Chapter 7: Sorting

Sorting refers to arranging a list of data in a particular order- ascending or descending. Most common orders are in numerical or lexicographical order. There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of **sorting** came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search. **Sorting** arranges data in a sequence which makes searching easier. The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted form.

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**. If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**. If the number of data is small enough to fits into the main memory, sorting is called *internal sorting*. If the number of data is so large that some of them reside on external storage during the sort, it is called *external sorting*.

Internal Sorting

An **internal sort** is any data sorting process that takes place entirely within the main memory of a computer. This is possible whenever the data to be sorted is small enough to all be held in the main memory. Some common internal sorting algorithms include: bubble sort, insertion sort, quick sort, heap sort, radix sort and selection sort.

External Sorting

External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.

One example of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together. We first divide the file into **runs** such that the size of a run is small enough to fit into main memory. Then sort each run in main memory using merge sort sorting algorithm. Finally merge the resulting runs together into successively bigger runs, until the file is sorted.

Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands. Consider you have 10 cards out of a deck of cards in your hand. And they are sorted, or arranged in the ascending order of their numbers. If your friend gives you another card, and ask you to **insert** the card in just the right position, so that the cards in your hand are still sorted. What will you do? Well, you will have to go through each card from the starting or the back and find the right position for the new card, comparing its value with each card. Once you find the right position, you will **insert** the card there. Similarly, if more new cards are provided to you, you can easily repeat the same process and insert the new cards and keep the cards sorted too. This is exactly how **insertion sort** works.

The first element in the array is already sorted. Therefore, insertion sort starts from the second element (index 1). Following are the steps involved in insertion sort:

1. We start by making the second element of the given array, i.e. element at index 1, the key.
2. We compare the current element with the element(s) before it, in this case, element at index 0:
    a. If the current element is less than the first element, we insert the key element before the first element.
    b. If the key element is greater than the first element, then we insert it after the first element.
3. Then, we take the third element of the array as current element and will compare it with all elements to it's left and insert it at the right position.
4. And we go on repeating this, until the array is sorted.

Algorithm

Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

Pseudocode

```
procedure insertionSort( A : array of items )
  int holePosition
  int valueToInsert

  for i = 1 to length(A) inclusive do:
```

```
    /* select value to be inserted */
    valueToInsert = A[i]
    holePosition = i

    /*locate hole position for the element to be inserted */

    while holePosition > 0 and A[holePosition-1] > valueToInsert do:
      A[holePosition] = A[holePosition-1]
      holePosition = holePosition -1
    end while

    /* insert the number at hole position */
    A[holePosition] = valueToInsert

  end for

end procedure
```

Example 1: Illustrate the operation of INSERTION-SORT on the array A = (31, 41, 59, 26, 41, 58).

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	Remarks
K = 0	31	41	59	26	41	58	**No shift**
K = 1	31	41	59	26	41	58	**No shift**
K = 2	31	41	59	26	41	58	**Shift = 3** **Move = 1**
K = 3	26	31	41	59	41	58	**Shift = 1** **Move = 1**
K = 4	26	31	41	41	59	58	**Shift = 1** **Move = 1**
K = 5	**26**	**31**	**41**	**41**	**58**	**59**	**Which is sorted**

Example 2: We color (bold) a sorted part in green, and an unsorted part in black. Here is an insertion sort step by step. We take an element from unsorted part and compare it with elements in sorted part, moving from right to left.

29, 20, 73, 34, 64

29, 20, 73, 34, 64

20, 29, 73, 34, 64

20, 29, 73, 34, 64

20, 29, 34, 73, 64

20, 29, 34, 64, 73

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(n2)
- Best Case Time Complexity: O(n)
- Average Time Complexity: O(n2)
- Space Complexity: **O(1)**

C++ program for insertion sort

```
#include <bits/stdc++.h>
using namespace std;

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++)
   {
      key = arr[i];
      j = i - 1;

      /* Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position */
      while (j >= 0 && arr[j] > key)
      {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}

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

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

int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    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

Selection sort is conceptually the simplest sorting algorithm. This algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position, then it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing this until the entire array is sorted. It is called selection sort because it repeatedly **selects** the next-smallest element and swaps it into the right place.

Following are the steps involved in selection sort(for sorting a given array in ascending order):

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
2. We then move on to the second position, and look for smallest element present in the subarray, starting from index 1, till the last index.
3. We replace the element at the **second** position in the original array, or we can say at the first position in the subarray, with the second smallest element.
4. This is repeated, until the array is completely sorted.

Algorithm

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

Pseudocode

```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for

end procedure
```

Example.

29, 64, 73, 34, **20**,

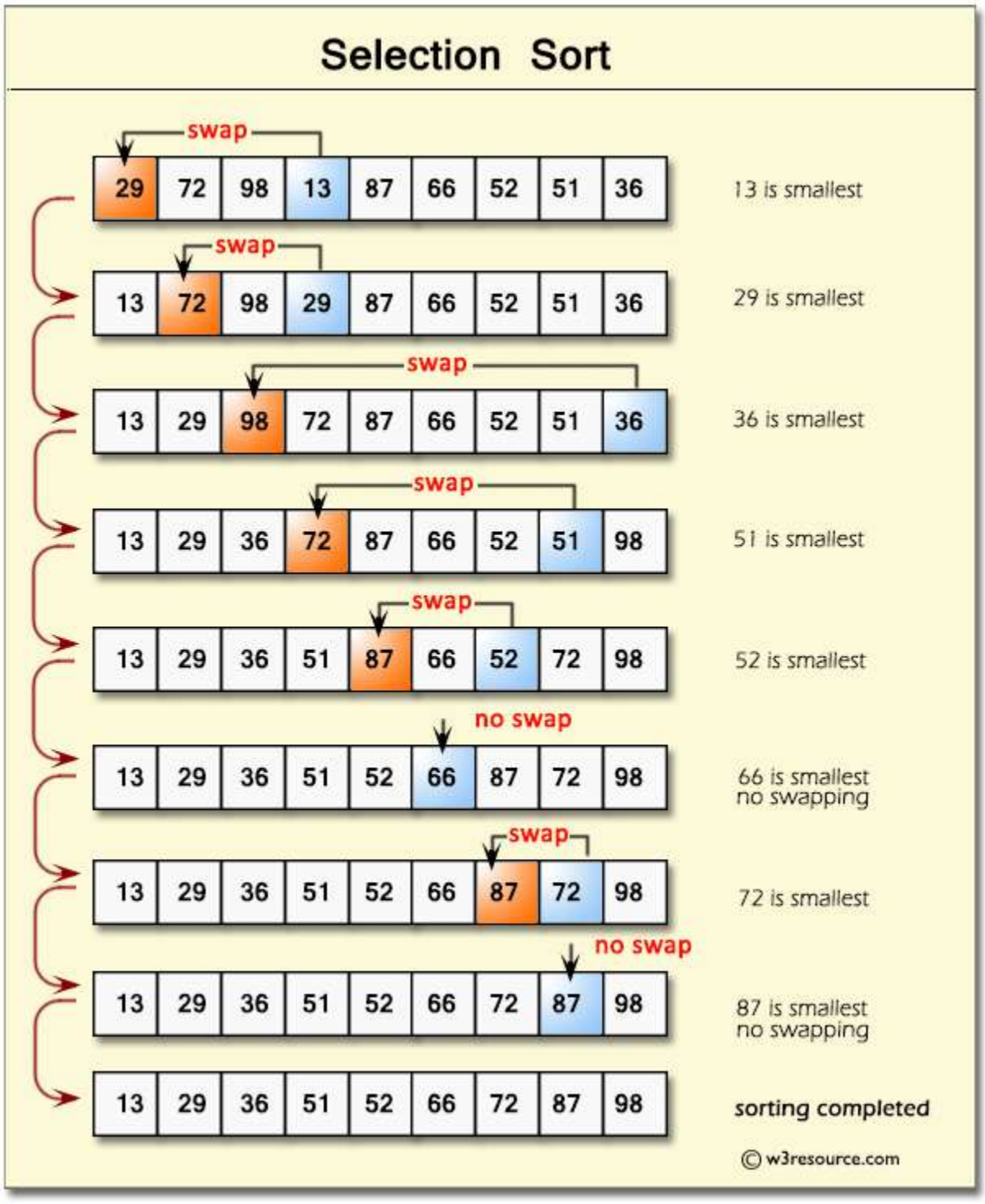
20, **64**, 73, 34, **29**,

20, 29, **73**, **34**, 64

20, 29, 34, **73**, **64**

20, 29, 34, 64, 73

Example 2:



C++ program for implementation of selection sort

include <stdio.h>

