Unit 6 (Sorting)

1. Types of sorting: internal and external
2. Insertion and selection sort
3. Exchange sort
4. Merge and Radix sort
5. Shell sort
6. Heap sort as a priority queue

Introduction:

Sorting

Sorting is among the most basic problems in algorithm design. We are given a sequence of items, each associated with a given key value. The problem is to permute the items so that they are in increasing (or decreasing) order by key. Sorting is important because it is often the first step in more complex algorithms. Sorting algorithms are usually divided into two classes, **internal sorting algorithms**, which assume that data is stored in an array in main memory, and **external sorting algorithm**, which assume that data is stored on disk or some other device that is best accessed sequentially. We will only consider internal sorting. Sorting algorithms often have additional properties that are of interest, depending on the application. Here are two important properties.

In brief the sorting is a process of arranging the items in a list in some order that is either ascending or descending order.

Let a[n] be an array of n elements a0,a1,a2,a3.....,an-1 in memory. The sorting of the array a[n] means arranging the content of a[n] in either increasing or decreasing order. i.e. a0<=a1<=a2<=a3<.=.....<=an-1

consider a list of values: 2 ,4 ,6 ,8 ,9 ,1 ,22 ,4 ,77 ,8 ,9

After sorting the values: 1, 2, 4, 4, 6, 8, 8,9 , 9 , 22, 77

Insertion Sort:

Idea: like sorting a hand of playing cards start with an empty left hand and the cards facing down on the table. Remove one card at a time from the table, and insert it into the correct position in the left hand. Compare it with each of the cards already in the hand, from right to left. The cards held in the left hand are 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.

Consider the following example of an unsorted array that we will sort with the help of the Insertion Sort algorithm.

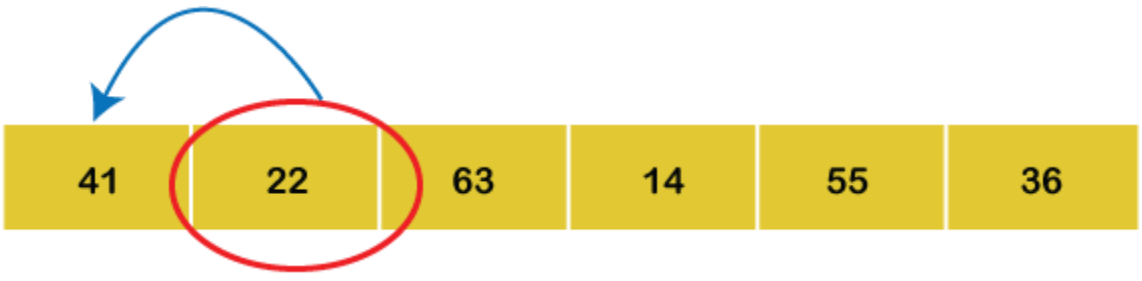
A = (41, 22, 63, 14, 55, 36)

Initially,

1st Iteration:

Set key = 22

Compare a1 with a0

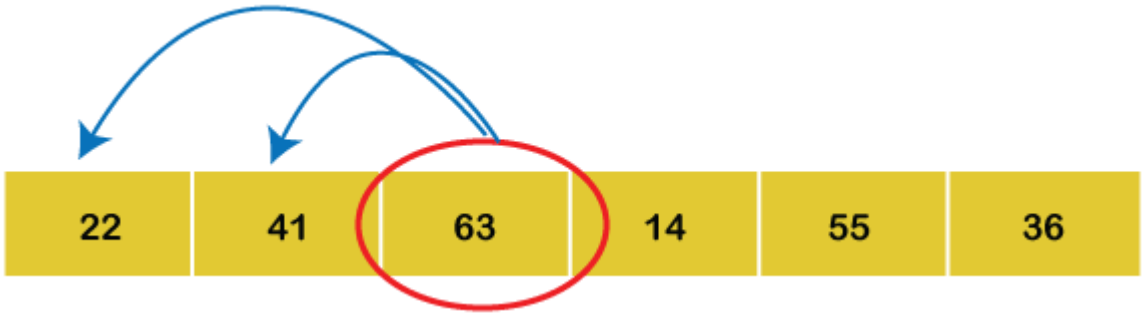


Since a0 > a1, swap both of them.

2nd Iteration:

Set key = 63

Compare a2 with a1 and a0



Since a2 > a1 > a0, keep the array as it is.

3rd Iteration:

Set key = 14

Compare a3 with a2, a1 and a0

Since a3 is the smallest among all the elements on the left-hand side, place a3 at the beginning of the array.

4th Iteration:

Set key = 55

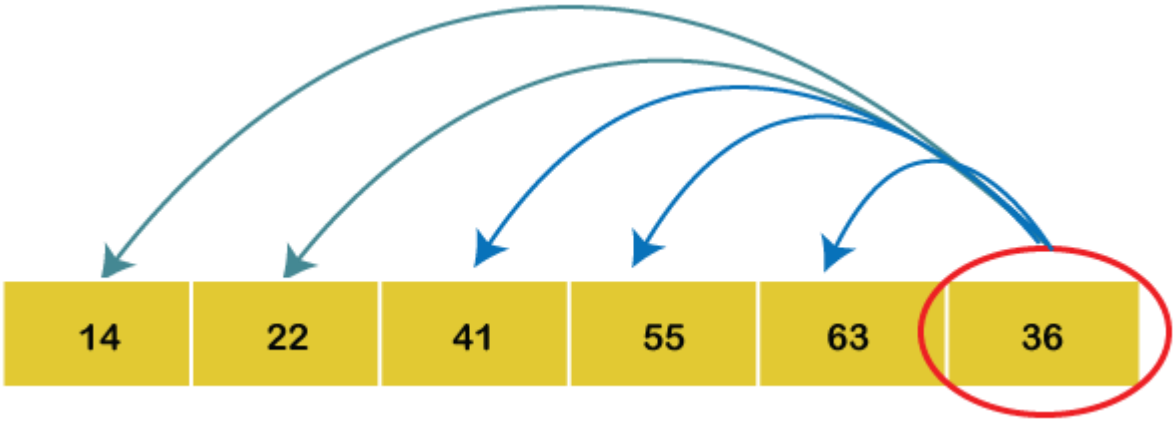
Compare a4 with a3, a2, a1 and a0.

As a4 < a3, swap both of them.

5th Iteration:

Set key = 36

Compare a5 with a4, a3, a2, a1 and a0.

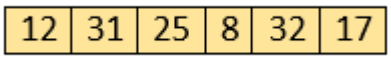


Since a5 < a2, so we will place the elements in their correct positions.

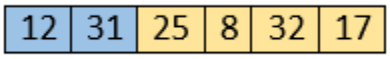
Hence the array is arranged in ascending order, so no more swapping is required.

Example 2:

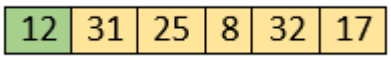
Let the elements of array are -



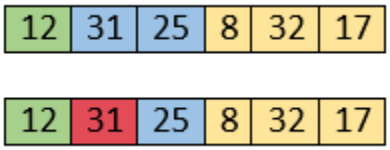
Initially, the first two elements are compared in insertion sort.



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.

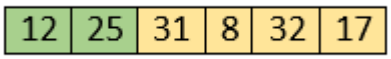


Now, move to the next two elements and compare them.

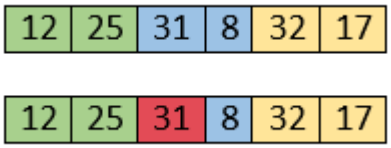


Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

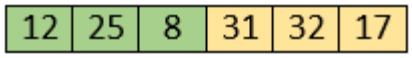
For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.



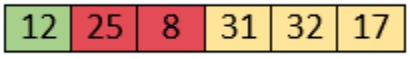
Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.



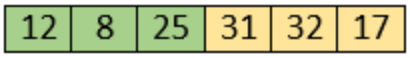
Both 31 and 8 are not sorted. So, swap them.



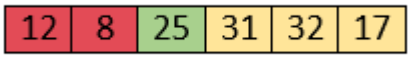
After swapping, elements 25 and 8 are unsorted.



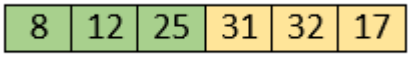
So, swap them.



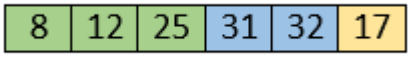
Now, elements 12 and 8 are unsorted.



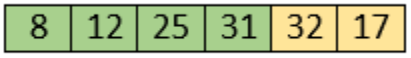
So, swap them too.



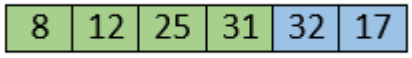
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



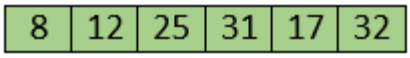
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

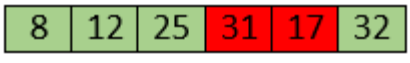


Move to the next elements that are 32 and 17.

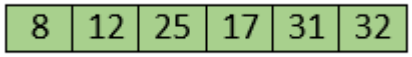


17 is smaller than 32. So, swap them.

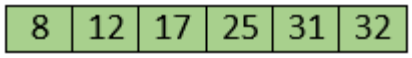




Swapping makes 31 and 17 unsorted. So, swap them too.



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



Now, the array is completely sorted.

Complexity Analysis of Insertion Sort

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given **n** elements, then in the first pass, it will make **n-1** comparisons; in the second pass, it will do **n-2**; in the third pass, it will do **n-3** and so on. Thus, the total number of comparisons can be found by;

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

Sum=

i.e., $O(n^2)$

Therefore, the insertion sort algorithm encompasses a time complexity of **$O(n^2)$** and a space complexity of **O(1)** because it necessitates some extra memory space for a **key** variable to perform swaps.

