Unit 5: Sorting Algorithms

Sorting:

The arrangement of data in a preferred order is called sorting in the data structure. By sorting data, it is easier to search through it quickly and easily. The simplest example of sorting is a dictionary. Before the era of the Internet, when you wanted to look up a word in a dictionary, you would do so in alphabetical order. This made it easy.

Imagine the panic if you had to go through a big book with all the English words from the world in a jumbled order! It is the same panic an engineer will go through if their data is not sorted and structured.

So, in short, sorting makes our lives easier.

Sorting Algorithm

A sorting algorithm is just a series of orders or instructions. In this, an array is an input, on which the sorting algorithm performs operations to give out a sorted array.

Here’s an example of what sorting does.

Let’s suppose you have an array of strings: [h,j,k,i,n,m,o,l]

Now, sorting would yield an output array in alphabetical order.

Output: [h,i,j,k,l,m,n,o]

1. Internal/External Sort, Stable/ Unstable Sort

External sorting and internal sorting are two approaches to sorting data, based on where the sorting process takes place and how the data is handled during the sorting operation.

1. **Internal Sorting**: Internal sorting refers to the process of sorting data that can fit entirely within the main memory (RAM) of a computer system. In internal sorting, the entire dataset to be sorted is loaded into memory, and the sorting algorithm operates on the data within the memory. The main advantage of internal sorting is that it allows for faster sorting operations since accessing data in memory is much faster compared to accessing data from external storage devices like hard disks.

Common internal sorting algorithms include:

- Quicksort
- Mergesort
- Heapsort
- Insertion sort
- Selection sort
- Bubble sort

Internal sorting is typically used when the dataset is relatively small and can fit comfortably within the available memory.

2. **External Sorting**: External sorting is used when the dataset to be sorted is too large to fit entirely in memory. It involves sorting data that is stored on external storage devices such as hard disks, solid-state drives (SSDs), or tapes. External sorting techniques efficiently manage the limited capacity of the main memory by dividing the dataset into smaller blocks that can be processed in memory.

The external sorting process typically involves multiple steps:

- Reading a block of data from the external storage into memory.
- Sorting the data within the memory using an internal sorting algorithm.
- Writing the sorted data back to the external storage.
- Repeating the above steps until all blocks have been read, sorted, and written back.

Common external sorting algorithms include:

- External mergesort
- Polyphase merge sort
- Replacement selection

External sorting is necessary when the dataset is too large to fit in memory, and it allows for sorting large datasets efficiently by minimizing disk I/O operations.

In summary, internal sorting operates entirely within the main memory and is suitable for smaller datasets, while external sorting handles larger datasets that cannot fit in memory by dividing them into manageable blocks and utilizing external storage devices.

Stable and unstable sorting are two different characteristics that describe how a sorting algorithm handles elements with equal keys during the sorting process.

1. Stable Sorting: A stable sorting algorithm maintains the relative order of elements with equal keys as they appear in the original input dataset. In other words, if two elements have the same key value, the element that appeared first in the input will also appear first in the sorted output. The stability of the sorting algorithm ensures that the original order of equal elements is preserved.

Stable sorting algorithms are particularly useful when the dataset contains elements with multiple attributes or when the sorting is performed in multiple passes or with different keys.

Examples of stable sorting algorithms include:

- Insertion sort
- Merge sort
- Bubble sort (with slight modifications)

2. Unstable Sorting: An unstable sorting algorithm does not guarantee the preservation of the original order of elements with equal keys. The relative order of elements with equal keys may change after the sorting process, and they may appear in a different order in the sorted output compared to their original order in the input dataset.

Unstable sorting algorithms typically optimize for efficiency or simplicity and do not prioritize maintaining the original order of equal elements.

Examples of unstable sorting algorithms include:

- Quicksort
- Heapsort
- Selection sort

It's important to note that stability is a property of the sorting algorithm itself, not the data being sorted. The same algorithm can be implemented in a stable or unstable manner by considering or disregarding the original order of equal elements.

The choice between stable and unstable sorting depends on the specific requirements of the application. If maintaining the original order of equal elements is important, a stable sorting algorithm should be chosen. If the relative order of equal elements is not significant or other factors like efficiency are more important, an unstable sorting algorithm may be preferred.

2. Insertion and Selection Sort

Insertion sort

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. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is **$O(n^2)$**, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as -

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

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.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - 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.

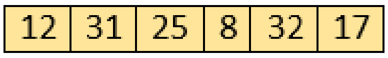
Step 6 - Repeat until the array is sorted.

Working of Insertion sort Algorithm

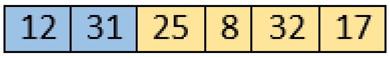
Now, let's see the working of the insertion sort Algorithm.

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.

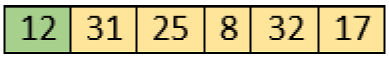
Let the elements of array are -



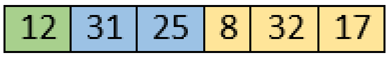
Initially, the first two elements are compared in insertion sort.

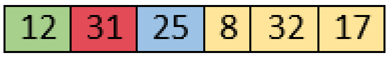


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



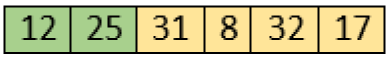
Now, move to the next two elements and compare them.



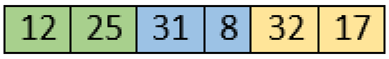


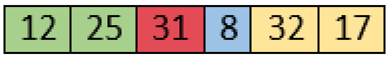
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

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.

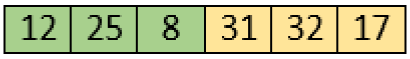


Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

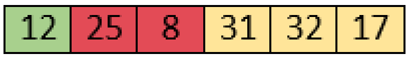




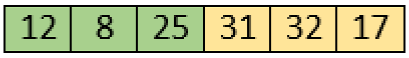
Both 31 and 8 are not sorted. So, swap them.



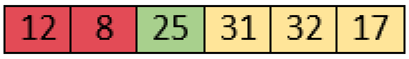
After swapping, elements 25 and 8 are unsorted.



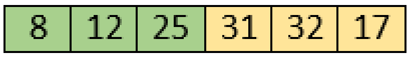
So, swap them.



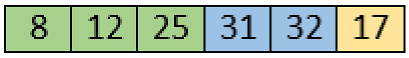
Now, elements 12 and 8 are unsorted.



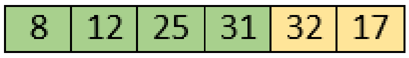
So, swap them too.