```
// function to swap elements at the given index values
void swap(int arr[], int firstIndex, int secondIndex)
{
   int temp;
   temp = arr[firstIndex];
   arr[firstIndex] = arr[secondIndex];
   arr[secondIndex] = temp;
}

// function to look for smallest element in the given subarray
int indexOfMinimum(int arr[], int startIndex, int n)
{
   int minValue = arr[startIndex];
   int minIndex = startIndex;

   for(int i = minIndex + 1; i < n; i++) {
      if(arr[i] < minValue)
      {
         minIndex = i;
         minValue = arr[i];
      }
   }
   return minIndex;
}

void selectionSort(int arr[], int n)
{
   for(int i = 0; i < n; i++)
   {
      int index = indexOfMinimum(arr, i, n);
      swap(arr, i, index);
   }

}

void printArray(int arr[], int size)
{
   int i;
   for(i = 0; i < size; i++)
   {
      printf("%d ", arr[i]);
   }
   printf("\n");
}

int main()
{
   int arr[] = {46, 52, 21, 22, 11};
```

```
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

C++ program for implementation of selection sort

```
#include <bits/stdc++.h>
using namespace std;

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}
```

```
// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Note: Selection sort is an **unstable sort** i.e it might change the occurrence of two similar elements in the list while sorting. But it can also work as a stable sort when it is implemented using linked list.

Complexity Analysis of Selection Sort

Selection Sort requires two nested `for` loops to complete itself, one `for` loop is in the function `selectionSort`, and inside the first loop we are making a call to another function `indexOfMinimum`, which has the second (inner) `for` loop. Hence for a given input size of `n`, following will be the time and space complexity for selection sort algorithm:

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

Bubble Sort

Bubble Sort is a simple algorithm which is used to sort a given set of `n` elements provided in form of an array with `n` number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.

If we have total `n` elements, then we need to repeat this process for `n-1` times.

It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required. It is a stable sort.

Algorithm

The algorithm works by comparing each item in the list with the item next to it, and swapping them if required. In other words, the largest element has bubbled to the top of the array. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items.

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of BubbleSort algorithm can be written as follows −

```
procedure bubbleSort( list : array of items )

   loop = list.count;

   for i = 0 to loop-1 do:
      swapped = false

      for j = 0 to loop-1 do:

         /* compare the adjacent elements */
         if list[j] > list[j+1] then
            /* swap them */
            swap( list[j], list[j+1] )
```

```
        swapped = true
      end if

    end for

    /*if no number was swapped that means
    array is sorted now, break the loop.*/

    if(not swapped) then
      break
    end if

  end for

end procedure return list
```

Example: Here is one step of the algorithm. The largest element - 7 - is bubbled to the top:

7, 5, 2, 4, 3, 9
5, **7, 2**, 4, 3, 9
5, 2, **7, 4**, 3, 9
5, 2, 4, **7, 3**, 9
5, 2, 4, 3, **7, 9**
5, 2, 4, 3, 7, 9

Repeat these step until all the largest element bubbled to their position.

C program for bubble sort

```
#include <stdio.h>

void bubbleSort(int arr[], int n)
{
   int i, j, temp;
   for(i = 0; i < n; i++)
   {
      for(j = 0; j < n-i-1; j++)
      {
         if( arr[j] > arr[j+1])
         {
            // swap the elements
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }
```

```
    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
    }
}

int main()
{
    int arr[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);

    return 0;
}
```

Although the above logic will sort an unsorted array, still the above algorithm is not efficient because as per the above logic, the outer for loop will keep on executing for **6** iterations even if the array gets sorted after the second iteration.

So, we can clearly optimize this algorithm.

Optimized Bubble Sort Algorithm

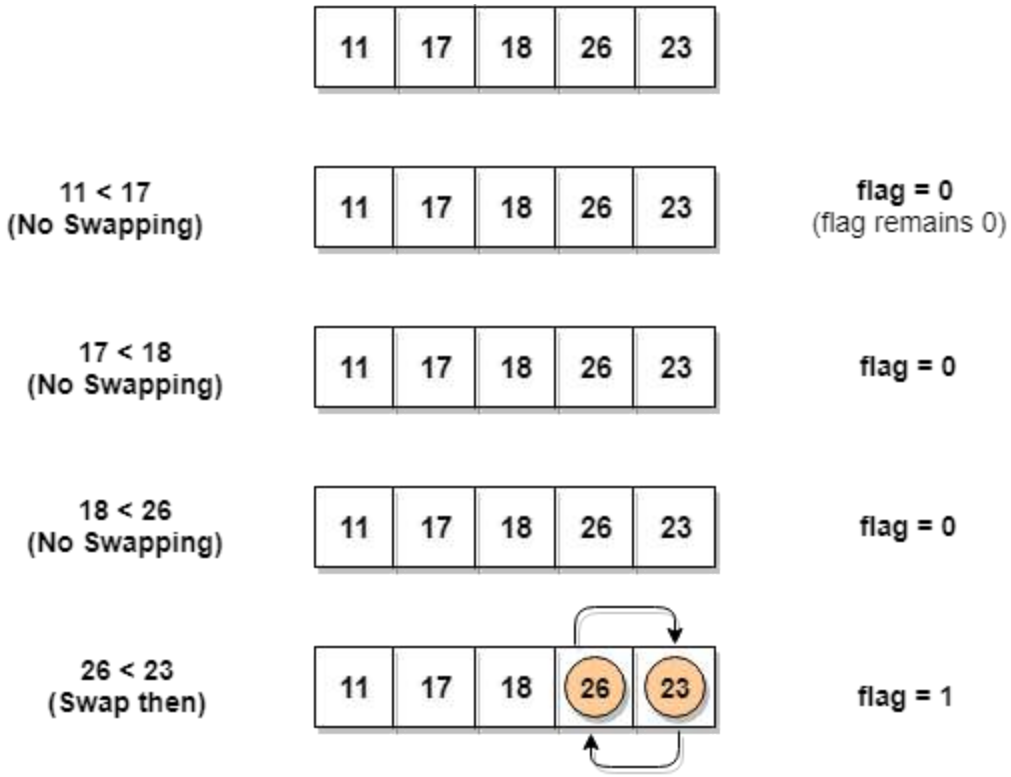
To optimize our bubble sort algorithm, we can introduce a flag to monitor whether elements are getting swapped inside the inner for loop.

Hence, in the inner for loop, we check whether swapping of elements is taking place or not, everytime.

If for a particular iteration, no swapping took place, it means the array has been sorted and we can jump out of the for loop, instead of executing all the iterations.

Let's consider an array with values {11, 17, 18, 26, 23}

Below, we have a pictorial representation of how the optimized bubble sort will sort the given array.



As we can see, in the first iteration, swapping took place, hence we updated our flag value to 1, as a result, the execution enters the for loop again. But in the second iteration, no swapping will occur, hence the value of flag will remain 0, and execution will break out of loop.

C++ program for implementation of Bubble sort

```
#include <bits/stdc++.h>
using namespace std;

void swap(int *xp, int *yp)
{
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

   // Last i elements are already in place
   for (j = 0; j < n-i-1; j++)
      if (arr[j] > arr[j+1])
```

```
            swap(&arr[j], &arr[j+1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver code
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    cout<<"Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

The above function always runs O(n^2) time even if the array is sorted. It can be optimized by stopping the algorithm if inner loop didn't cause any swap.