Time Complexities:

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

Compiled by: Er. Sandesh S Poudel

- **Best Case Complexity:** The insertion sort algorithm has a best-case time complexity of **O(n)** for the already sorted array because here, only the outer loop is running n times, and the inner loop is kept still.
- **Average Case Complexity:** The average-case time complexity for the insertion sort algorithm is **$O(n^2)$**, which is incurred when the existing elements are in jumbled order, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also **$O(n^2)$**, which occurs when we sort the ascending order of an array into the descending order.
  In this algorithm, every individual element is compared with the rest of the elements, due to which n-1 comparisons are made for every n^{th} element.

The insertion sort algorithm is highly recommended, especially when a few elements are left for sorting or in case the array encompasses few elements.

Space Complexity

Space Complexity	**O(1)**
Stable	YES

The insertion sort encompasses a space complexity of **O(1)** due to the usage of an extra variable **key**.

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

Insertion Sort Applications

The insertion sort algorithm is used in the following cases:

- When the array contains only a few elements.
- When there exist few elements to sort.

Advantages of Insertion Sort

1. It is simple to implement.
2. It is efficient on small datasets.
3. It is stable (does not change the relative order of elements with equal keys)
4. It is in-place (only requires a constant amount O (1) of extra memory space).

5. It is an online algorithm, which can sort a list when it is received.

Implementation of insertion sort

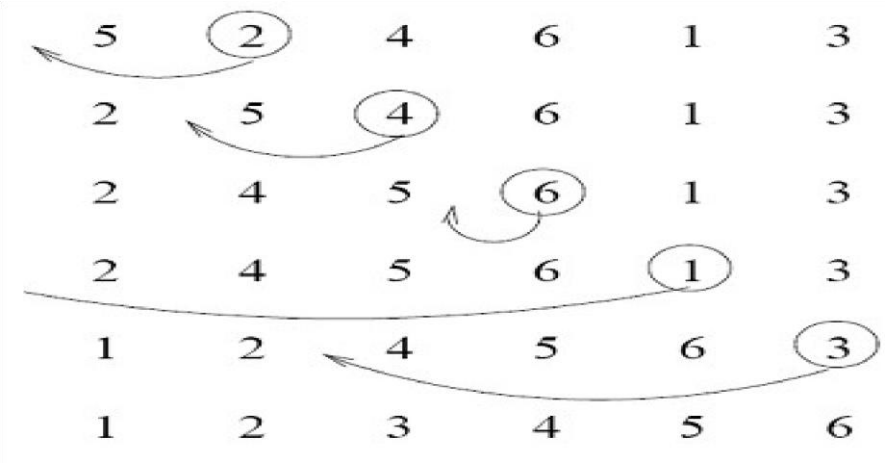
Now, let's see the programs of insertion sort in different programming languages.

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

```
#include <stdio.h>

void insert(int a[], int n) /* function to sort an aay with insertion sort */

{

   int i, j, temp;

   for (i = 1; i < n; i++) {

      temp = a[i];

      j = i - 1;

 while(j>=0 && temp <= a[j])  /* Move the elements greater than temp to one position ahead from their current position*/

      {

         a[j+1] = a[j];

         j = j-1;

      }

      a[j+1] = temp;

   }

}

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

{

   int i;

   for (i = 0; i < n; i++)
```

```
        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);
        insert(a, n);
        printf("\nAfter sorting array elements are - \n");
        printArr(a, n);
        return 0;
    }
```

Example 3:



```
Algorithm:
InsertionSort(A)
        {
                for (i=1;i<n;i++)
                {
                        key = A[ i]
                        for(j=i; j>0 && A[j] >key; j--)
                        {
                                A[j + 1] = A[j]
                        }
                        A[j + 1] = key

                }
        }
```

Selection Sort:

Idea: Find the least (or greatest) value in the array, swap it into the leftmost(or rightmost) component (where it belongs), and then forget the leftmost component. Do this repeatedly. Let a[n] be a linear array of n elements.

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

The selection sort works as follows:

pass 1: Find the location loc of the smallest element int the list of n elements a[0], a[1], a[2], a[3], ….....,a[n-1] and then interchange a[loc] and a[0].

Pass 2: Find the location loc of the smallest element int the sub-list of n-1 elements a[1], a[2], a[3], ….....,a[n-1] and then interchange a[loc] and a[1] such that a[0], a[1] are sorted.

…..... and so on.

Then we will get the sorted list a[0]<=a[1]<= a[2]<=a[3]<= ….....<=a[n-1].

Algorithm:

```
SelectionSort(A)
{
        for( i = 0;i < n ;i++)
        {
```

```
                least=A[i]; p=i;
                for ( j = i + 1;j < n ;j++)
                { if (A[j] < A[i])
                        least= A[j]; p=j;
}
            }
            swap(A[i],A[p]);
        }
```

Working of Selection sort Algorithm

Consider the following example of an unsorted array that we will sort with the help of the Selection Sort algorithm.

A [] = (7, 4, 3, 6, 5).

A [] =

1st Iteration:

Set minimum = 7

- Compare a_0 and a_1

As, $a_0 > a_1$, set minimum = 4.

- Compare a_1 and a_2

As, $a_1 > a_2$, set minimum = 3.

- Compare a_2 and a_3

As, $a_2 < a_3$, set minimum= 3.

- Compare a_2 and a_4

As, $a_2 < a_4$, set minimum =3.

Since 3 is the smallest element, so we will swap a_0 and a_2.

2nd Iteration:

Set minimum = 4

- Compare a_1 and a_2

As, $a_1 < a_2$, set minimum = 4.

- Compare a_1 and a_3

As, A[1] < A[3], set minimum = 4.

- Compare a_1 and a_4

Again, $a_1 < a_4$, set minimum = 4.

Since the minimum is already placed in the correct position, so there will be no swapping.

3rd Iteration:

Set minimum = 7

- Compare a_2 and a_3

As, $a_2 > a_3$, set minimum = 6.

- Compare a_3 and a_4

As, $a_3 > a_4$, set minimum = 5.

Since 5 is the smallest element among the leftover unsorted elements, so we will swap 7 and 5.

4th Iteration:

Set minimum = 6

- Compare a_3 and a_4

As $a_3 < a_4$, set minimum = 6.

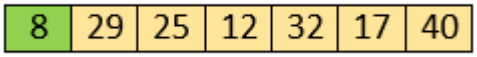
Since the minimum is already placed in the correct position, so there will be no swapping.

Example 2: Let the elements of array are -

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

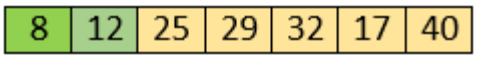
At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

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

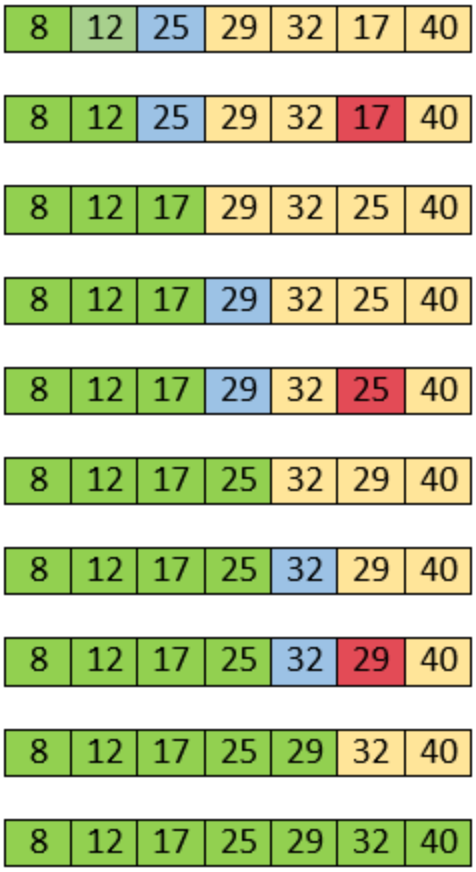


For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

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	$O(n^2)$

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + ……. +2 +1

$= O(n^2)$

There is no best-case linear time complexity for this algorithm, but number of swap operations is reduced greatly.

Compiled by: Er. Sandesh S Poudel

- **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

Since no extra space besides 5 variables is needed for sorting Space complexity = O(n)

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

Now, let's see the programs of selection sort in different programming languages.