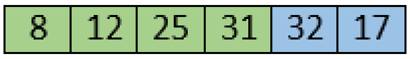
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



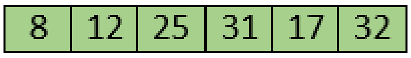
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

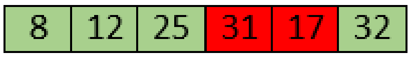


Move to the next elements that are 32 and 17.

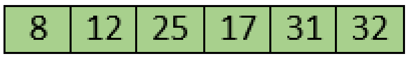


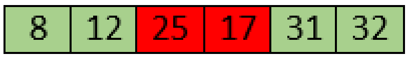
17 is smaller than 32. So, swap them.



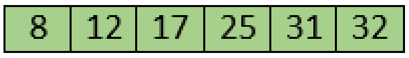


Swapping makes 31 and 17 unsorted. So, swap them too.





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



Now, the array is completely sorted.

Time Complexity Analysis:

Even though insertion sort is efficient, still, if we provide an already sorted array to the insertion sort algorithm, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array of n elements, which makes its best case time complexity a linear function of n. Wherein for an unsorted array, it takes for an element to compare with all the other elements which mean every n element compared with all other n elements. Thus, making it for n x n, i.e., n2 comparisons.

- Worst Case Time Complexity: O(n^2)

- Best Case Time Complexity: O(n)
- Average Time Complexity: O(n^2)
- Space Complexity: O(1)

C++ program for insertion sort

```
#include <iostream>

using namespace std;

void insertionSort(int arr[], int n) {
   for (int i = 1; i < n; i++) {
      int key = arr[i];
      int j = i - 1;

      // Move elements greater than key to one position ahead
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j--;
      }

      arr[j + 1] = key;
   }
}

int main() {
   int arr[] = {5, 2, 8, 12, 3, 1};
   int n = sizeof(arr) / sizeof(arr[0]);

   cout << "Original array: ";
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }
   cout << endl;

   insertionSort(arr, n);

   cout << "Sorted array: ";
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }
```

```
    cout << endl;

    return 0;
}
```

Following are some of the important characteristics of Insertion Sort:

1. It is efficient for smaller data sets, but very inefficient for larger lists.
2. Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
3. It is better than Selection Sort and Bubble Sort algorithms.
4. Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
5. It is a stable sorting technique, as it does not change the relative order of elements which are equal.

Selection Sort

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.

The average and worst-case complexity of selection sort is **$O(n^2)$**, where **n** is the number of items. Due to this, it is not suitable for large data sets.

Selection sort is generally used when -

- A small array is to be sorted
- Swapping cost doesn't matter
- It is compulsory to check all elements

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]
```

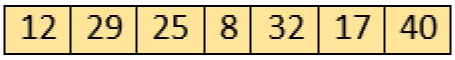
```
Step 4: EXIT

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

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

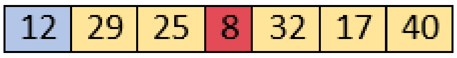
To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are -

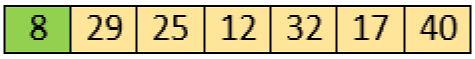


Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

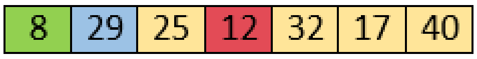
At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.



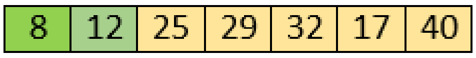
So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.



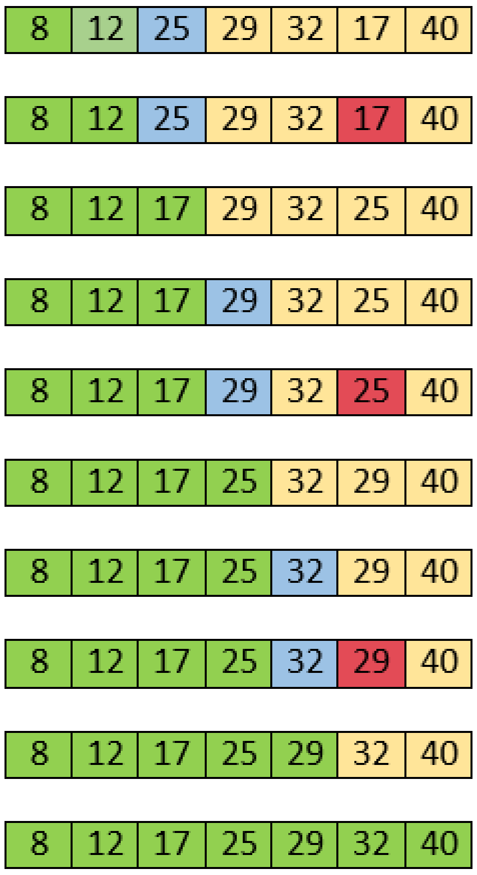
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.



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.

Selection sort complexity

Now, let's see the time complexity of selection sort in best case, average case, and in worst case. We will also see the space complexity of the selection sort.

1. Time Complexity

Case	**Time Complexity**
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is **O(n^2)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **O(n^2)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is **O(n^2)**.

2. Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.

Implementation of selection sort

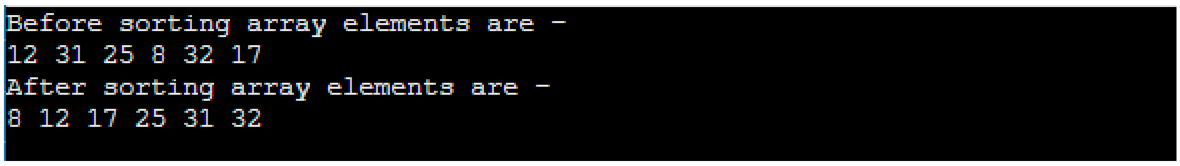
Program: Write a program to implement selection sort in C language.

```
#include <stdio.h>

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++)
      printf("%d ", a[i]);
}
```

```
int main()
{
   int a[] = { 12, 31, 25, 8, 32, 17 };
   int n = sizeof(a) / sizeof(a[0]);
   printf("Before sorting array elements are - \n");
   printArr(a, n);
   selection(a, n);
   printf("\nAfter sorting array elements are - \n");
   printArr(a, n);
   return 0;
}
```

Output:

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



```
Before sorting array elements are -
12 31 25 8 32 17
After sorting array elements are -
8 12 17 25 31 32
```

3. Bubble and Exchange Sort

<u>Bubble Sort</u>

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.

Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is **$O(n^2)$,** where **n** is a number of items.