C++ program with optimized implementation of Bubble sort

```
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   bool swapped;
   for (i = 0; i < n-1; i++)
   {
     swapped = false;
     for (j = 0; j < n-i-1; j++)
     {
```

```
      if (arr[j] > arr[j+1])
      {
        swap(&arr[j], &arr[j+1]);
        swapped = true;
      }
    }

    // IF no two elements were swapped by inner loop, then break
    if (swapped == false)
      break;
  }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
   int i;
   for (i=0; i < size; i++)
      printf("%d ", arr[i]);
   printf("n");
}

// Driver program to test above functions
int main()
{
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr)/sizeof(arr[0]);
   bubbleSort(arr, n);
   printf("Sorted array: \n");
   printArray(arr, n);
   return 0;
}
```

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 $O(n^2)$

Hence the **time complexity** of Bubble Sort is **$O(n^2)$**.

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

Exchange Sort

The exchange sort is almost similar as the bubble sort. The exchange sort compares each element of an array and swap those elements that are not in their proper position, just like a bubble sort does. The only difference between the two sorting algorithms is the manner in which they compare the elements.

The exchange sort compares the first element with each element of the array, making a swap where is necessary.
In some situations the exchange sort is slightly more efficient than its counterpart- bubble sort. The bubble sort needs a final pass to determine that it is finished, thus is slightly less efficient than the exchange sort, because the exchange sort doesn't need a final pass.

Algorithm

1. Set i to 0
2. Set j to i + 1
3. If a[i] > a[j], exchange their values
4. Set j to j + 1. If j < n goto step 3
5. Set i to i + 1. If i < n - 1 goto step 2
6. a is now sorted in ascending order.

Note: n is the number of elements in the array.

Example:

Sort the list 83 91 7 56 45 73 37 19 10 64 using Exchange Sort algorithm:

Pass # 1

Switch elements 83 7 at indices 1 3

83 91 **7** 56 45 73 37 19 10 64

7 91 **83** 56 45 73 37 19 10 64

Pass # 2

Switch elements 91 83 at indices 2 3

7 **91** **83** 56 45 73 37 19 10 64

7 **83** **91** 56 45 73 37 19 10 64

Switch elements 83 56 at indices 2 4

7 **83** 91 **56** 45 73 37 19 10 64

7 **56** 91 **83** 45 73 37 19 10 64

Switch elements 56 45 at indices 2 5

7 **56** 91 83 **45** 73 37 19 10 64

7 **45** 91 83 **56** 73 37 19 10 64

Switch elements 45 37 at indices 2 7

7 **45** 91 83 56 73 **37** 19 10 64

7 **37** 91 83 56 73 **45** 19 10 64

Switch elements 37 19 at indices 2 8

7 **37** 91 83 56 73 45 **19** 10 64

7 **19** 91 83 56 73 45 **37** 10 64

Switch elements 19 10 at indices 2 9

7 **19** 91 83 56 73 45 37 **10** 64

7 **10** 91 83 56 73 45 37 **19** 64

Pass # 3

Switch elements 91 83 at indices 3 4

7 10 **91** **83** 56 73 45 37 19 64

7 10 **83** **91** 56 73 45 37 19 64

Switch elements 83 56 at indices 3 5

7 10 **83** 91 **56** 73 45 37 19 64

7 10 **56** 91 **83** 73 45 37 19 64

Switch elements 56 45 at indices 3 7

7 10 **56** 91 83 73 **45** 37 19 64

7 10 **45** 91 83 73 **56** 37 19 64

Switch elements 45 37 at indices 3 8

7 10 **45** 91 83 73 56 **37** 19 64

7 10 **37** 91 83 73 56 **45** 19 64

Switch elements 37 19 at indices 3 9

7 10 **37** 91 83 73 56 45 **19** 64

7 10 **19** 91 83 73 56 45 **37** 64

Pass # 4

Switch elements 91 83 at indices 4 5

7 10 19 **91** **83** 73 56 45 37 64

7 10 19 **83** **91** 73 56 45 37 64

Switch elements 83 73 at indices 4 6

7 10 19 **83** 91 **73** 56 45 37 64

7 10 19 **73** 91 **83** 56 45 37 64

Switch elements 73 56 at indices 4 7

7 10 19 **73** 91 83 **56** 45 37 64

7 10 19 **56** 91 83 **73** 45 37 64

Switch elements 56 45 at indices 4 8

7 10 19 **56** 91 83 73 **45** 37 64

7 10 19 **45** 91 83 73 **56** 37 64

Switch elements 45 37 at indices 4 9

7 10 19 **45** 91 83 73 56 **37** 64

7 10 19 **37** 91 83 73 56 **45** 64

Pass # 5

Switch elements 91 83 at indices 5 6

7 10 19 37 **91** **83** 73 56 45 64

7 10 19 37 **83** **91** 73 56 45 64

Switch elements 83 73 at indices 5 7

7 10 19 37 **83** 91 **73** 56 45 64

7 10 19 37 **73** 91 **83** 56 45 64

Switch elements 73 56 at indices 5 8

7 10 19 37 **73** 91 83 **56** 45 64

7 10 19 37 **56** 91 83 **73** 45 64

Switch elements 56 45 at indices 5 9

7 10 19 37 **56** 91 83 73 **45** 64

7 10 19 37 **45** 91 83 73 **56** 64

Pass # 6

Switch elements 91 83 at indices 6 7

7 10 19 37 45 **91** **83** 73 56 64

7 10 19 37 45 **83** **91** 73 56 64

Switch elements 83 73 at indices 6 8

7 10 19 37 45 **83** 91 **73** 56 64

7 10 19 37 45 **73** 91 **83** 56 64

Switch elements 73 56 at indices 6 9

7 10 19 37 45 **73** 91 83 **56** 64

7 10 19 37 45 **56** 91 83 **73** 64

Pass # 7

Switch elements 91 83 at indices 7 8

7 10 19 37 45 56 **91** **83** 73 64

7 10 19 37 45 56 **83** **91** 73 64

Switch elements 83 73 at indices 7 9

7 10 19 37 45 56 **83** 91 **73** 64

7 10 19 37 45 56 **73** 91 **83** 64

Switch elements 73 64 at indices 7 10

7 10 19 37 45 56 **73** 91 83 **64**

7 10 19 37 45 56 **64** 91 83 **73**

Pass # 8

Switch elements 91 83 at indices 8 9

7 10 19 37 45 56 64 **91** **83** 73

7 10 19 37 45 56 64 **83** **91** 73

Switch elements 83 73 at indices 8 10

7 10 19 37 45 56 64 **83** 91 **73**

7 10 19 37 45 56 64 **73** 91 **83**

Pass # 9

Switch elements 91 83 at indices 9 10

7 10 19 37 45 56 64 73 **91** **83**

7 10 19 37 45 56 64 73 **83** **91**

Summary:

7 91 83 56 45 73 37 19 10 64

7 **10** 91 83 56 73 45 37 19 64

7 **10** **19** 91 83 73 56 45 37 64

7 **10** **19** **37** 91 83 73 56 45 64

7 **10** **19** **37** **45** 91 83 73 56 64

7 **10** **19** **37** **45** **56** 91 83 73 64

7 **10** **19** **37** **45** **56** **64** 91 83 73

7 **10** **19** **37** **45** **56** **64** **73** 91 83

7 **10** **19** **37** **45** **56** **64** **73** **83** 91

Implementation of Exchange Sort using C

```
#include <stdio.h>
void sort( int [], int );
void sort( int a[], int elements )
{
        int i, j, temp;
```