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;
}
```

Exchange Sort:

Bubble Sort:

The basic idea of this sort is to pass through the array sequentially several times. Each pass consists of comparing each element in the array with its successor (for example a[i] with a[i + 1]) and interchanging the two elements if they are not in the proper order. For example, consider the following array:

Algorithm

```
BubbleSort(A, n)
{
        for(i = 0; i <n-1; i++)
        {
```

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

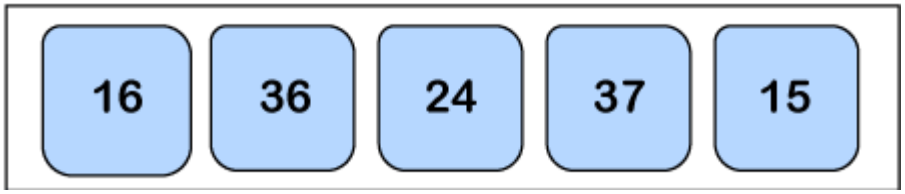
Working of Bubble sort Algorithm

Now, let's see the 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 **$O(n^2)$.**

Consider the following example of an unsorted array that we will sort with the help of the Bubble Sort algorithm.

Initially,



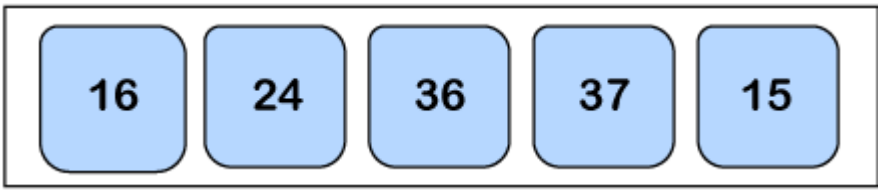
Pass 1:

- **Compare a_0 and a_1**

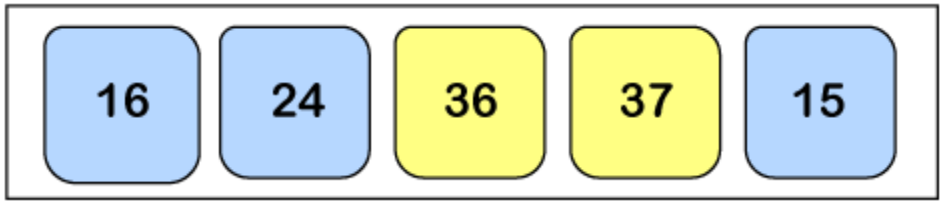
As $a_0 < a_1$ so the array will remain as it is.

- **Compare a_1 and a_2**

Now $a_1 > a_2$, so we will swap both of them.

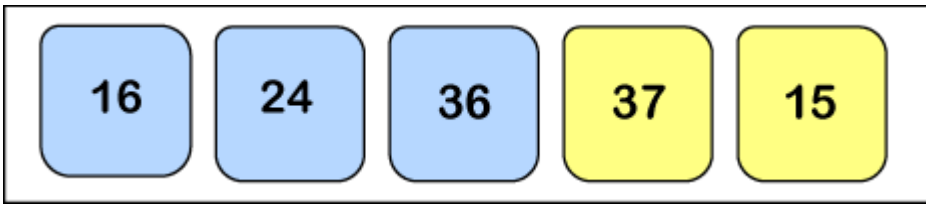


- **Compare a_2 and a_3**

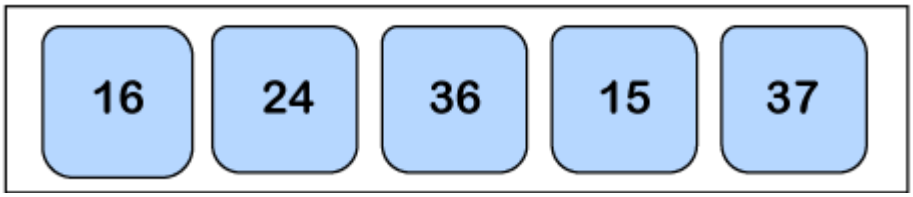


As $a_2 < a_3$ so the array will remain as it is.

- **Compare a_3 and a_4**

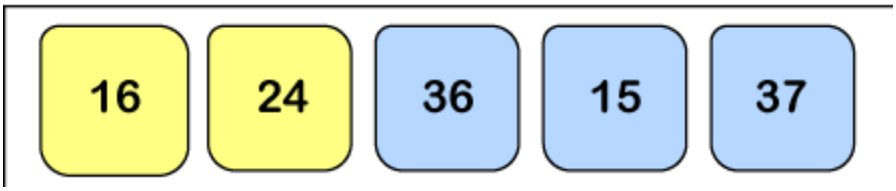


Here $a_3 > a_4$, so we will again swap both of them.



Pass 2:

- **Compare a_0 and a_1**

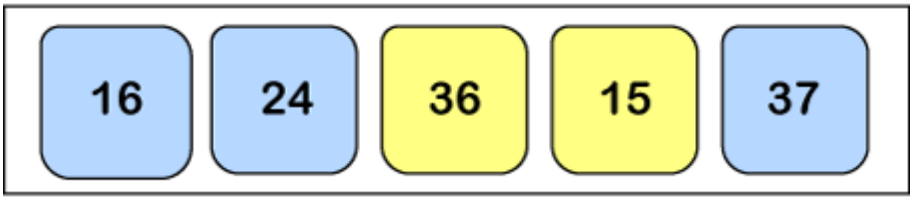


As $a_0 < a_1$ so the array will remain as it is.

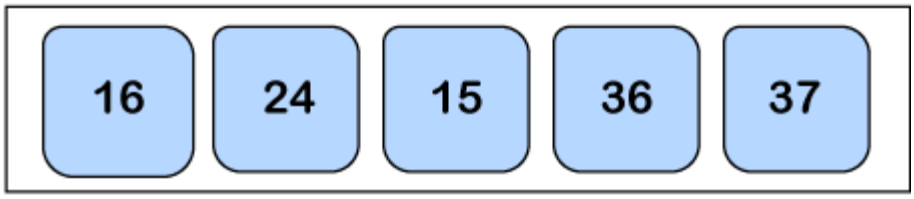
- **Compare a_1 and a_2**

Here $a_1 < a_2$, so the array will remain as it is.

- **Compare a_2 and a_3**

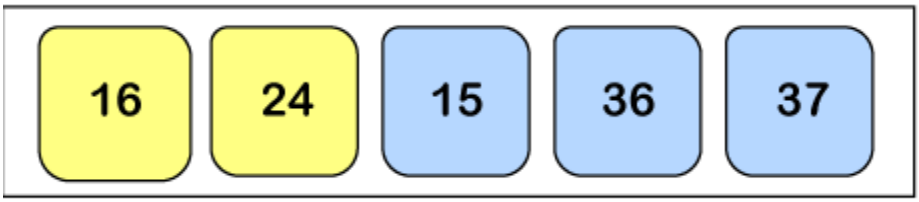


In this case, $a_2 > a_3$, so both of them will get swapped.



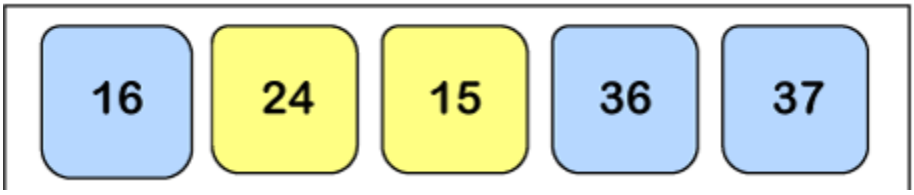
Pass 3:

- **Compare a_0 and a_1**



As $a_0 < a_1$ so the array will remain as it is.

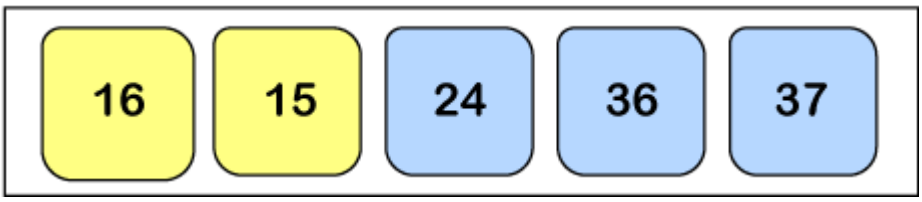
- **Compare a_1 and a_2**



Now $a_1 > a_2$, so both of them will get swapped.

Pass 4:

- **Compare a_0 and a_1**

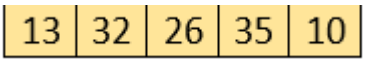


Here $a_0 > a_1$, so we will swap both of them.

Hence the array is sorted as no more swapping is required.

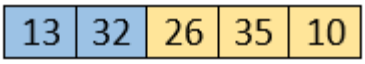
Example 2:

Let the elements of array are -

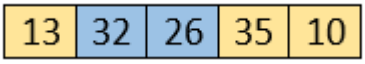


First Pass

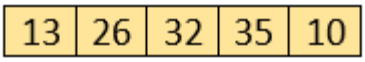
Sorting will start from the initial two elements. Let compare them to check which is greater.



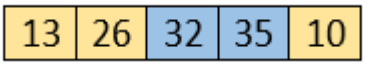
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



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

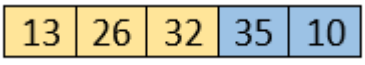


Now, compare 32 and 35.



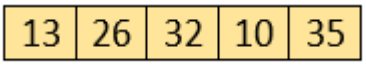
Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

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.

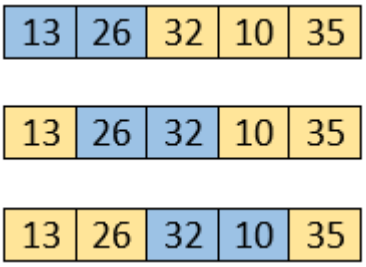
After first pass, the array will be -



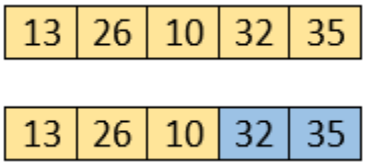
Now, move to the second iteration.

Second Pass

The same process will be followed for second iteration.



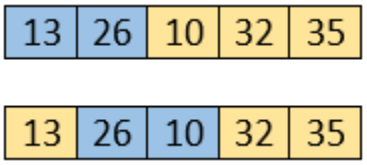
Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -



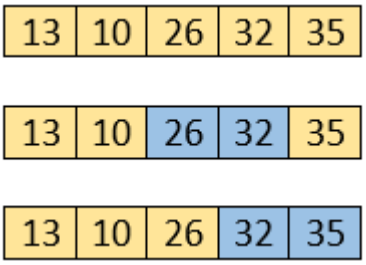
Now, move to the third iteration.

Third Pass

The same process will be followed for third iteration.



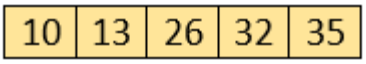
Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



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.

Example: 3

Initially:

25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7

After Pass 1:

25	48	37	12	57	86	33	92
0	1	2	3	4	5	6	7

After Pass 2:

25	37	12	48	57	33	86	92
0	1	2	3	4	5	6	7

After Pass 3:

25	12	37	48	33	57	86	92
0	1	2	3	4	5	6	7

After Pass 4:

12	25	37	33	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 5:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 6:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 7:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

Here, we notice that after each pass, an element is placed in its proper order and is not considered in succeeding passes. Furthermore, we need n – 1 passes to sort n elements.

Bubble sort complexity

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

Complexity Analysis of Bubble Sort

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given n elements, then in the first pass, it will do n-1 comparisons; in the second pass, it will do n-2; in the third pass, it will do n-3 and so on. Thus, the total number of comparisons can be found by;

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$

Therefore, the bubble sort algorithm encompasses a time complexity of **$O(n^2)$** and a space complexity of **O(1)** because it necessitates some extra memory space for temp variable for swapping.

1. Time Complexity

Case	Time Complexity
Best Case	O(n)
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 bubble sort is **O(n).**
- **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 bubble 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 bubble sort is **$O(n^2)$.**

2. Space Complexity

Space Complexity	**O(1)**
Stable	YES

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

Advantages of Bubble Sort

1. Easily understandable.
2. Does not necessitates any extra memory.
3. The code can be written easily for this algorithm.
4. Minimal space requirement than that of other sorting algorithms.

Compiled by: Er. Sandesh S Poudel

Disadvantages of Bubble Sort

1. It does not work well when we have large unsorted lists, and it necessitates more resources that end up taking so much of time.
2. It is only meant for academic purposes, not for practical implementations.
3. It involves the n^2 order of steps to sort an algorithm.

Implementation of Bubble sort