Bubble short is majorly used where -

- complexity does not matter
- simple and shortcode is preferred

Algorithm

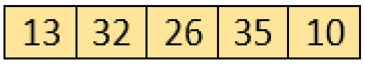
In the algorithm given below, suppose **arr** is an array of **n** elements. The assumed **swap** function in the algorithm will swap the values of given array elements.

```
1. begin BubbleSort(arr)
2.    for all array elements
3.       if arr[i] > arr[i+1]
4.          swap(arr[i], arr[i+1])
5.       end if
6.    end for
7.    return arr
8. end BubbleSort
```

Working of Bubble sort Algorithm

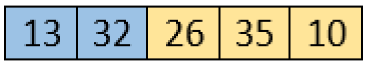
To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is **O(n^2).**

Let the elements of array are -

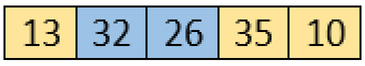


First Pass

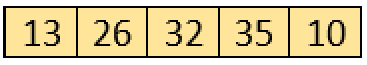
Sorting will start from the initial two elements. Let compare them to check which is greater.



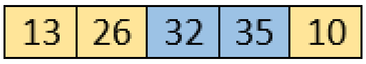
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



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

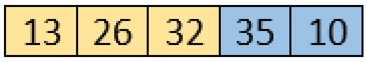


Now, compare 32 and 35.

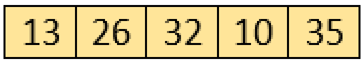


Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

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



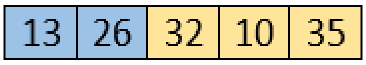
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

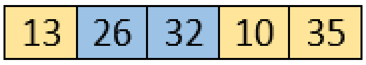


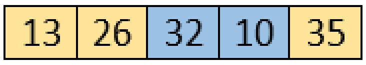
Now, move to the second iteration.

Second Pass

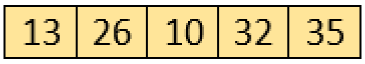
The same process will be followed for second iteration.

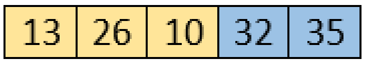






Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

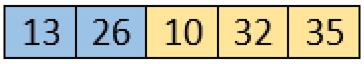


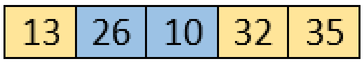


Now, move to the third iteration.

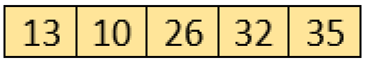
Third Pass

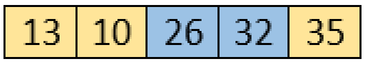
The same process will be followed for third iteration.

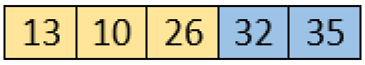




Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



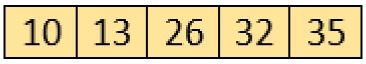




Now, move to the fourth iteration.

Fourth pass

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



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

Complexity Analysis of Bubble Sort

In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,
(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1
Sum = n(n-1)/2
i.e O(n2)

Hence the time complexity of Bubble Sort is O(n2).

The main advantage of Bubble Sort is the simplicity of the algorithm.
The space complexity for Bubble Sort is O(1), because only a single additional memory space is required i.e. for temp variable.

Also, the best case time complexity will be O(n), it is when the list is already sorted.
Following are the Time and Space complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity: O(n2)
- Best Case Time Complexity: O(n)
- Average Time Complexity: O(n2)

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

```
#include <iostream>
using namespace std;

// perform bubble sort
void bubbleSort(int array[], int n) {

  // loop to access each array element
  for (int j = 0; j < n; ++j) {

    // loop to compare array elements
    for (int i = 0; i < n - j - 1; ++i) {

      // compare two adjacent elements
      // change > to < to sort in descending order
      if (array[i] > array[i + 1]) {

        // swapping elements if elements
        // are not in the intended order
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}

// print array
void printArray(int array[], int n) {
  for (int i = 0; i < n; ++i) {
    cout << "  " << array[i];

  }
  cout << "\n";
}

int main() {
  int data[] = {-2, 5, 50, 11, -9};
  // find array's length
  int n = sizeof(data) / sizeof(data[0]);
  bubbleSort(data, n);

  cout << "Sorted Array in Ascending Order:\n";
  printArray(data, n);
}
```

Output

```
Sorted Array in Ascending Order:
  -9  -2  5  11  50
```

Bubble Sort for String

```
#include <iostream>
#include <string>
using namespace std;

// perform bubble sort
void bubbleSort(string array[], int n) {
  // loop to access each array element
  for (int j = 0; j < n; ++j) {
    // loop to compare array elements
    for (int i = 0; i < n - j - 1; ++i) {
      // compare two adjacent elements
      // change > to < to sort in descending order
      if (array[i] > array[i + 1]) {
        // swapping elements if elements
        // are not in the intended order
        string temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}

// print array
void printArray(string array[], int n) {
  for (int i = 0; i < n; ++i) {
    cout << "  " << array[i];
  }
  cout << "\n";
}

int main() {
  string data[] = {"Blue", "Red", "Orange", "Yellow"};

  // find array's length
  int n = sizeof(data) / sizeof(data[0]);
  bubbleSort(data, n);

  cout << "Sorted Array in Ascending Order:\n";
  printArray(data, n);
}
```

Exchange Sort

Exchange sort is an algorithm used to sort in **ascending** as well as **descending** order. It compares the **first** element with every element if any element seems out of order it **swaps**.

Example:

Input: arr[] = {5, 1, 4, 2, 8}
Output: {1, 2, 4, 5, 8}
Explanation: Working of exchange sort:

- **1st Pass**:
  Exchange sort starts with the very first elements, comparing with other elements to check which one is greater.
  (**5 1 4 2 8**) –> (**1 5 4 2 8**).
  Here, the algorithm compares the first two elements and swaps since **5 > 1**.
  No swap since none of the elements is smaller than 1 so after 1st iteration (**1 5 4 2 8**)
- **2nd Pass**:
  (**1 5 4 2 8**) –> (**1 4 5 2 8**), since **4 < 5**
  (**1 4 5 2 8**) –> (**1 2 5 4 8**), since **2 < 4**
  (**1 2 5 4 8**) No change since in this there is no other element smaller than 2
- **3rd Pass:**
  (**1 2 5 4 8**) -> (**1 2 4 5 8**), since **4 < 5**
  after completion of the iteration, we found array is sorted
- After completing the iteration it will come out of the loop, Therefore array is sorted.