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

main()
{
        int numbers[] = { 10, 9, 8, 23, 19, 11, 2, 7, 1, 13, 12 };
        int loop;
        printf("Before the sort the array was \n");
        for( loop = 0; loop < 11; loop++ )
                printf(" %d ", numbers[loop] );
        sort(numbers, 11 );
        printf("After the sort the array was \n");
        for( loop = 0; loop < 11; loop++ )
                printf(" %d ", numbers[loop] );
}
```

Quick Sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(nLogn) and $O(n^2)$, respectively.

Quick Sort is based on the concept of **Divide and Conquer**, just like merge sort. But in quick sort all the heavy lifting(major work) is done while **dividing** the array into subarrays, while in case of merge sort, all the real work happens during **merging** the subarrays. In case of quick sort, the combine step does absolutely nothing.

It is also called **partition-exchange sort**. This algorithm divides the list into three main parts:

1. Elements less than the **Pivot** element
2. Pivot element(Central element)

3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. We will take the rightmost element or the last element as **pivot**.

For example: In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take 25 as **pivot**. So after the first pass, the list will be changed like this.

{6 8 17 14 **25** 63 37 52}

Hence after the first pass, pivot will be set at its position, with all the elements **smaller** to it on its left and all the elements **larger** than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate sunarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted. The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Partitioning Algorithm

Step 1 − Choose the highest index value as pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

Quick Sort Algorithm

Using partitioning algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

```
Step 1 - Make the right-most index value pivot
Step 2 - partition the array using pivot value
Step 3 - quicksort left partition recursively
Step 4 - quicksort right partition recursively
```

C Program Implementing Quick Sort

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 7
int array[MAX] = {4,6,3,2,1,9,7};

void display() {
   int i;
   printf("[");
```

```
   // navigate through all items
   for(i = 0;i < MAX;i++) {
      printf("%d ",array[i]);
   }

   printf("]\n");
}

void swap(int num1, int num2) {
   int temp = array[num1];
   array[num1] = array[num2];
   array[num2] = temp;
}

int partition(int left, int right, int pivot)
{
   int leftPointer = left -1;
   int rightPointer = right;
   while(true) {
      while(array[++leftPointer] < pivot) {
         //do nothing
      }
      while(rightPointer > 0 && array[--rightPointer] > pivot) {
         //do nothing
      }

      if(leftPointer >= rightPointer) {
         break;
      }
      else {
         printf("item swapped :%d,%d\n", array[leftPointer],array[rightPointer]);
         swap(leftPointer,rightPointer);
      }
   }
   printf("pivot swapped :%d,%d\n", array[leftPointer],array[right]);
   swap(leftPointer,right);
   printf("Updated Array: ");
   display();
   return leftPointer;
}

void quickSort(int left, int right)
{
   if(right-left <= 0) {
      return;
   } else {
      int pivot = array[right];
      int partitionPoint = partition(left, right, pivot);
      quickSort(left,partitionPoint-1);
      quickSort(partitionPoint+1,right);
   }
}
```

```
int main() {
   printf("Input Array: ");
   display();
   quickSort(0,MAX-1);
   printf("Output Array: ");
   display();
}
```

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 $O(n^2)$

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: **$O(n^2)$**
- Best Case Time Complexity: **O(n log n)**
- Average Time Complexity: **O(n log n)**
- Space Complexity: **O(n log n)**

Merge Sort

Merge Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements, recursively, hence consuming less time. If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem. This is the idea behind **Divide and Conquer** algorithm.

In **Merge Sort**, the given unsorted array with n elements, is divided into n subarrays, each having **one** element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

The Selection Sort and Insertion Sort have a worst-case running time of $O(n^2)$. As the size of input grows, insertion and selection sort can take a long time to run. Merge sort, on the other hand, runs in $O(n*\log n)$ time in all the cases.

How Merge Sort Works?

In merge sort, we break the given array midway, for example if the original array had 6 elements, then merge sort will break it down into two subarrays with 3 elements each. So we again break these subarrays into even smaller subarrays, until we have multiple subarrays with **single element** in them. Now, the idea here is that an array with a single element is already sorted, so once we break the original array into subarrays which has only a single element, we have successfully broken down our problem into base problems. Then, we have to merge all these sorted subarrays, step by step to form one single sorted array.

In merge sort we follow the following steps:

1. We take a variable `p` and store the starting index of our array in this. And we take another variable `r` and store the last index of array in it.
2. Then we find the middle of the array using the formula `(p + r)/2` and mark the middle index as `q`, and break the array into two subarrays, from `p` to `q` and from `q + 1` to `r` index.
3. Then we divide these 2 subarrays again, just like we divided our main array and this continues.
4. Once we have divided the main array into subarrays with single elements, then we start merging the subarrays.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.

Pseudocode

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while

   while ( a has elements )
```

```
    add a[0] to the end of c
    remove a[0] from a
  end while

  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while

  return c

end procedure
```

Implementation in C

```
/*
   a[] is the array, p is starting index, that is 0,
   and r is the last index of array.
*/

#include <stdio.h>

// lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

// merge sort function
void mergeSort(int a[], int p, int r)
{
   int q;
   if(p < r)
   {
      q = (p + r) / 2;
      mergeSort(a, p, q);
      mergeSort(a, q+1, r);
      merge(a, p, q, r);
   }
}

// function to merge the subarrays
void merge(int a[], int p, int q, int r)
{
   int b[5];   //same size of a[]
   int i, j, k;
   k = 0;
   i = p;
   j = q + 1;
   while(i <= q && j <= r)
   {
      if(a[i] < a[j])
```

```
        {
            b[k++] = a[i++];    // same as b[k]=a[i]; k++; i++;
        }
        else
        {
            b[k++] = a[j++];
        }
    }

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

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

    for(i=r; i >= p; i--)
    {
        a[i] = b[--k];  // copying back the sorted list to a[]
    }
}

// function to print the array
void printArray(int a[], int size)
{
    int i;
    for (i=0; i < size; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");
}

int main()
{
    int arr[] = {32, 45, 67, 2, 7};
    int len = sizeof(arr)/sizeof(arr[0]);

    printf("Given array: \n");
    printArray(arr, len);

    // calling merge sort
    mergeSort(arr, 0, len - 1);