```
#include<stdio.h>
 void print(int a[], int n) //function to print array elements
    {
    int i;
    for(i = 0; i < n; i++)
    {
        printf("%d ",a[i]);
    }
    }
 void bubble(int a[], int n) // function to implement bubble sort
 {
   int i, j, temp;
   for(i = 0; i < n; i++)
    {
      for(j = i+1; j < n; j++)
        {
            if(a[j] < a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
 }
void main ()
{
    int i, j,temp;
    int a[5] = { 10, 35, 32, 13, 26};
    int n = sizeof(a)/sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    print(a, n);
    bubble(a, n);
    printf("\nAfter sorting array elements are - \n");
    print(a, n);
}
```

Quick Sort:

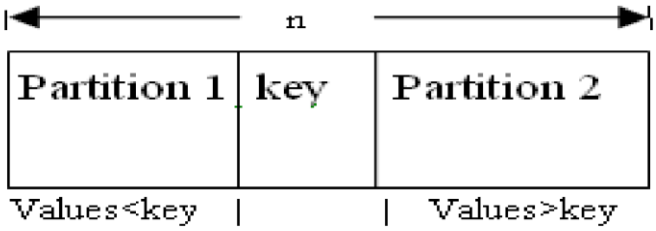
Quick sort developed by C.A.R Hoare is an unstable sorting. In practice this is the fastest sorting method. It possesses very good average case complexity among all the sorting algorithms. This algorithm is based on the divide and conquer paradigm. The main idea behind this sorting is partitioning of the elements.

Steps for Quick Sort:

Divide: partition the array into two nonempty sub arrays.

Conquer: two sub arrays are sorted recursively.

Combine: two sub arrays are already sorted in place so no need to combine.



Algorithm:

```
QuickSort(A,l,r)
{
        f(l<r)
        {
                p = Partition(A,l,r); QuickSort(A,l,p-1);
                QuickSort(A,p+1,r);
        }
}