To sort in **Ascending** order:

```
procedure ExchangeSort(num: list of sortable items)
  n = length(A)
  // outer loop
  for i = 1 to n – 2 do

  // inner loop

      for j = i + 1 to n-1 do

          if num[i] > num[j] do

             swap(num[i], num[j])
          end if
      end for
   end for
end procedure
```

Steps Involved in Implementation for ascending order sorting:

- First, we will iterate over the array from **arr[1]** to **n – 2** in the outer loop to compare every single element with every other element in an array, inner loops will take of comparing that single element in the outer loop with all the other elements in an array.
- The inner loop will start from **i + 1st** index where i is the index of the outer loop
- We compare if the ith element is bigger than the jth element we swap in case of ascending order
- To sort in descending order we swap array elements if the jth element is bigger than the ith element
- If there is no case where the condition doesn’t meet that means it is already in desired order so we won’t perform any operations
- Here both if and the inner loop end and so does the outer loop after that we didn’t take the last element in the outer loop since the inner loop’s current index is i+1th so eventually, when the current index of the outer loop is n-2 it will automatically take care of last element due to i+1th index of the inner loop. this case can also be considered a corner case for this algorithm

Implementation of Exchange Sort using C++

```
#include<iostream>
using namespace std;

int main(void)
{
        int array[]={5,15,4,8,6};                // An array of integers.
        int length = sizeof(array) / sizeof(array[0]);;                // Length of the array.
        int i, j;
        int temp;
        //Exchange Sorting for ascending
        for(i = 0; i < (length -1); i++)
        {
                for (j=(i + 1); j < length; j++)
                {
                        if (array[i] > array[j])
                        {
                                temp = array[i];
                                array[i] = array[j];
                                array[j] = temp;
                        }
                }
        }
        //Displaying Sorted Array
        for (i = 0; i < length; i++)
        {
                cout << array[i] << " ";
        }
}
```

4. Quick and Merge Sort

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.

Complexity Analysis of Quick Sort

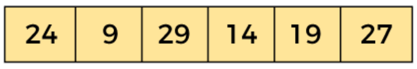
For an array, in which partitioning leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the pivot, hence on the right side.
And if keep on getting unbalanced subarrays, then the running time is the worst case, which is O(n2)
Where as if partitioning leads to almost equal subarrays, then the running time is the best, with time complexity as O(n*log n).

- Worst Case Time Complexity: O(n2)
- Best Case Time Complexity: O(n log n)
- Average Time Complexity: O(n log n)
- Space Complexity: O(n log n)

Working of Quick Sort Algorithm

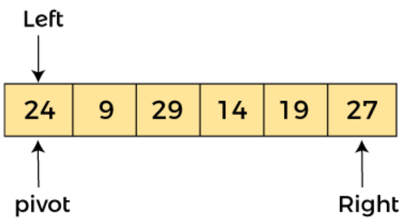
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

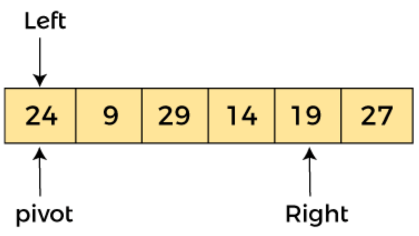


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

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

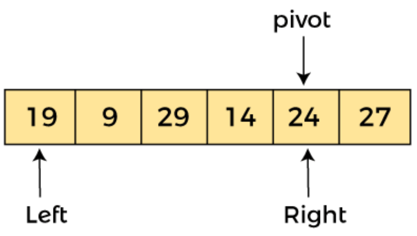


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



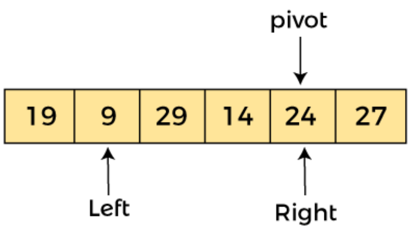
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

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

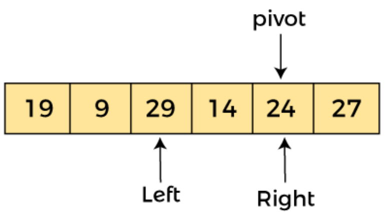


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

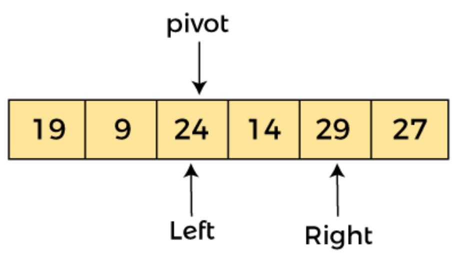
As a[pivot] > a[left], so algorithm moves one position to right as -



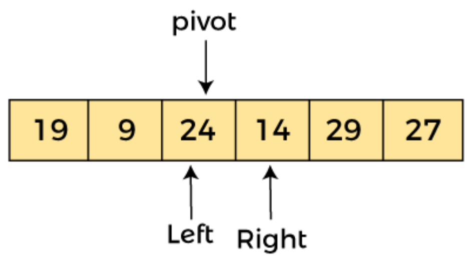
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



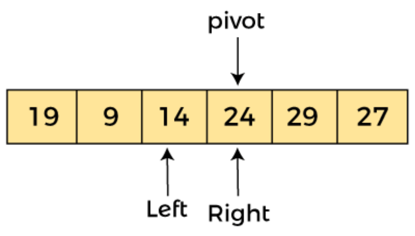
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



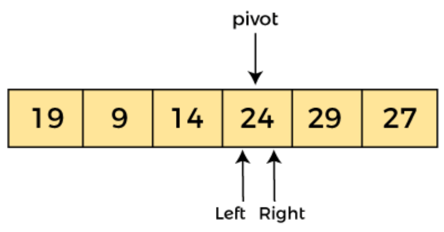
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



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. -



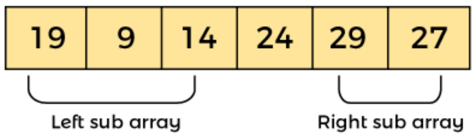
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



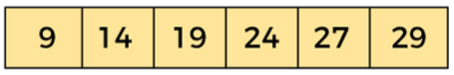
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

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 -

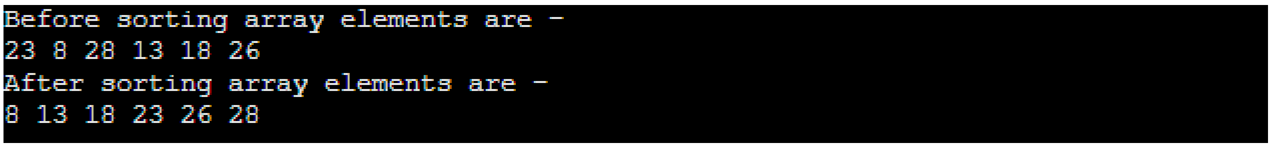


Implementation of Quick sort

```
#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 = Endin
g 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
```

<u>Merge Sort</u>

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**.

