value 1.

8. Linear searc

Ans :

Linear search
Linear search, we s
whose location is to
the algorithm return

The steps use

- First, we hav
- In each iterat
- If the elemen
- If the elemen
- If there is no

9. Binary sea

Ans :

Binary searc
element into some

Binary searc
and the item is con
middle element is
produced through

Algorithm

Binary_Sea
index of the first a
search.

10. Write a pr

Ans :

```
#include <iostrea
using namespace
int linearSearch(i
    // Going th
    for (int i =
    {
        if (a[i
        return
    }
    return -1;
}
```

8. Linear search

Ans :

Linear search is also called as sequential search algorithm. It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

The steps used in the implementation of Linear Search are listed as follows.

- First, we have to traverse the array elements using a for loop.
- In each iteration of for loop, compare the search element with the current array element, and
- If the element matches, then return the index of the corresponding array element.
- If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return-1.

9. Binary search

Ans :

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

Algorithm

Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_ bound' is the index of the last arrayelement, 'val' is the value to search.

10. Write a program to implement linear search in C++.

Ans :

```
#include <iostream>
using namespace std;
int linearSearch(int a[], int n, int val) {
    // Going through array linearly
    for (int i = 0; i < n; i++)
    {
        if (a[i] == val)
        return i+1;
    }
    return -1;
}
```

```
int main() {
    int a[] = {69, 39, 29, 10, 56, 40, 24, 13,  51};  // given array
    int val = 56;                                     // value to be searched
    int n = sizeof(a)                                 // sizeof(a[0]); size of array
    int res = linearSearch(a, n, val);                // Store result
    cout<<"The elements of the array are-";
    for (int i = 0; i < n; i++)
    cout<<a[i]<<" ";
    cout<<"\nElement to be searched is -"<<val;
    if (res == -1)
    cout<<"\nElement is not present in the array";
    else
    cout<<"\nElement is present at"<< res<<"position of array";
  return 0;
}
```

Output

The elements of the array are

69 39 29 10 56 40 24 13 51

Element to be searched is 56

Element is present at 5 position of array

Choose the Correct Answers

1. Which of the following is not a stable sorting algorithm? [b]
   (a) Insertion sort (b) Selection sort
   (c) Bubble sort (d) Merge sort

2. If the given input array is sorted or nearly sorted, which of the following algorithm gives the best performance? [a]
   (a) Insertion sort (b) Selection sort
   (c) Quick sort (d) Merge sort

3. Which of the following algorithm pays the least attention to the ordering of the elements in the input list? [b]
   (a) Insertion sort (b) Selection sort
   (c) Quick sort (d) None

4. The complexity of merge sort algorithm is _____ [d]
   (a) O(n) (b) O(logn)
   (c) O(n2) (d) O(n logn)

5. Which of the following sorting algorithm is of priority queue sorting type? [d]
   (a) Bubble sort (b) Insertion sort
   (c) Merge sort (d) Selection sort

6. What is the best case for linear search? [d]
   (a) O(nlogn) (b) O(logn)
   (c) O(n) (d) O(1)

7. Which of the following is not the required condition for a binary search algorithm? [c]
   (a) The list must be sorted
   (b) There should be direct access to the middle element in any sublist
   (c) There must be a mechanism to delete and/or insert elements in the list.
   (d) Number values should only be present

8. _____ is rearranging pairs of elements which are out of order, until no such pairs remain.[b]
   (a) Insertion (b) Exchange
   (c) Selection (d) Distribution

9. Binary Search can be categorized into which of the following? [b]
   (a) Brute Force technique (b) Divide and conquer
   (c) Greedy algorithm (d) Dynamic programming

10. Binary search algorithm cannot be applied to _____ [a]
    (a) Sorted linked list (b) Sorted binary trees
    (c) Sorted linear array (d) Pointer array

Fill in the Blanks

1. The complexity of the sorting algorithm measures the _____ as a function of the number n of items to be sorter.
2. The worst-case occur in linear search algorithm when _____
3. If the number of records to be sorted is small, then _____ sorting can be efficient.
4. Counting sort performs _____ Numbers of comparisons between input elements.
5. Merge sort uses _____ technique
6. The complexity of bubble sort algorithm is _____
7. _____ sort technique consumes less time to describe whether the input array is sorted or not.
8. Partition and exchange sort is _____
9. _____ sorting algorithm is frequently used when n is small where n is total number of elements.
10. _____ search is used When the list has only a few elements and When performing a single search in an unordered list

Answers

1. Running time
2. Item is the last element in the array or item is not there at all
3. Selection
4. 0
5. Divide and conquer
6. O(n2)
7. Bubble sort
8. Quick sort
9. Insertion
10. Linear search