    printf("\nSorted array: \n");
```

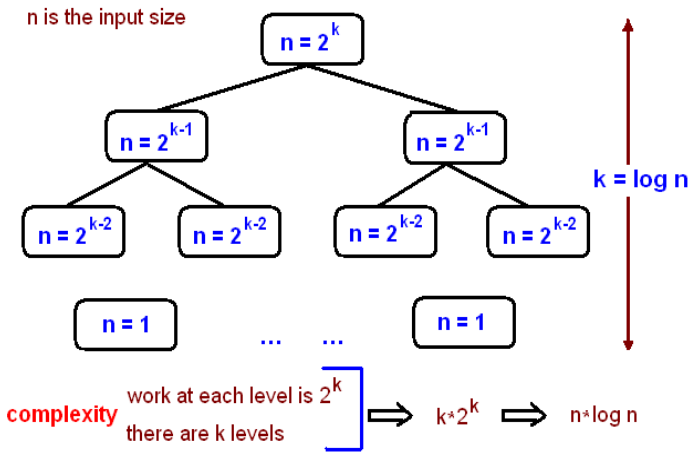
```
    printArray(arr, len);
    return 0;
}
```

Complexity of Mergesort

Suppose T(n) is the number of comparisons needed to sort an array of n elements by the MergeSort algorithm. By splitting an array in two parts we reduced a problem to sorting two parts but smaller sizes, namely n/2. Each part can be sort in T(n/2). Finally, on the last step we perform n-1 comparisons to merge these two parts in one. All together, we have the following equation

$T(n) = 2*T(n/2) + n - 1$

The solution to this equation is beyond the scope of this course. However I will give you a resoning using a binary tree. We visualize the mergesort dividing process as a tree



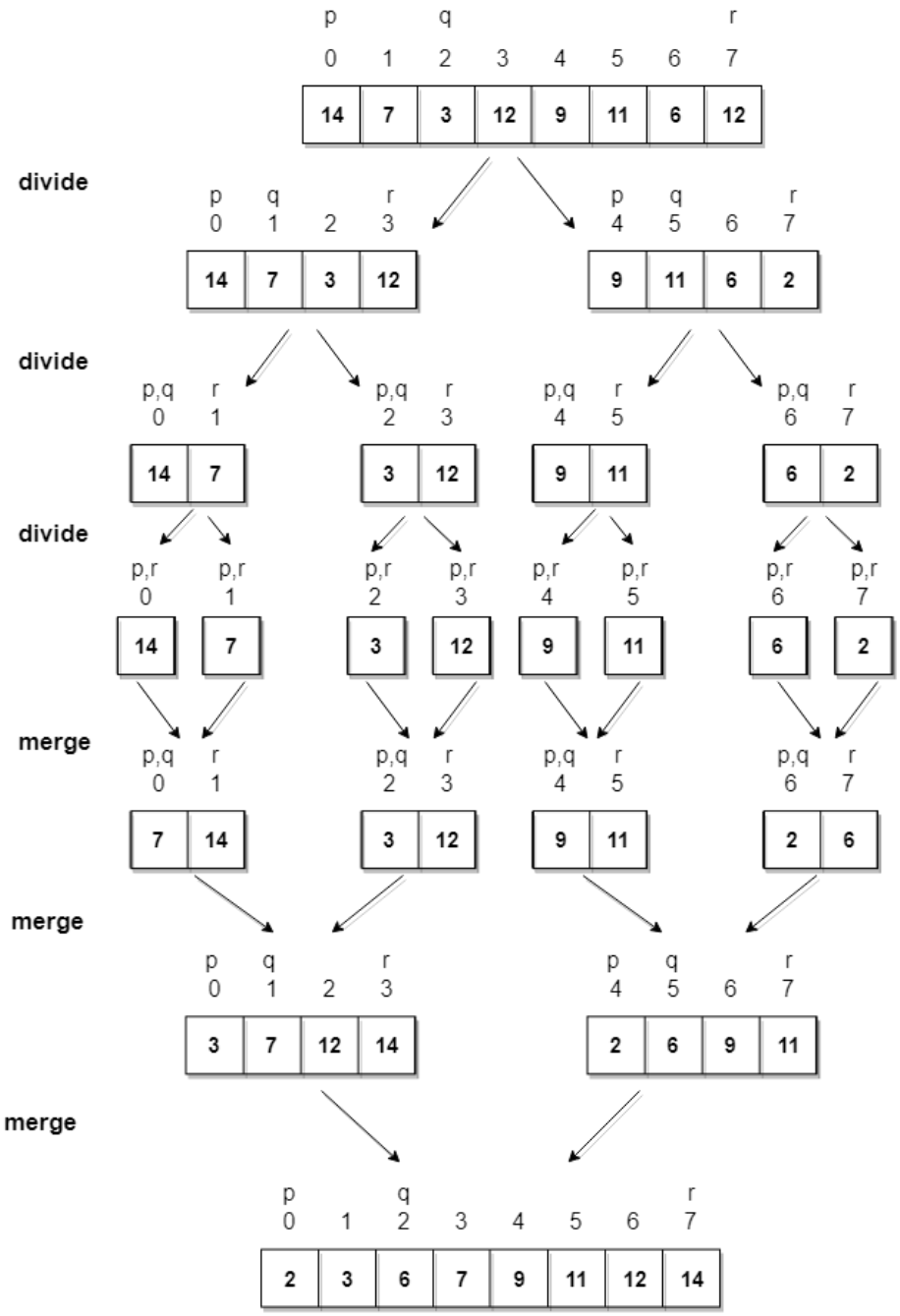
Merge Sort is quite fast, and has a time complexity of O(n*log n). It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

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

Time complexity of Merge Sort is O(n*Log n) in all the 3 cases (worst, average and best) as merge sort always **divides** the array in two halves and takes linear time to **merge** two halves. It requires **equal amount of additional space** as the unsorted array. Hence it's not at all recommended for searching large unsorted arrays.

Example

Let's consider an array with values {14, 7, 3, 12, 9, 11, 6, 2}. Below, we have a pictorial representation of how merge sort will sort the given array.



Radix Sort

Radix sort works by sorting each digit from least significant digit to most significant digit. So in base 10 (the decimal system), radix sort would sort by the digits in the 1's place, then the 10's place, and so on. To do this, radix sort uses counting sort as a subroutine to sort the digits in each place value. This means that for a three-digit number in base 10, counting sort will be called to sort the 1's place, then it will be called to sort the 10's place, and finally, it will be called to sort the 100's place, resulting in a completely sorted list. Radix sort algorithm requires the number of passes which are equal to the number of digits present in the largest number among the list of numbers. For example, if the largest number is a 3 digit number then that list is sorted with 3 passes.

Radix sort is a **stable sort**, which means it preserves the relative order of elements that have the same key value.

Radix sort is an integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits that share the same significant position and value (place value). Radix sort uses counting sort as a subroutine to sort an array of numbers. Because integers can be used to represent strings (by hashing the strings to integers), radix sort works on data types other than just integers.

Radix sort is a non-comparative sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix. For elements with more than one significant digit, this bucketing process is repeated for each digit, while preserving the ordering of the prior step, until all digits have been considered. For this reason, radix sort has also been called bucket sort and digital sort.

Radix sort can be applied to data that can be sorted lexicographically, be they integers, words, punch cards, playing cards, or the mail.

How Radix Sort Works?

1. Find the largest element in the array, i.e. `max`. Let `X` be the number of digits in `max`. The `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`).
3. Now, sort the elements based on digits at tens place.
4. Finally, sort the elements based on the digits at hundreds place.

Algorithm

For each digit i where i varies from the least significant digit to the most significant digit of a number, Sort input array using countsort algorithm according to ith digit.

```
radixSort(array)
  d <- maximum number of digits in the largest element
  create d buckets of size 0-9
  for i <- 0 to d
    sort the elements according to ith place digits using countingSort

countingSort(array, d)
  max <- find largest element among dth place elements
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique digit in dth place of elements and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
```

```
for j <- size down to 1
  restore the elements to array
  decrease count of each element restored by 1
```

Example 1:

Assume the input array is: 10,21,17,34,44,11,654,123

Based on the algorithm, we will sort the input array according to the one's digit (least significant digit).

0: 10

1: 21 11

2:

3: 123

4: 34 44 654

5:

6:

7: 17

8:

9:

So, the array becomes 10,21,11,123,24,44,654,17

Now, we'll sort according to the ten's digit:

0:

1: 10 11 17

2: 21 123

3: 34

4: 44

5: 654

6:

7:

8:

9:

Now, the array becomes : 10,11,17,21,123,34,44,654

Finally , we sort according to the hundred's digit (most significant digit):

0: 010 011 017 021 034 044

1: 123

2:

3:

4:

5:

6: 654

7:

8:

9:

The array becomes : 10,11,17,21,34,44,123,654 which is sorted. This is how our algorithm works.

Radix Sort in C++ Programming