Merge sort complexity

Now, let's see the time complexity of merge sort in best case, average case, and in worst case. We will also see the space complexity of the merge sort.

1. Time Complexity

Case	**Time Complexity**
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n*logn)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n*logn)**.

2. Space Complexity

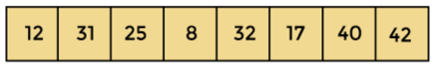
Space Complexity	O(n)
Stable	YES

- The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.

Working of Merge sort Algorithm

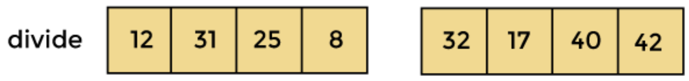
To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

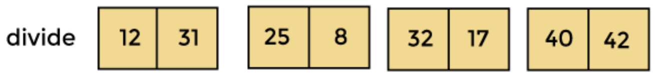


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

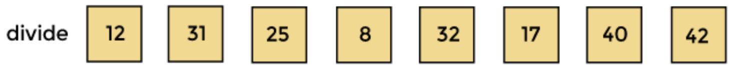
As there are eight elements in the given array, so it is divided into two arrays of size 4.



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



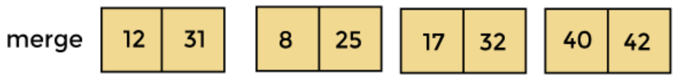
Now, again divide these arrays to get the atomic value that cannot be further divided.



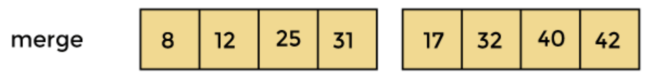
Now, combine them in the same manner they were broken.

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

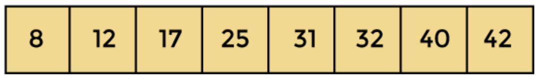
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



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 -



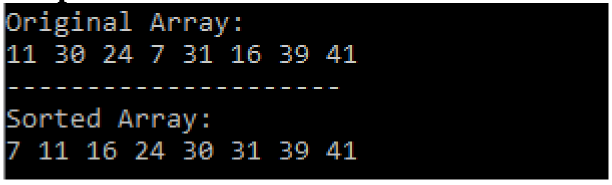
Now, the array is completely sorted.

Implementation of Merge Sort

```
#include <iostream>
using namespace std;
void merge(int a[],int lb, int mid, int ub){
        int b[ub-lb+1];
        int i=lb;
        int j=mid+1;
        int k=lb;
        while(i<=mid&&j<=ub){
                if(a[i]<=a[j]){
                        b[k]=a[i];
                        k++;i++;
                }
                else
                {
                        b[k]=a[j];
                        k++;j++;
                }
        }
        if(i>mid)
        {
                while(j<=ub){
                        b[k]=a[j];
                        k++;j++;
                }
        }
        else{

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

```
	for(int x=lb;x<=ub;x++)
		a[x]=b[x];
}
void mergeSort(int a[] , int lb, int ub){
	if(lb<ub){
		int mid=(lb+ub)/2;
		mergeSort(a,lb, mid);
		mergeSort (a,mid+1,ub);
		merge(a,lb,mid,ub);
	}
}

void printArr(int a[],int n){
	for(int 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<<"Original Array:"<<endl;
	printArr(a,n);
	cout<<"\n---------------------"<<endl;
	mergeSort(a,0,n-1);

	cout<<"Sorted Array:"<<endl;
	printArr(a,n);
	return 0;

}
```

Output:



```
Original Array:
11 30 24 7 31 16 39 41
---------------------
Sorted Array:
7 11 16 24 30 31 39 41
```

5. Radix Sort

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

The process of radix sort works similar to the sorting of students names, according to the alphabetical order. In this case, there are 26 radix formed due to the 26 alphabets in English. In the first pass, the names of students are grouped according to the ascending order of the first letter of their names. After that, in the second pass, their names are grouped according to the ascending order of the second letter of their name. And the process continues until we find the sorted list.

Algorithm

1. radixSort(arr)
2. max = largest element in the given array
3. d = number of digits in the largest element (or, max)
4. Now, create d buckets of size 0 - 9
5. for i -> 0 to d
6. sort the array elements using counting sort (or any stable sort) according to the digits at
7. the ith place

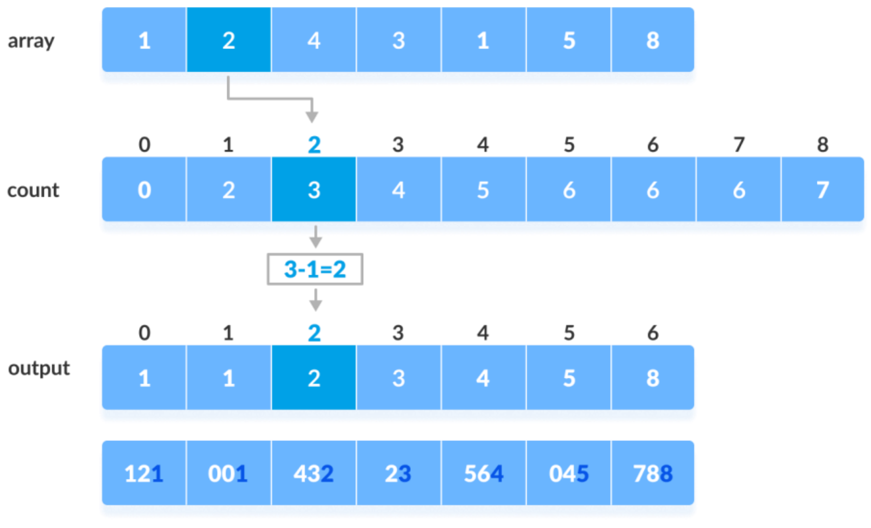
Working of Radix sort Algorithm

1. Find the largest element in the array, i.e. *max*. Let X be the number of digits in max. X is calculated because we have to go through all the significant places of all elements.

   In this array [121, 432, 564, 23, 1, 45, 788], we have the largest number 788. It has 3 digits. Therefore, the loop should go up to hundreds place (3 times).
2. Now, go through each significant place one by one.

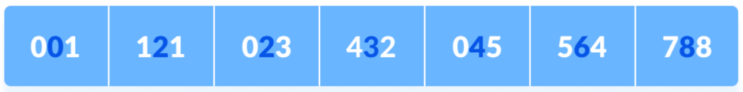
   Use any stable sorting technique to sort the digits at each significant place. We have used counting sort for this.

   Sort the elements based on the unit place digits (X=0).



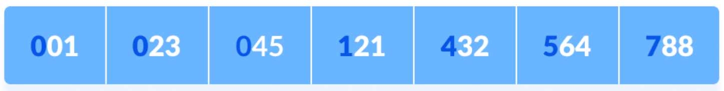
Using counting sort to sort elements based on unit place

3. Now, sort the elements based on digits at tens place.



Sort elements based on tens place

4. Finally, sort the elements based on the digits at hundreds place.



Sort elements based on hundreds place

Radix sort complexity

Now, let's see the time complexity of Radix sort in best case, average case, and worst case. We will also see the space complexity of Radix sort.

1. Time Complexity

Case	**Time Complexity**
Best Case	Ω(n+k)
Average Case	θ(nk)
Worst Case	O(nk)

- **Best Case Complexity** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of Radix sort is **Ω(n+k)**.
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Radix sort is **θ(nk)**.
- **Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Radix sort is **O(nk)**.

Radix sort is a non-comparative sorting algorithm that is better than the comparative sorting algorithms. It has linear time complexity that is better than the comparative algorithms with complexity O(n logn).

2. Space Complexity

Space Complexity	O(n + k)
Stable	YES

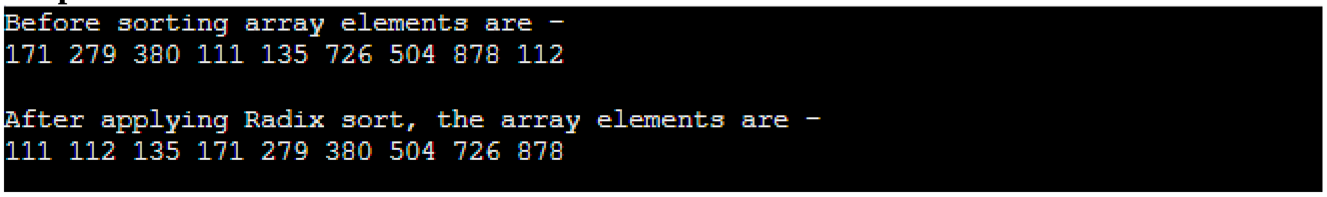
- The space complexity of Radix sort is O(n + k).

Implementation of Radix sort