Partition(A,l,r)
{
        x =l; y =r ; p = A[l]; while(x<y)
        { while(A[x] <= p) x++;
                while(A[y] >=p)
                        y--;
                if(x<y) swap(A[x],A[y]);
```

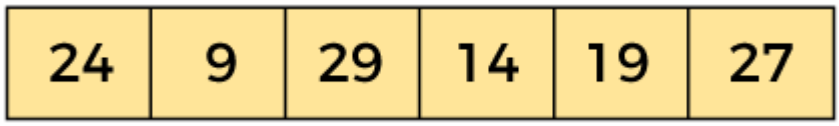
```
}
A[l] = A[y]; A[y] = p;
return y;          //return position of pivot }
```

Working of Quick Sort Algorithm

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

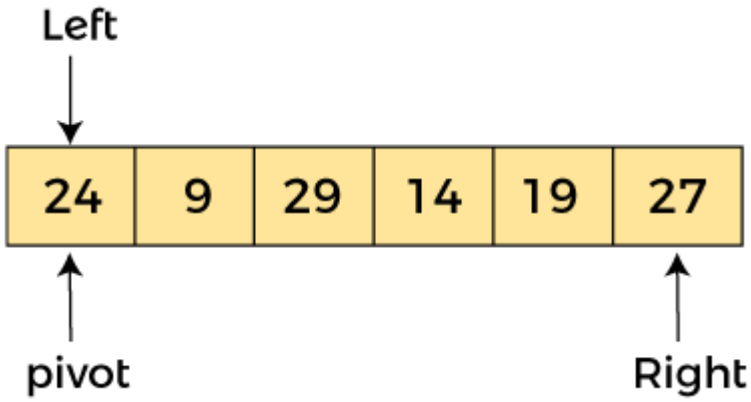
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

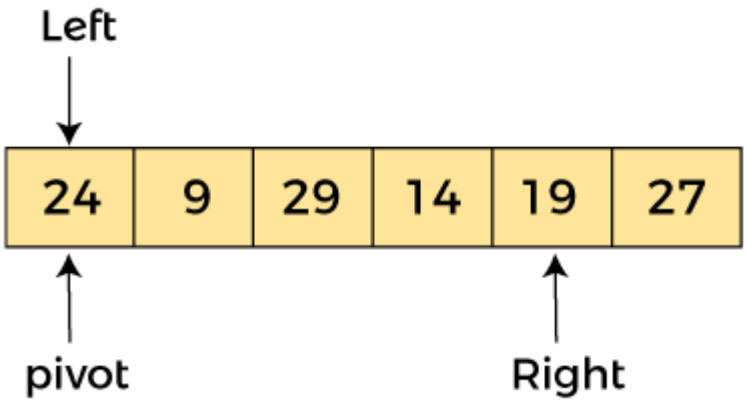


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

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

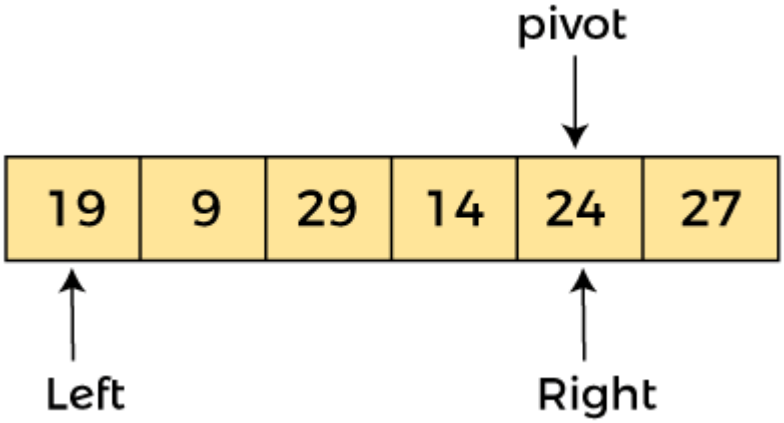


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



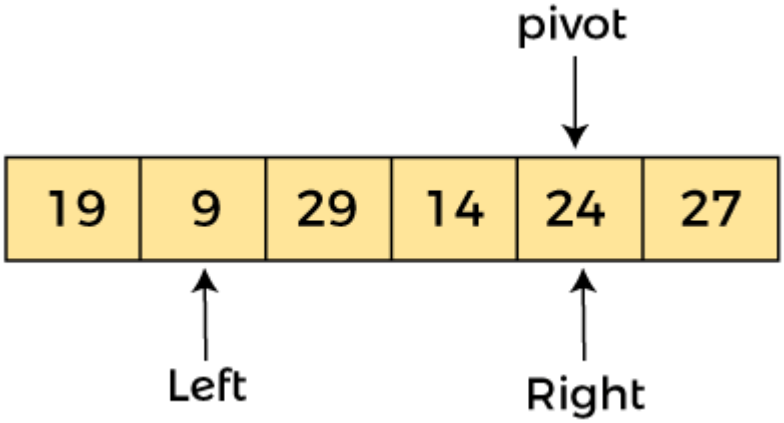
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

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

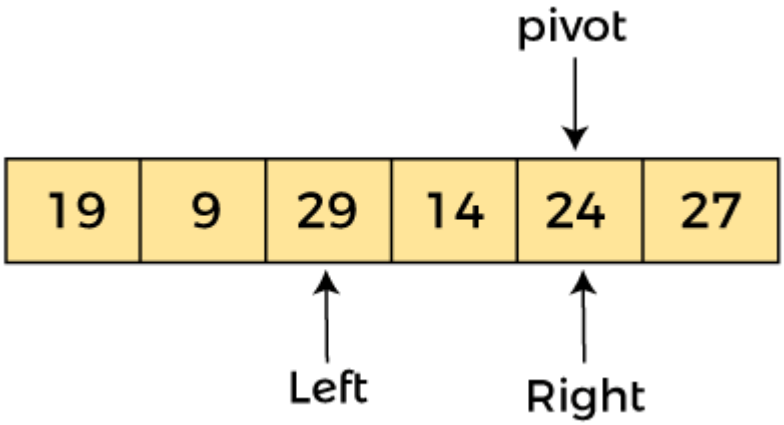


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

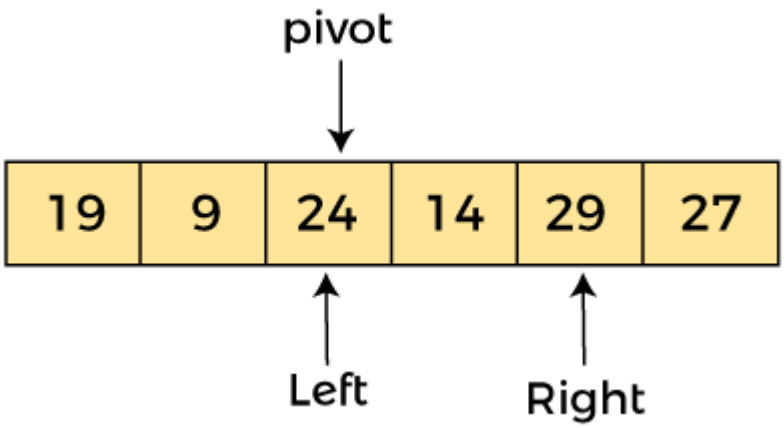
As a[pivot] > a[left], so algorithm moves one position to right as -



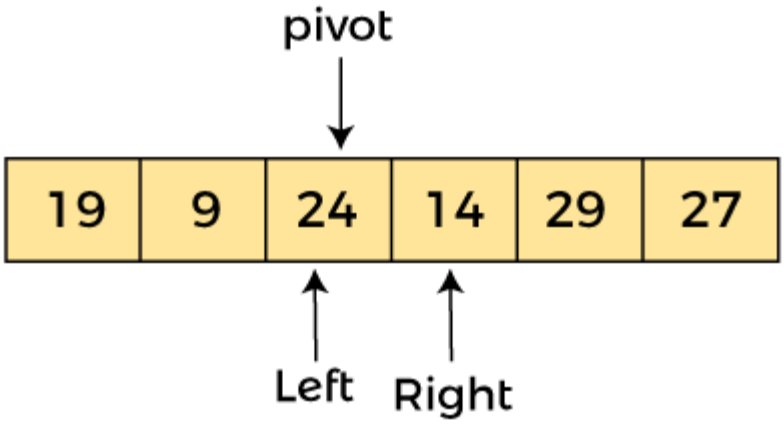
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



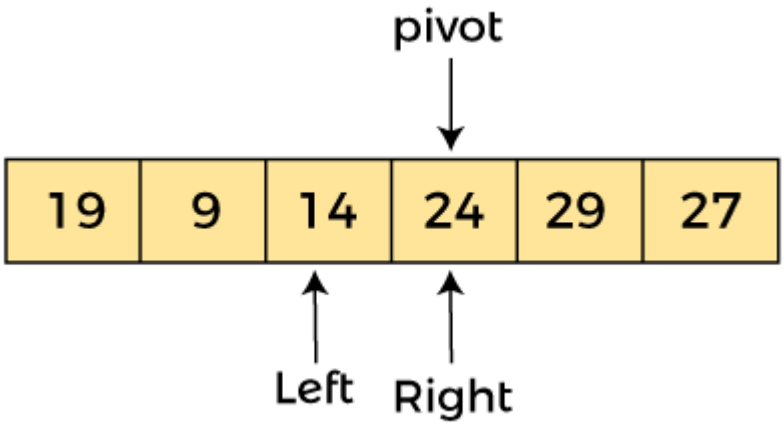
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



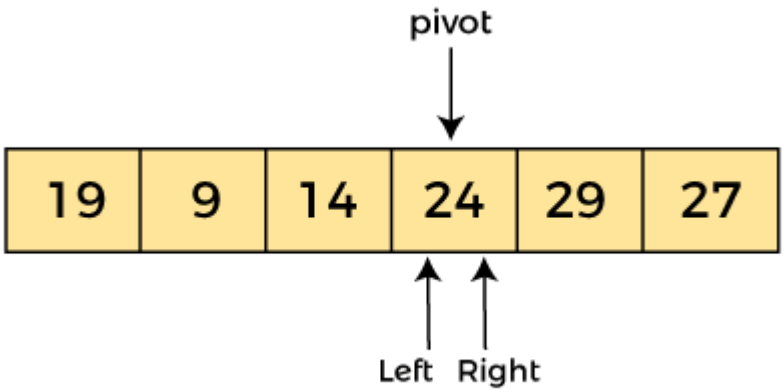
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



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. -



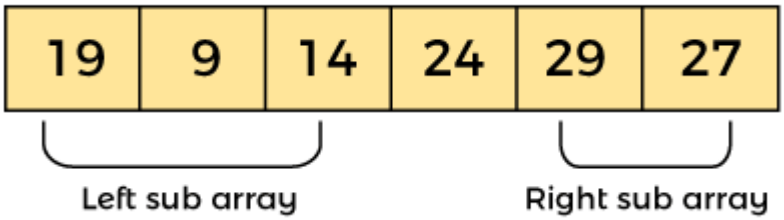
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



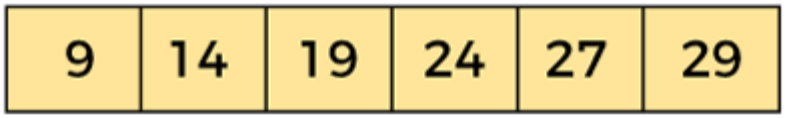
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

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 -



Quicksort complexity

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

1. Time Complexity

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

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort 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 quicksort is **O(n*logn)**.

Compiled by: Er. Sandesh S Poudel

- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$**.

Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

2. Space Complexity

Space Complexity	**O(n*logn)**
Stable	NO

- The space complexity of quicksort is O(n*logn).

Implementation of quicksort

Now, let's see the programs of quicksort in different programming languages.

Program: Write a program to implement quicksort in C language.

```
#include <stdio.h>
/* function that consider last element as pivot, place the pivot at its exact position, and place
smaller elements to left of pivot and greater elements to right of pivot.  */
int partition (int a[], int start, int end)
{
   int pivot = a[end]; // pivot element
   int i = (start - 1);
   for (int j = start; j <= end - 1; j++)
   {
      // If current element is smaller than the pivot
      if (a[j] < pivot)
      {
         i++; // increment index of smaller element
```

```
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    int t = a[i+1];
    a[i+1] = a[end];
    a[end] = t;
    return (i + 1);
}
/* function to implement quick sort */
void quick(int a[], int start, int end) /* a[] = array to be sorted, start = Starting index, end = Ending index
 */
{
    if (start < end)
    {
        int p = partition(a, start, end); //p is the partitioning index
        quick(a, start, p - 1);
        quick(a, p + 1, end);
    }
}
/* function to print an array */
void printArr(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}
int main()
{
    int a[] = { 24, 9, 29, 14, 19, 27 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
```

```
    printArr(a, n);
    quick(a, 0, n - 1);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
    return 0;
}
```

Example: a[]={5, 3, 2, 6, 4, 1, 3, 7}

```
5     3     2     6     4     1     3     7
x                                   y

5     3     2     6     4     1     3     7
                  x                 y     {swap x & y}

5     3     2     3     4     1     6     7
                              y     x     {swap y and pivot}