```
#include <iostream>
using namespace std;
int getMax(int array[], int n)
{
  int max = array[0];
  for (int i = 1; i < n; i++)
    if (array[i] > max)
      max = array[i];
  return max;
}
void countingSort(int array[], int size, int place)
{
  const int max = 10;
  int output[size];
  int count[max];

  for (int i = 0; i < max; ++i)
    count[i] = 0;

  for (int i = 0; i < size; i++)
    count[(array[i] / place) % 10]++;

  for (int i = 1; i < max; i++)
    count[i] += count[i - 1];
```

```
  for (int i = size - 1; i >= 0; i--)
  {
    output[count[(array[i] / place) % 10] - 1] = array[i];
    count[(array[i] / place) % 10]--;
  }
  for (int i = 0; i < size; i++)
    array[i] = output[i];
}
void radixsort(int array[], int size)
{
  int max = getMax(array, size);

  for (int place = 1; max / place > 0; place *= 10)
    countingSort(array, size, place);
}
void printArray(int array[], int size)
{
  int i;
  for (i = 0; i < size; i++)
    cout << array[i] << " ";
  cout << endl;
}
int main()
{
  int array[] = {121, 432, 564, 23, 1, 45, 788};
  int n = sizeof(array) / sizeof(array[0]);
  radixsort(array, n);
  printArray(array, n);
}
```

Complexity Analysis

Radix sort will operate on *n d*-digit numbers where each digit can be one of at most *b* different values (since *b* is the base being used). For example, in base 10, a digit can be 0,1,2,3,4,5,6,7,8,or 9.

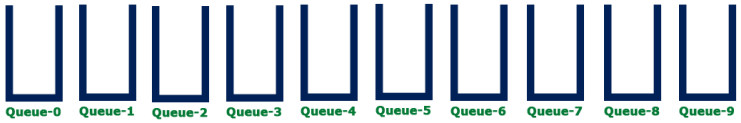
Radix sort uses counting sort on each digit. Each pass over *n d*-digit numbers will take $O(n+b)$ time, and there are *d* passes total. Therefore, the total running time of radix sort is $O(d(n+b))$. When *d* is a constant and *b* isn't much larger than *n* (in other words, $b=O(n)$), then radix sort takes linear time.

Example 2:

Consider the following list of unsorted integer numbers

82, 901, 100, 12, 150, 77, 55 & 23

Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.



Pass - 1

Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

8**2**, 90**1**, 10**0**, 1**2**, 15**0**, 7**7**, 5**5** & 2**3**

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

Pass - 2

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

1**0**0, 1**5**0, 9**0**1, **8**2, **1**2, **2**3, **5**5 & **7**7

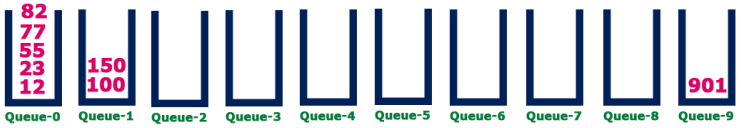
Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

Pass - 3

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundres placed digit) of every number.

100, **9**01, 12, 23, **1**50, 55, 77 & 82



Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the incresing order.

Shell Sort

ShellSort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shellSort is to allow exchange of far items. In shellSort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element is sorted.

Shell sort is an algorithm that first sorts the elements far apart from each other and successively reduces the interval between the elements to be sorted. It is a generalized version of insertion sort.

In shell sort, elements at a specific interval are sorted. The interval between the elements is gradually decreased based on the sequence used. the performance of the shell sort depends on the type of sequence used for a given input array.

Some of the optimal sequences used are:

1. Shell's original sequence: N/2 , N/4 , …, 1
2. Knuth's increments: 1, 4, 13, …, (3k – 1) / 2
3. Sedgewick's increments: 1, 8, 23, 77, 281, 1073, 4193, 16577...4j+1+ 3·2j+ 1.
4. Hibbard's increments: 1, 3, 7, 15, 31, 63, 127, 255, 511…
5. Papernov & Stasevich increment: 1, 3, 5, 9, 17, 33, 65,...
6. Pratt: 1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, 36, 54, 81....

How Shell Sort Works?

1. Suppose, we need to sort the following array.

2. We are using the shell's original sequence (N/2, N/4, ...1) as intervals in our algorithm.

   In the first loop, if the array size is N = 8 then, the elements lying at the interval of N/2 = 4 are compared and swapped if they are not in order.

   a. The 0th element is compared with the 4th element.
   b. If the 0th element is greater than the 4th one then, the 4th element is first stored in temp variable and the 0th element (ie. greater element) is stored in the 4th position and the element stored in temp is stored in the 0th position.

   This process goes on for all the remaining elements.

3. In the second loop, an interval of N/4 = 8/4 = 2 is taken and again the elements lying at these intervals are sorted. The same process goes on for remaining elements.
4. Finally, when the interval is N/8 = 8/8 =1 then the array elements lying at the interval of 1 are sorted. The array is now completely sorted.

Algorithm

Following is the algorithm for shell sort.

Step 1 − Initialize the value of *h*
Step 2 − Divide the list into smaller sub-list of equal interval *h*
Step 3 − Sort these sub-lists using **insertion sort**
Step 3 − Repeat until complete list is sorted

Shell Sort in C++ programming

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

void shellSort(int array[], int n)
{
  for (int gap = n / 2; gap > 0; gap /= 2)
  {
    for (int i = gap; i < n; i += 1)
    {
      int temp = array[i];
      int j;
      for (j = i; j >= gap && array[j - gap] > temp; j -= gap)
      {
        array[j] = array[j - gap];
      }
      array[j] = temp;
    }
  }
}
void printArray(int array[], int size)
{
  int i;
  for (i = 0; i < size; i++)
    cout << array[i] << " ";
  cout << endl;
}
int main()
{
  int data[] = {9, 8, 3, 7, 5, 6, 4, 1};
  int size = sizeof(data) / sizeof(data[0]);
  shellSort(data, size);
  cout << "Sorted array: \n";
  printArray(data, size);
}
```

HeapSort

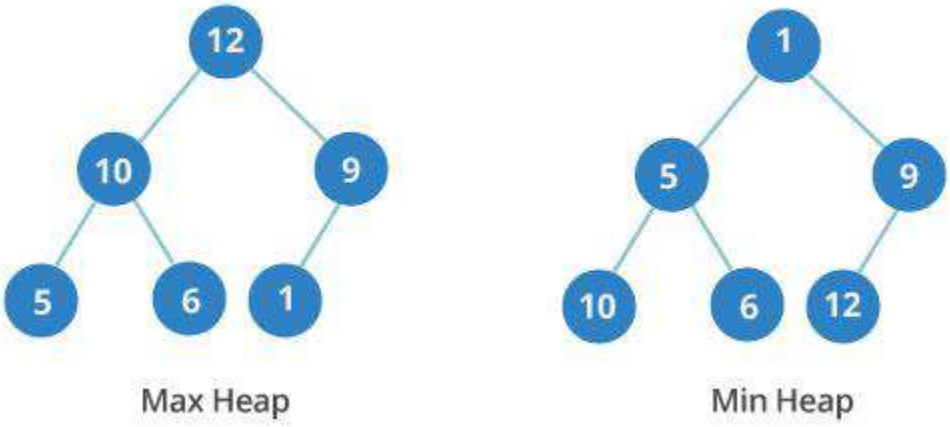
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

What is Heap Data Structure ?

Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if

1. it is a complete binary tree (**shape property**).
2. All nodes in the tree follow the property that they are greater than or equal to their children i.e. the largest element is at the root and both its children and smaller than the root and so on (**Heap property**). Such a heap is called a max-heap. If instead all nodes are smaller than or equal to their children, it is called a min-heap

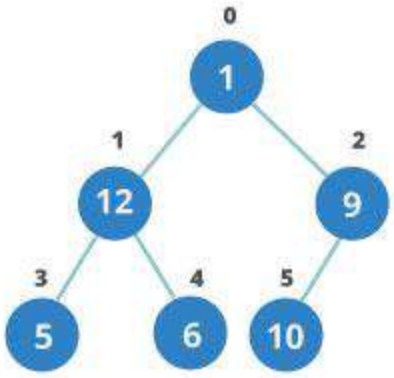
Following example diagram shows Max-Heap and Min-Heap.



Relationship between array indexes and tree elements

Complete binary tree has an interesting property that we can use to find the children and parents of any node.

If the index of any element in the array is i, the element in the index $2i+1$ will become the left child and element in $2i+2$ index will become the right child. Also, the parent of any element at index i is given by the lower bound of $(i-1)/2$.