```
#include <iostream>

using namespace std;

int getMax(int a[], int n) {
   int max = a[0];
   for(int i = 1; i<n; i++) {
      if(a[i] > max)
         max = a[i];
   }
   return max; //maximum element from the array
}

void countingSort(int a[], int n, int place) // function to implement counting sort
{
  int output[n + 1];
  int count[10] = {0};
   // Calculate count of elements
  for (int i = 0; i < n; i++)
    count[(a[i] / place) % 10]++;

  // Calculate cumulative frequency
  for (int i = 1; i < 10; i++)
    count[i] += count[i - 1];

  // Place the elements in sorted order
  for (int i = n - 1; i >= 0; i--) {
    output[--count[(a[i] / place) % 10]] = a[i];
  }

  for (int i = 0; i < n; i++)
    a[i] = output[i];
}

// function to implement radix sort
void radixsort(int a[], int n) {

  // get maximum element from array
  int max = getMax(a, n);

  // Apply counting sort to sort elements based on place value
  for (int place = 1; max / place > 0; place *= 10)
    countingSort(a, n, place);
}
```

```
  // function to print array elements
void printArray(int a[], int n) {
  for (int i = 0; i < n; ++i)
    cout<<a[i]<<" ";
}

int main() {
  int a[] = {171, 279, 380, 111, 135, 726, 504, 878, 112};
  int n = sizeof(a) / sizeof(a[0]);
  cout<<"Before sorting array elements are - \n";
  printArray(a,n);
  radixsort(a, n);
  cout<<"\n\nAfter applying Radix sort, the array elements are - \n";
  printArray(a, n);
  return 0;
}
```

Output:



```
Before sorting array elements are -
171 279 380 111 135 726 504 878 112

After applying Radix sort, the array elements are -
111 112 135 171 279 380 504 726 878
```

6. Shell Sort

Shell sort is the generalization of insertion sort, which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

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 -

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

Now, let's see the algorithm of shell sort.

Algorithm

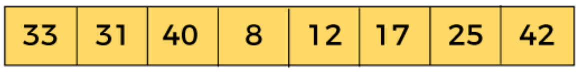
The simple steps of achieving the shell sort are listed as follows -

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

Working of 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 -

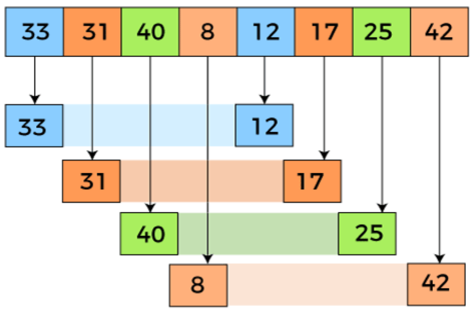


We will use the original sequence of shell sort, i.e., N/2, N/4,....,1 as the intervals.

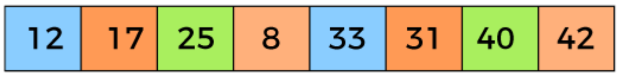
In the first loop, n is equal to 8 (size of the array), so the elements are lying at the interval of 4 (n/2 = 4). Elements will be compared and swapped if they are not in order.

Here, in the first loop, the element at the 0th position will be compared with the element at 4th position. If the 0th element is greater, it will be swapped with the element at 4th 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}.

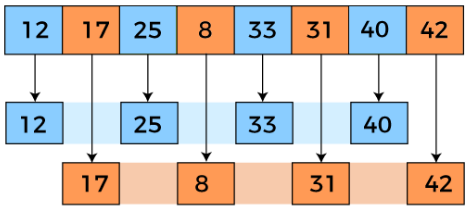


Now, we have to compare the values in every sub-list. After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows -

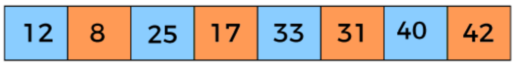


In the second loop, elements are lying at the interval of 2 (n/4 = 2), where n = 8.

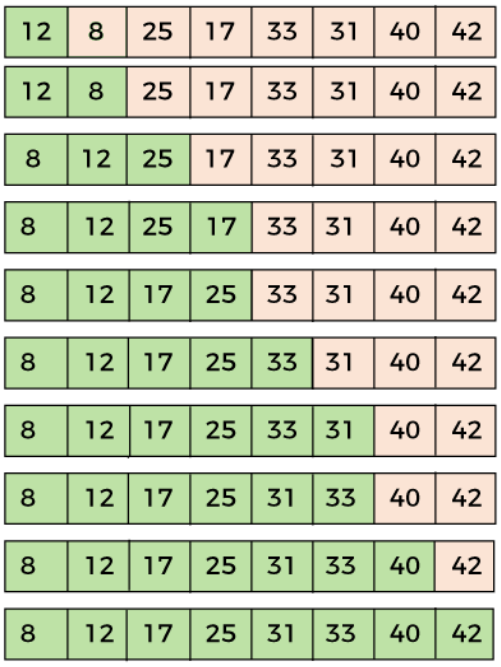
Now, we are taking the interval of **2** to sort the rest of the array. With an interval of 2, two sublists will be generated - {12, 25, 33, 40}, and {17, 8, 31, 42}.



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 -



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.



Now, the array is sorted in ascending order.

Shell sort complexity

Now, let's see the time complexity of Shell sort in the best case, average case, and worst case. We will also see the space complexity of the Shell sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	$O(n*\log(n)^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e., the array is already sorted. The best-case time complexity of Shell sort is **O(n*logn).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Shell sort is **O(n*logn).**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Shell sort is **$O(n^2)$.**

2. Space Complexity

Space Complexity	O(1)
Stable	NO

- The space complexity of Shell sort is O(1).

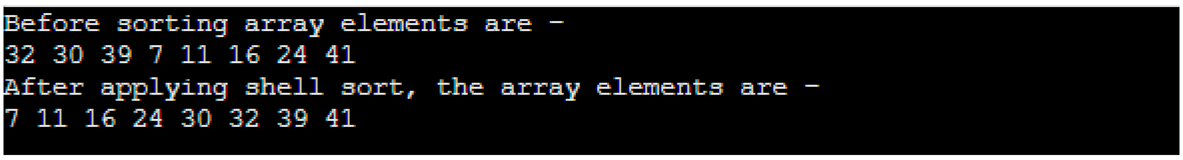
Implementation of Shell sort