1     3     2     3     4     5     6     7
                              p
```

and continue this process for each sub-arrays and finally we get a sorted array.

Time Complexity:

Best Case:

Divides the array into two partitions of equal size, therefore

$T(n) = 2T(n/2) + O(n)$, Solving this recurrence we get, $T(n)=O(nlogn)$

Worst case:

when one partition contains the n-1 elements and another partition contains only one element.

Therefore its recurrence relation is:

$T(n) = T(n-1) + O(n)$, Solving this recurrence we get $T(n)=O(n^2)$

Average case:

Good and bad splits are randomly distributed across throughout the tree

$T1(n)= 2T'(n/2) + O(n)$ Balanced $T'(n)= T(n –1) + O(n)$

Unbalanced Solving:

$$B(n)= 2(B(n/2 –1) + \Theta(n/2)) + \Theta(n)$$

$= 2B(n/2 – 1) + \Theta(n)$

$= O(nlogn)$

$=>T(n)=O(nlogn)$

Merge Sort

To sort an array A[l . . r]:

- **Divide**
  - Divide the n-element sequence to be sorted into two sub-sequences of n/2 elements
- **Conquer**
  - Sort the sub-sequences recursively using merge sort. When the size of the sequences is 1, there is nothing more to do

•Combine

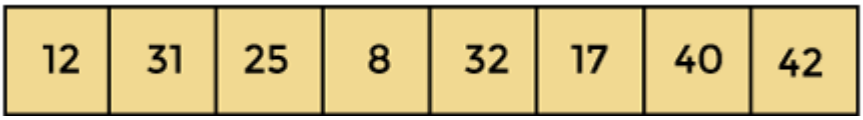
Merge the two sorted sub-sequences

Working of Merge sort Algorithm

Now, let's see the 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.

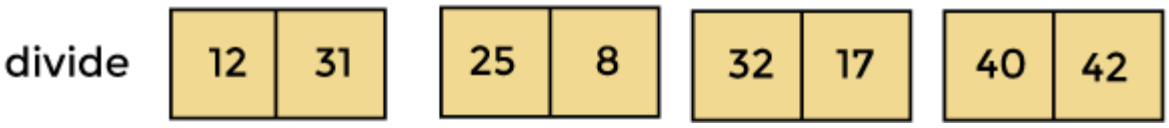
Example 1: Let the elements of array are -



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.

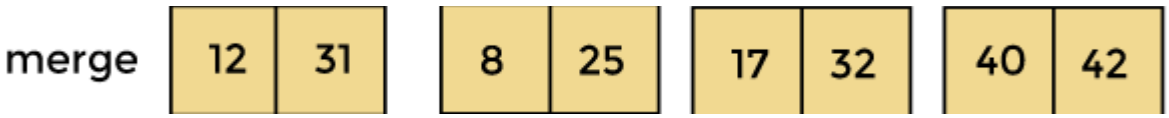


Now, again divide these arrays to get the atomic value that cannot be further divided.

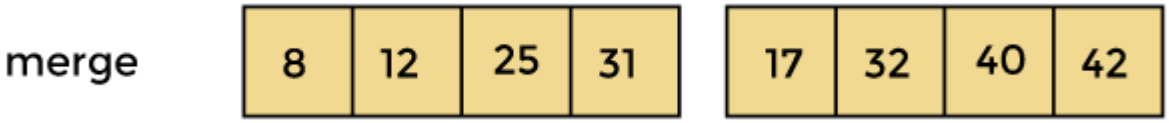
Now, combine them in the same manner they were broken.

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

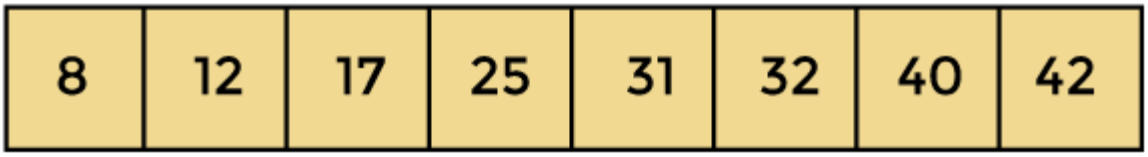
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



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.

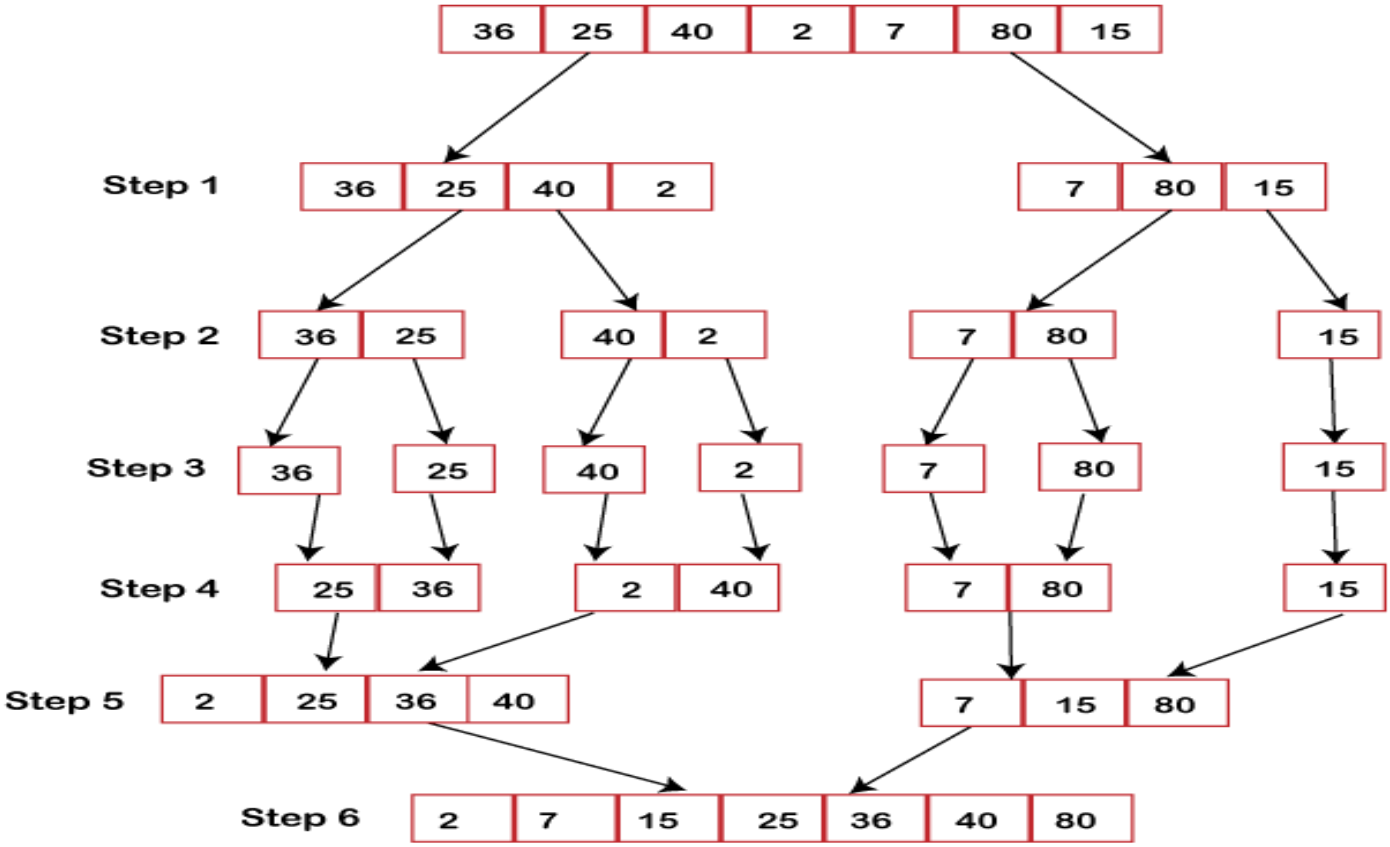


Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

Example 2:



Hence the array is sorted.

Example 3:

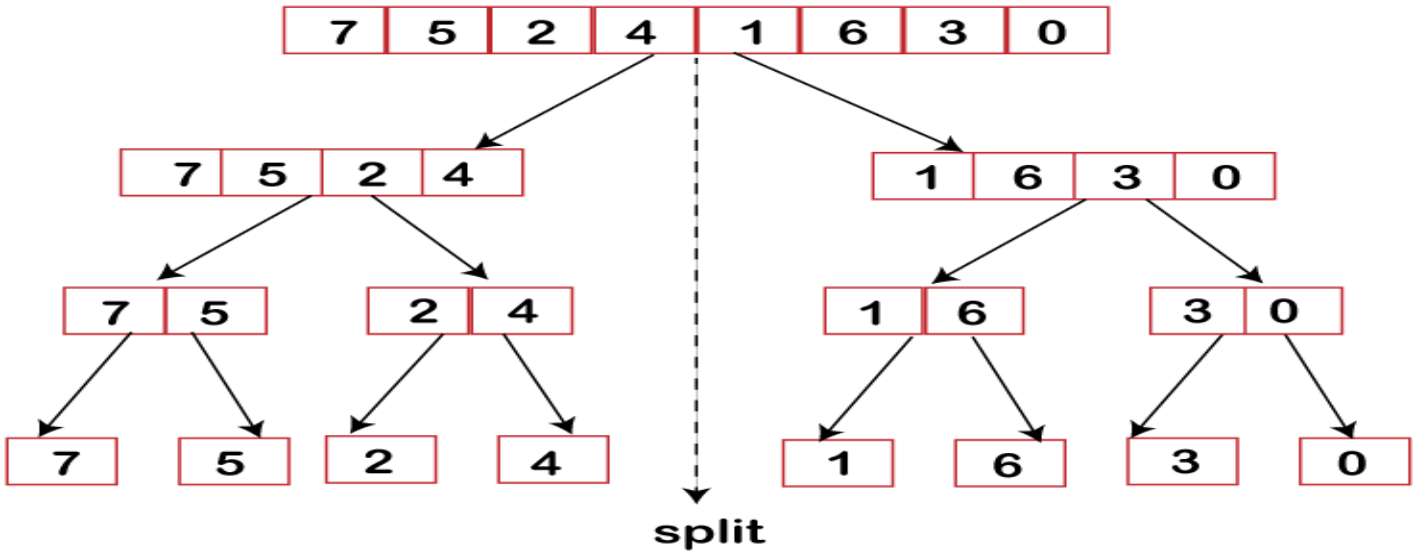


Figure 1: Merge Sort Divide Phase

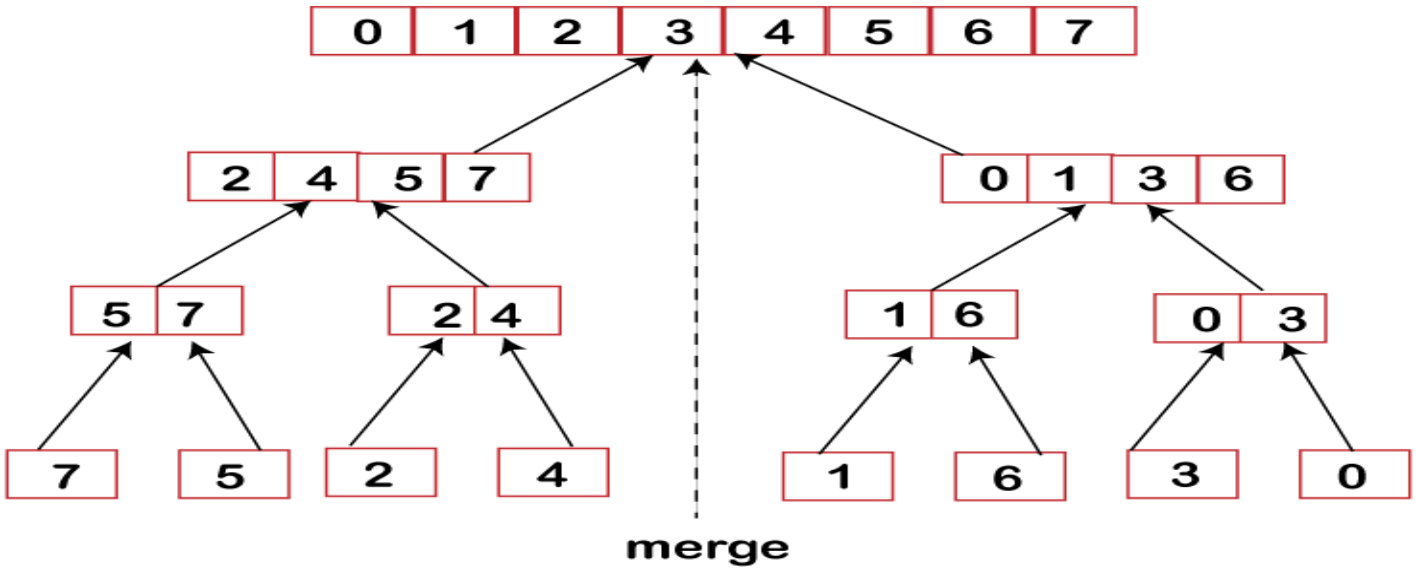
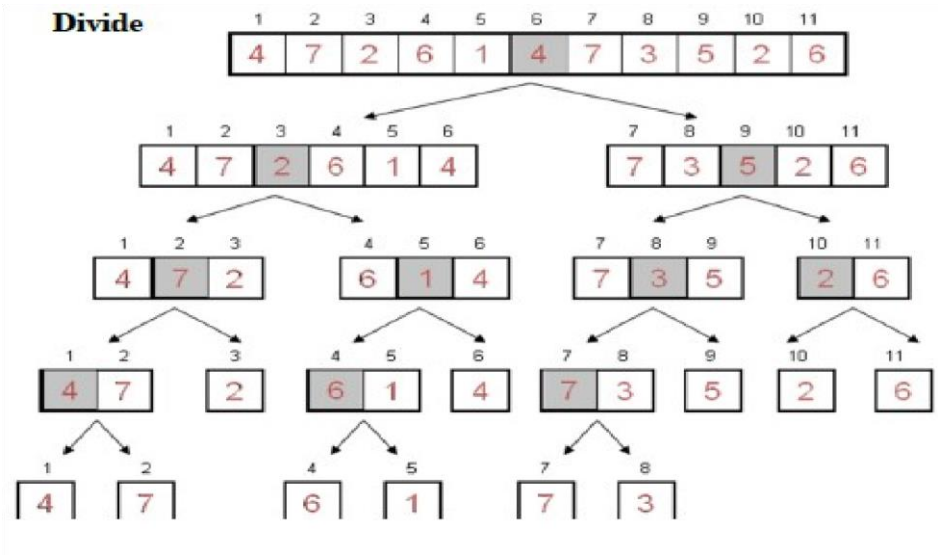
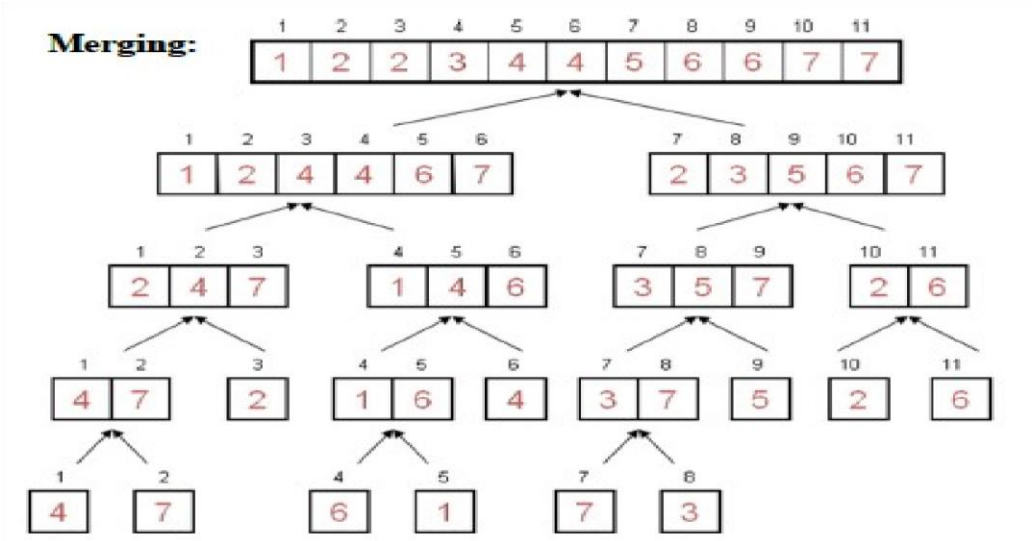


Figure 2: Merge Sort Combine Phase

Example 2: a[]={4, 7, 2, 6, 1, 4, 7, 3, 5, 2, 6}





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.

Implementation of merge sort

Now, let's see the programs of merge sort in different programming languages.

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