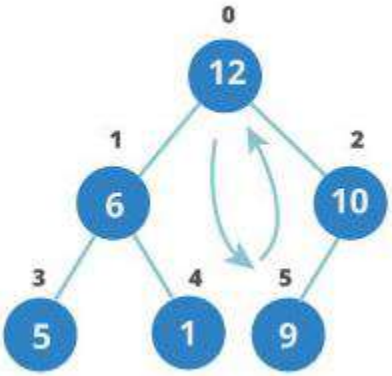


0	1	2	3	4	5
1	12	9	5	6	10

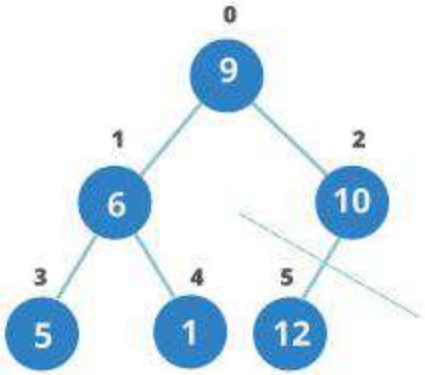
Heap Sort Algorithm

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1.
3. Heapify the root of tree.
4. Repeat above steps while size of heap is greater than 1.

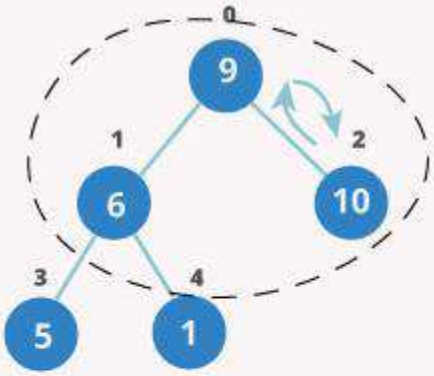
Example



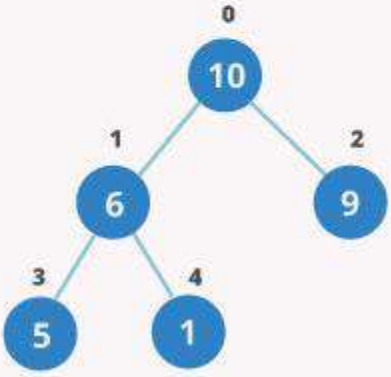
0	1	2	3	4	5
12	6	10	5	1	9



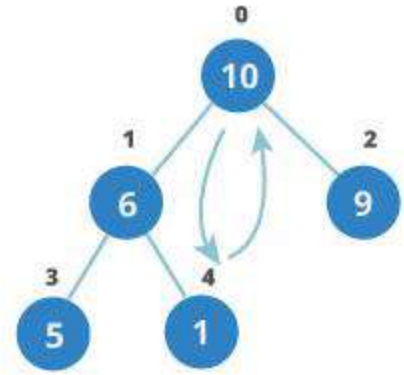
0	1	2	3	4	5
9	6	10	5	1	12

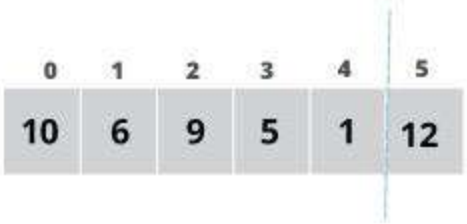


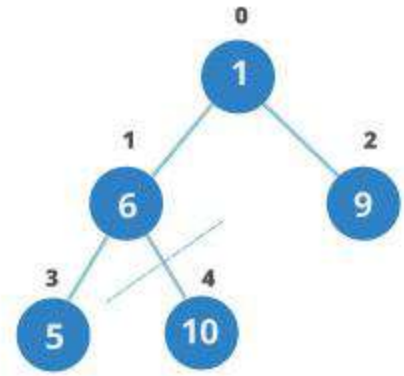
0	1	2	3	4	5
9	6	10	5	1	12

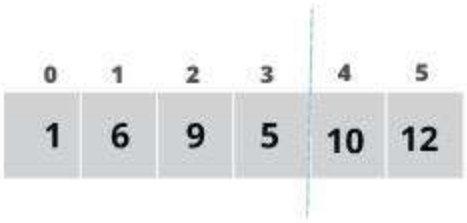


0	1	2	3	4	5
10	6	9	5	1	12





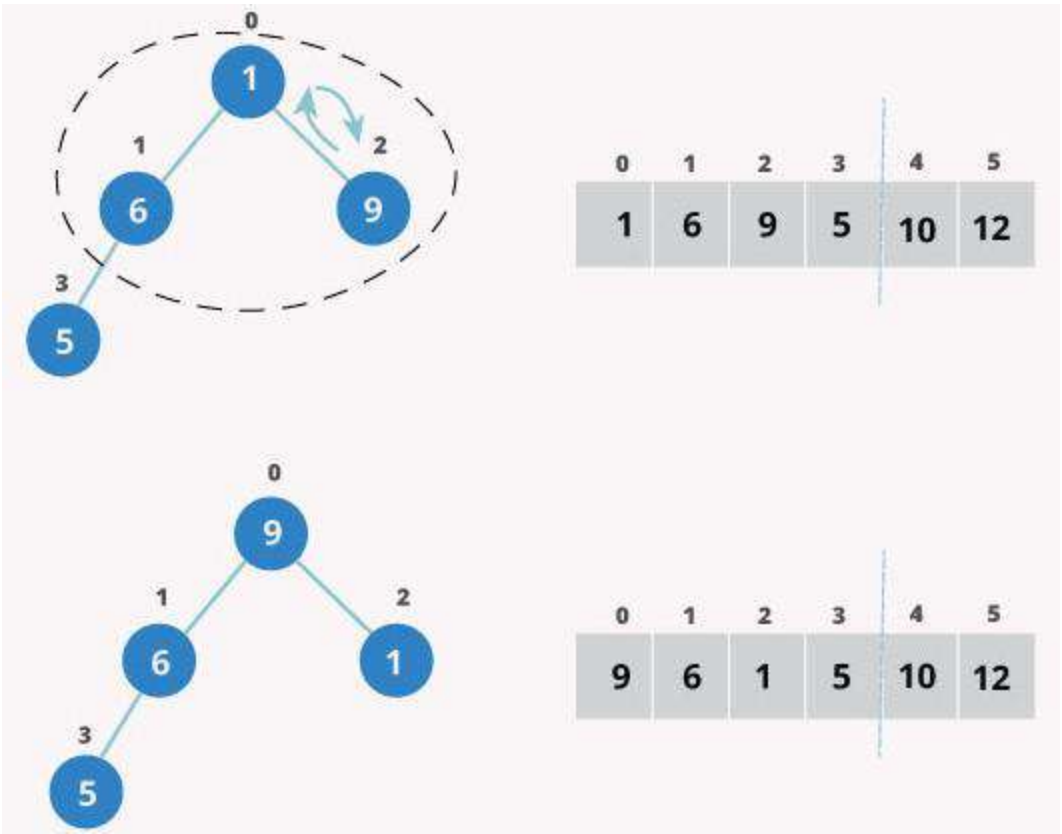


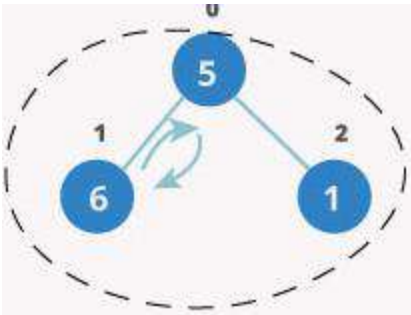


C++ program for implementation of Heap Sort

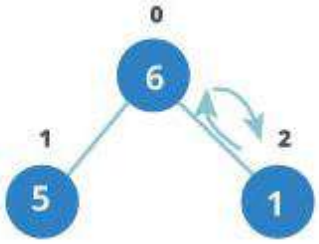
```
#include <iostream>
using namespace std;

// 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
    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";
}

int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    heapSort(arr, n);

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

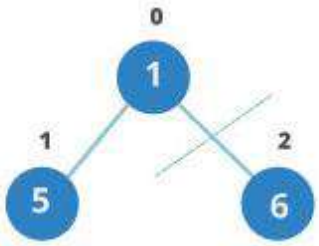

0
1
1
6
2
9
3
5
0 1 2 3 4 5
1 6 9 5 10 12
0
9
1
6
2
1
3
5
0 1 2 3 4 5
9 6 1 5 10 12
0
5
1
6
2
1
3
9
0 1 2 3 4 5
5 6 1 9 10 12
0
5
1
6
2
1
3
9
0 1 2 3 4 5
5 6 1 9 10 12



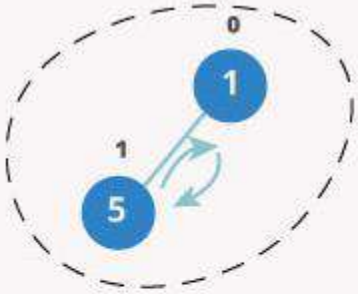
0	1	2	3	4	5
5	6	1	9	10	12



0	1	2	3	4	5
6	5	1	9	10	12

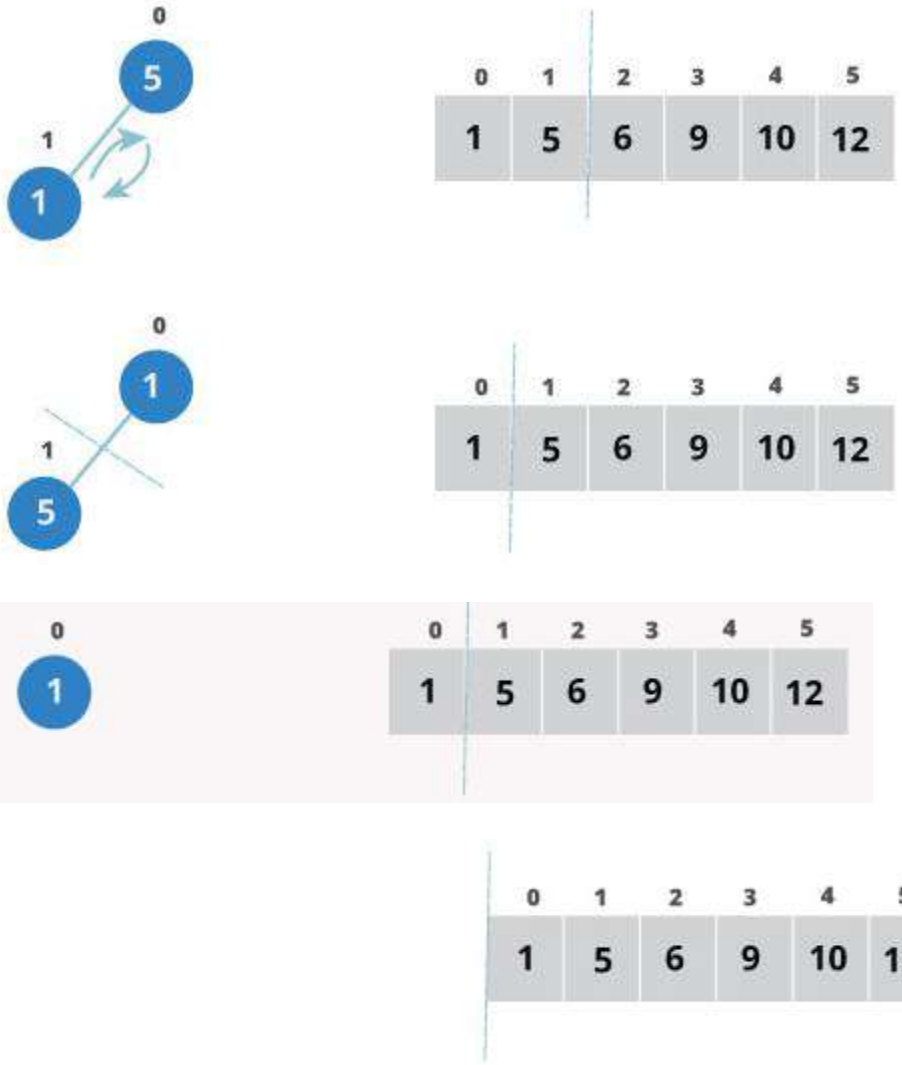


0	1	2	3	4	5
1	5	6	9	10	12



0	1	2	3	4	5
1	5	6	9	10	12

0	1	2	3	4	5
1	5	6	9	10	12



Implementation of Priority Queue

Priority queue can be implemented using an array, a linked list, a heap data structure or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.