```
#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 gap = n/2; gap >= 1; gap /= 2){
      for (int j = gap; j < n; j++){
           for(int i = j-gap; i>=0; i=i-gap){
                        if(a[i+gap] > a[i]){
                                break;
                        }
                        else{
                                int temp = a[i+gap];
                                a[i+gap] = a[i];
                                a[i] = 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 applying shell sort, the array elements are -
7 11 16 24 30 32 39 41
```

7. Heap Sort as priority queue

Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations -

- Build a heap H, using the elements of array.
- Repeatedly delete the root element of the heap formed in 1st phase.

Heap

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

Heap Sort

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.

Working of Heap sort Algorithm

Now, let's see the working of the Heapsort Algorithm.

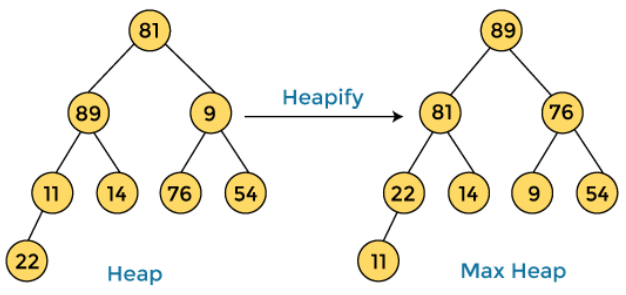
In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

- 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

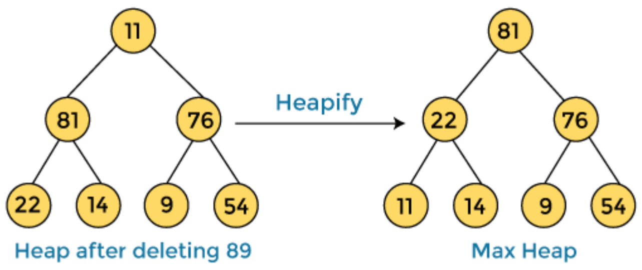
First, we have to construct a heap from the given array and convert it into max heap.



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

89	81	76	22	14	9	54	11

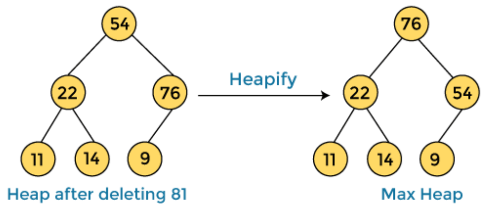
Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



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

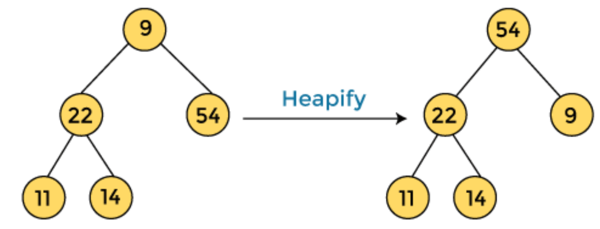
In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping 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.

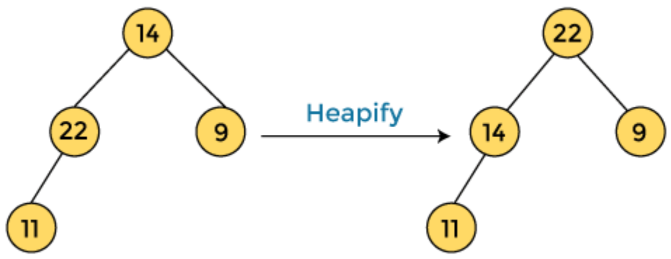


Heap after deleting 76 Max Heap

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.

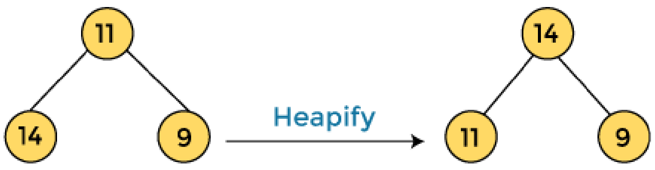


Heap after deleting 54 Max Heap

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

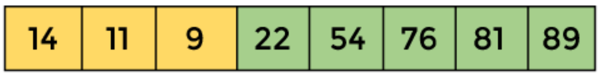
22	14	9	11	54	76	81	89

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.

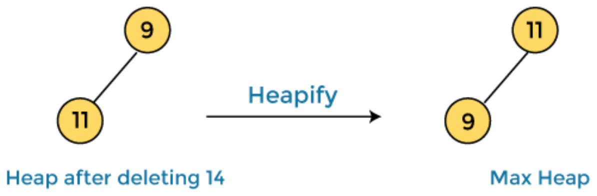


Heap after deleting 22 Max Heap

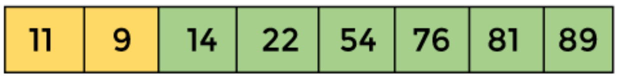
After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -



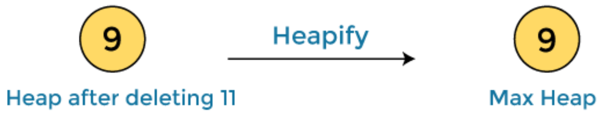
In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



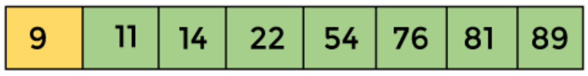
After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -



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.



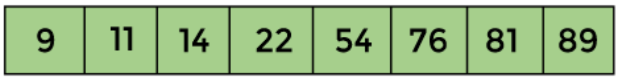
After swapping the array element **11** with **9,** the elements of array are -



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

9
Remove 9
Empty

After completion of sorting, the array elements are -



Now, the array is completely sorted.

Heap sort complexity

Now, let's see the time complexity of Heap sort in the best case, average case, and worst case. We will also see the space complexity of Heapsort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n logn)
Average Case	O(n log n)
Worst Case	O(n log n)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**

The time complexity of heap sort is **O(n logn)** in all three cases (best case, average case, and worst case). The height of a complete binary tree having n elements is **logn.**

2. Space Complexity

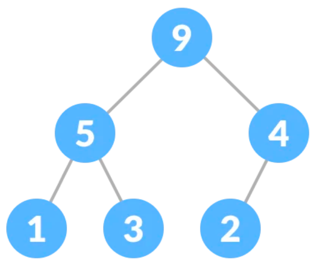
Space Complexity	O(1)
Stable	NO

- The space complexity of Heap sort is O(1).

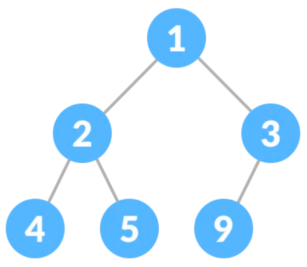
Priority Queue - using Heap

If you use the max heap, the element with highest value is removed first.

If you use the min heap, the element with smallest value is removed first.



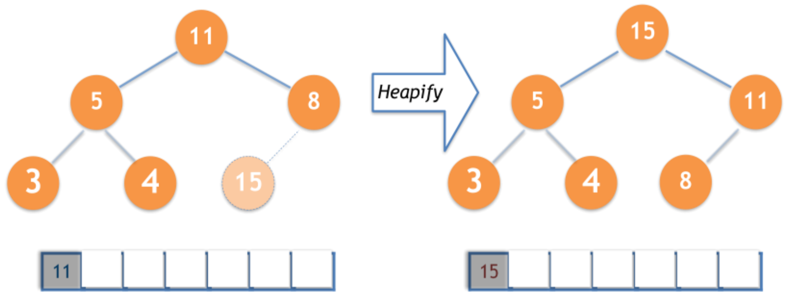
Max Heap



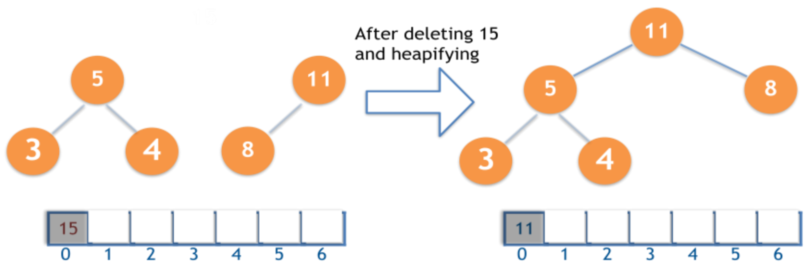
Min Heap

Priority Queue using Max Heap:

- **Insertion operation:**
    1. Add the element to the bottom level of the heap at the leftmost open space.
    2. Compare the added element with its parent; if they ate in the correct order, stop.
    3. If not, swap the element with its parent and return to the previous step 2.
- The time complexity of insertion is O($\log_2 n$) since the maximum number of swaps that can takes to heapify is the number of levels to height of the heap in worst case.



- **Deletion Operation:**
    1. Pick the value from the root of the heap
    2. Replace the root with the last element on the last level.
    3. Compare the new root with its children; if they are in the correct order, **stop**.
    4. If not, swap the element with one of its children and return to the previous step 2.
    - The deletion operation every time the element with highest value is removed (i.e. element with highest priority is removed in max heap)
    - Deletion also takes O($\log_2 n$) time while pick operation takes O(1) time.



Implementation of Heap Sort