```
#include <stdio.h>
/* Function to merge the subarrays of a[] */
void merge(int a[], int beg, int mid, int end)
{
   int i, j, k;
   int n1 = mid - beg + 1;
```

```
    int n2 = end - mid;
    int LeftArray[n1], RightArray[n2]; //temporary arrays
    /* copy data to temp arrays */
    for (int i = 0; i < n1; i++)
    LeftArray[i] = a[beg + i];
    for (int j = 0; j < n2; j++)
    RightArray[j] = a[mid + 1 + j];
    i = 0; /* initial index of first sub-array */
    j = 0; /* initial index of second sub-array */
    k = beg;  /* initial index of merged sub-array */
    while (i < n1 && j < n2)
    {
        if(LeftArray[i] <= RightArray[j])
        {
            a[k] = LeftArray[i];
            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }
    while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}
void mergeSort(int a[], int beg, int end)
{
    if (beg < end)
    {
        int mid = (beg + end) / 2;
        mergeSort(a, beg, mid);
        mergeSort(a, mid + 1, end);
        merge(a, beg, mid, end);
    }
}
void printArray(int a[], int n)
{
    int i;
```

```
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}
int main()
{
    int a[] = { 12, 31, 25, 8, 32, 17, 40, 42 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArray(a, n);
    mergeSort(a, 0, n - 1);
    printf("After sorting array elements are - \n");
    printArray(a, n);
    return 0;
}
```

Algorithm:

```
MergeSort(A, l, r)
{
        If (l < r)
        {                                       //Check for base case
           m = ⌊(l + r)/2⌋                      //Divide
              MergeSort(A, l, m)                //Conquer
              MergeSort(A, m + 1, r)            //Conquer
              Merge(A, l, m+1, r)               //Combine
        }
}
```

```
Merge(A,B,l,m,r)
{
        x=l, y=m;
        k=l;
        while(x<m && y<r)
        {
                if(A[x] < A[y])
                {
                        B[k]= A[x];
                        k++; x++;
                }
                else
                {
                        B[k] = A[y];
                        k++; y++;
                }