A comparative analysis of different implementations of priority queue is given below.

	peek	insert	delete
Linked List	O(1)	O(n)	O(1)
Binary Heap	O(1)	O(log n)	O(log n)
Binary Search Tree	O(1)	O(log n)	O(log n)

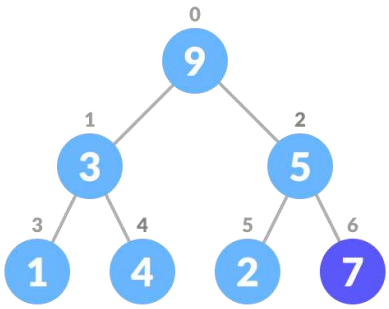
Priority Queue Operations

Basic operations of a priority queue are inserting, removing and peeking elements.

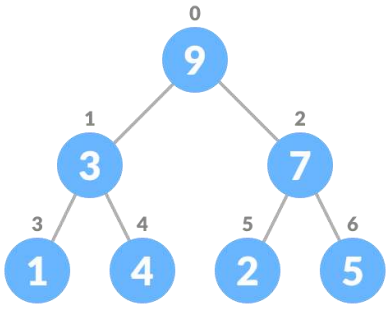
Inserting an Element into the Priority Queue

Inserting an element into a priority queue (max heap) is done by following steps.

1. Insert the new element at the end of the tree.



2. Heapify the tree.



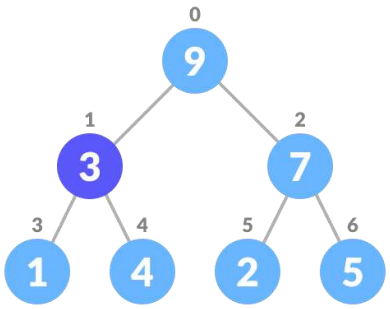
Algorithm for insertion of an element into priority queue (max-heap)

```
If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to right.)

heapify the array
```

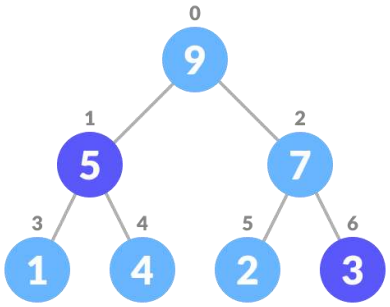
Deleting an Element from the Priority Queue

Deleting an element from a priority queue (max heap) is done as follows:

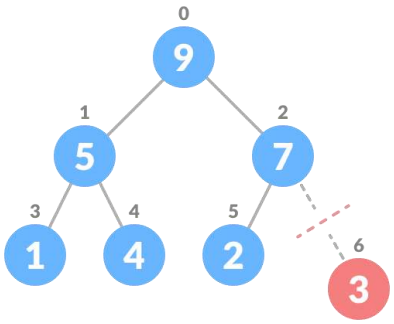
1. Select the element to be deleted.



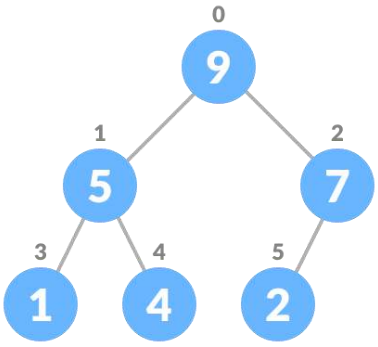
2. Swap it with the last element.



3. Remove the last element.



4. Heapify the tree.



Algorithm for deletion of an element in the priority queue (max-heap)

```
If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode
  remove noteToBeDeleted

heapify the array
```

Peeking from the Priority Queue (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

For both Max heap and Min Heap

```
return rootNode
```

Priority Queue Applications

Some of the applications of a priority queue are:

1. CPU Scheduling
2. For Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3. for implementing stack
4. for load balancing and interrupt handling in an operating system
5. for data compression in Huffman code