```
#include <iostream>
using namespace std;

// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
}
	int largest = i; // Initialize largest as root Since we are using 0 based indexing
	int l = 2 * i + 1; // left = 2*i + 1
	int r = 2 * i + 2; // right = 2*i + 2

	// If left child is larger than root
	if (l < n && arr[l] > arr[largest])
	largest = l;

	// If right child is larger than largest so far
	if (r < n && arr[r] > arr[largest])
	largest = r;

	// If largest is not root
	if (largest != i) {
		swap(arr[i], arr[largest]);

		// Recursively heapify the affected sub-tree
		heapify(arr, n, largest);
	}
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
	// Build heap (rearrange array)
	for (int i = n / 2 - 1; i >= 0; i--)
	heapify(arr, n, i);

	// One by one extract an element from heap
	for (int i = n - 1; i >= 0; i--) {
		// Move current root to end
		swap(arr[0], arr[i]);

		// call max heapify on the reduced heap
		heapify(arr, i, 0);
	}
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
```

```
{
        for (int i = 0; i < n; ++i)
                cout << arr[i] << " ";
        cout << "\n";
}

// Driver program
int main()
{
        int arr[] = { 60 ,20 ,40 ,70, 30, 10};
        int n = sizeof(arr) / sizeof(arr[0]);
        //heapify algorithm
        // the loop must go reverse you will get after analyzing manually
        // (i=n/2 -1) because other nodes/ ele's are leaf nodes
        // (i=n/2 -1) for 0 based indexing
        // (i=n/2)  for 1 based indexing
        for(int i=n/2 -1;i>=0;i--){
                heapify(arr,n,i);
        }

        cout << "After heapifying array is \n";
        printArray(arr, n);

        heapSort(arr, n);

        cout << "Sorted array is \n";
        printArray(arr, n);
        return 0;
}
```

Extract Maximum: In this operation, the maximum element will be returned and the last element of heap will be placed at index 1 and max_heapify will be performed on node 1 as placing last element on index 1 will violate the property of max-heap.

```
int extract_maximum (int Arr[ ])
{
        if(length == 0)
        {
                cout<<  "Can't  remove  element
        as queue is empty";  return -1;                    }
        }
        int max = Arr[1];
        Arr[1] = Arr[length];
        length = length -1;
        max_heapify(Arr, 1);
        return max;
```