        }
        while(x<m)
        {
                A[k] = A[x];
                k++; x++;
        }
        while(y<r)
        {
                A[k] = A[y];
                k++; y++;
        }
        for(i=l;i<= r; i++)
        {
                A[i] = B[i]
        }
    }
```

Radix Sort Algorithm

In this article, we will discuss the Radix sort Algorithm. 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.

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

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

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

The steps used in the sorting of radix sort are listed as follows -

- First, we have to find the largest element (suppose **max**) from the given array. Suppose **'x'** be the number of digits in **max**. The **'x'** is calculated because we need to go through the significant places of all elements.
- After that, go through one by one each significant place. Here, we have to use any stable sorting algorithm to sort the digits of each significant place.

Compiled by: Er. Sandesh S Poudel

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

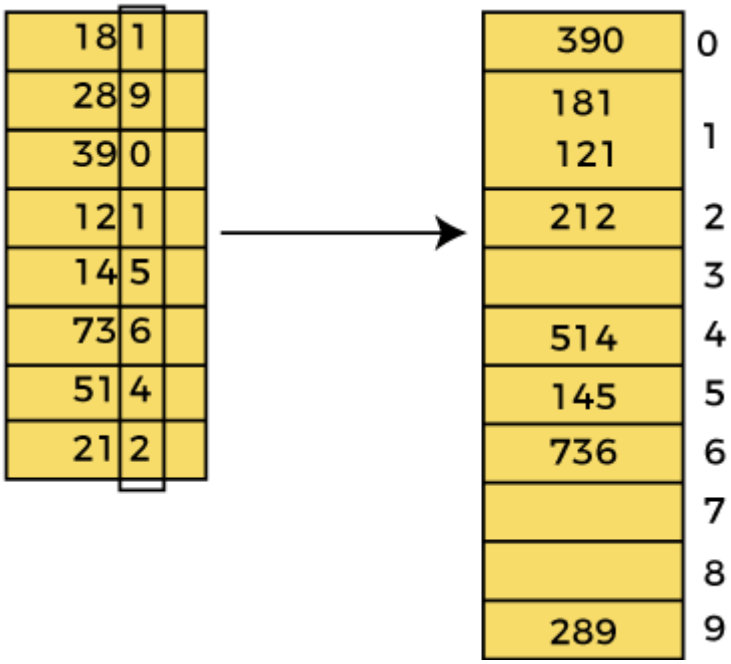
181	289	390	121	145	736	514	212

In the given array, the largest element is **736** that have **3** digits in it. So, the loop will run up to three times (i.e., to the **hundreds place**). That means three passes are required to sort the array.

Now, first sort the elements on the basis of unit place digits (i.e., **x = 0**). Here, we are using the counting sort algorithm to sort the elements.

Pass 1:

In the first pass, the list is sorted on the basis of the digits at 0's place.

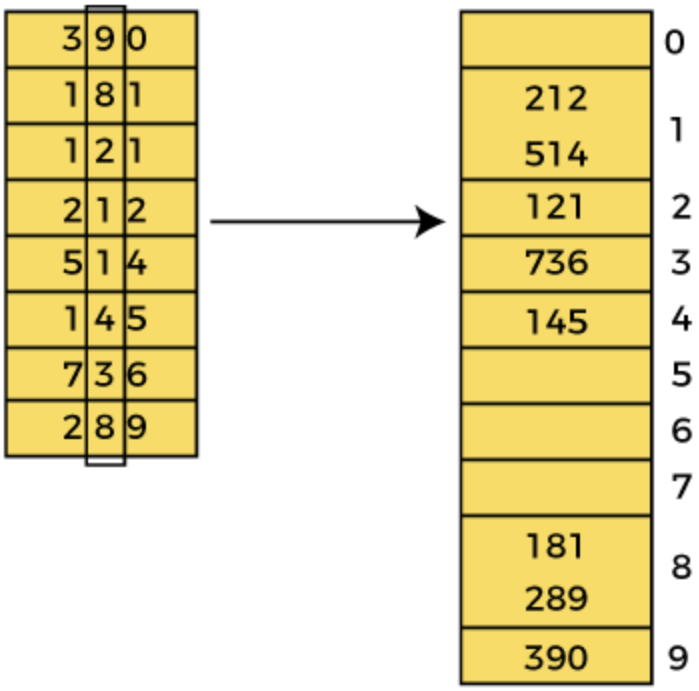


After the first pass, the array elements are -

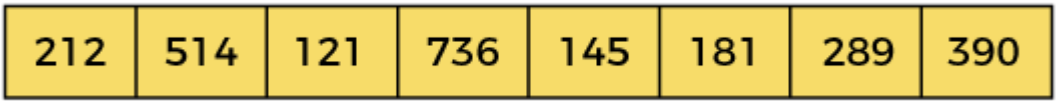
390	181	121	212	514	145	736	289

Pass 2:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10^{th} place).

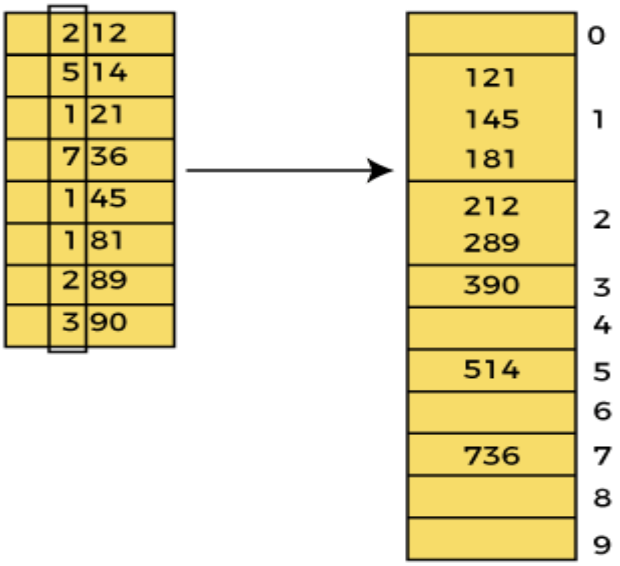


After the second pass, the array elements are -

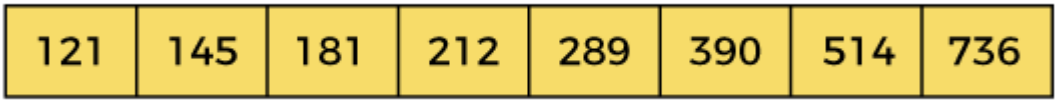


Pass 3:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 100th place).



After the third pass, the array elements are -



Compiled by: Er. Sandesh S Poudel

Now, the array is sorted in ascending order.

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)

where n is the length of the array, k is the maximum number of digits.

- **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

Now, let's see the programs of Radix sort in different programming languages.

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

```
#include <stdio.h>
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] - 1] = a[i];
    count[(a[i] / place) % 10]--;
  }
  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) {
    printf("%d  ", a[i]);
  }
  printf("\n");
}
int main() {
  int a[] = {181, 289, 390, 121, 145, 736, 514, 888, 122};
  int n = sizeof(a) / sizeof(a[0]);
  printf("Before sorting array elements are - \n");
  printArray(a,n);
  radixsort(a, n);
  printf("After applying Radix sort, the array elements are - \n");
  printArray(a, n);
}
```

Shell Sort Algorithm

In this article, we will discuss the shell sort algorithm. 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 -

hh = h * 3 + 1 (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 -

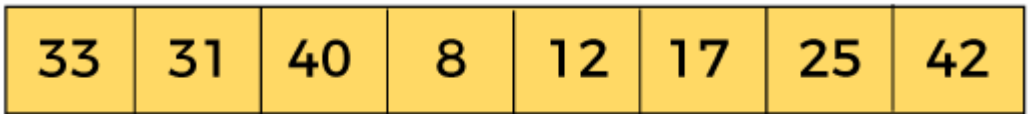
```
ShellSort(a, n) // 'a' is the given array, 'n' is the size of array
for (interval = n/2; interval > 0; interval /= 2)
for ( i = interval; i < n; i += 1)
temp = a[i];
for (j = i; j >= interval && a[j - interval] > temp; j -= interval)
a[j] = a[j - interval];
a[j] = temp;
End ShellSort
```

Working of Shell sort Algorithm

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

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

Let the elements of array are -

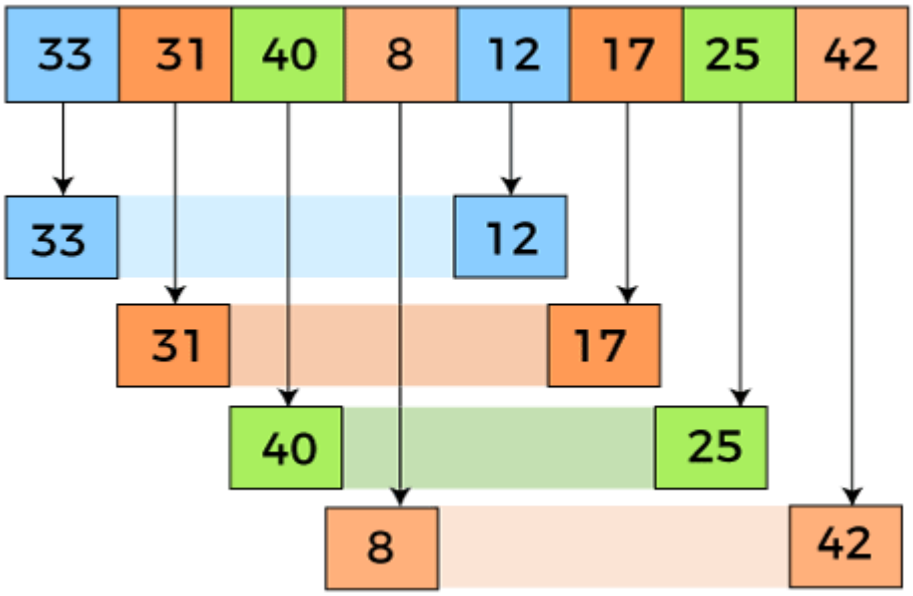


We will use the original sequence of shell sort, i.e., N/2, N/4,....,1 as the intervals.

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

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

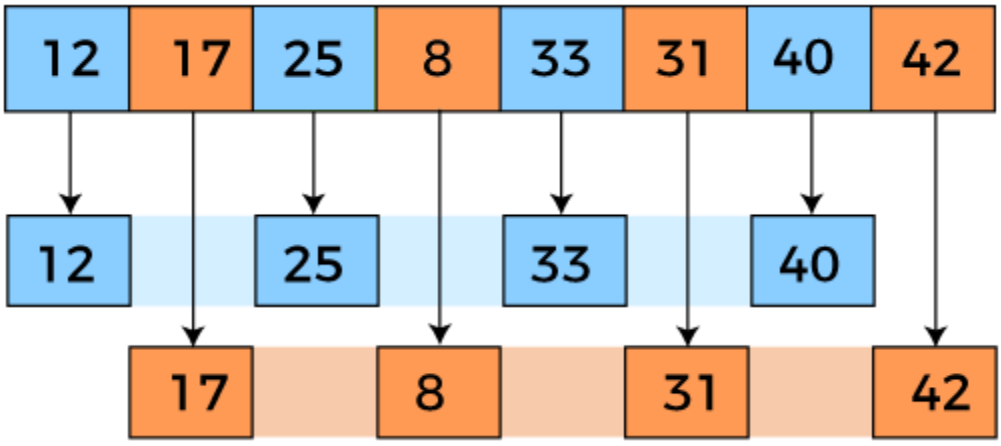
At the interval of 4, the sublists are {33, 12}, {31, 17}, {40, 25}, {8, 42}.



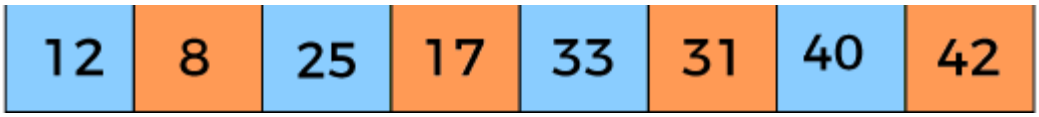
Now, we have to compare the values in every sub-list. After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows -

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

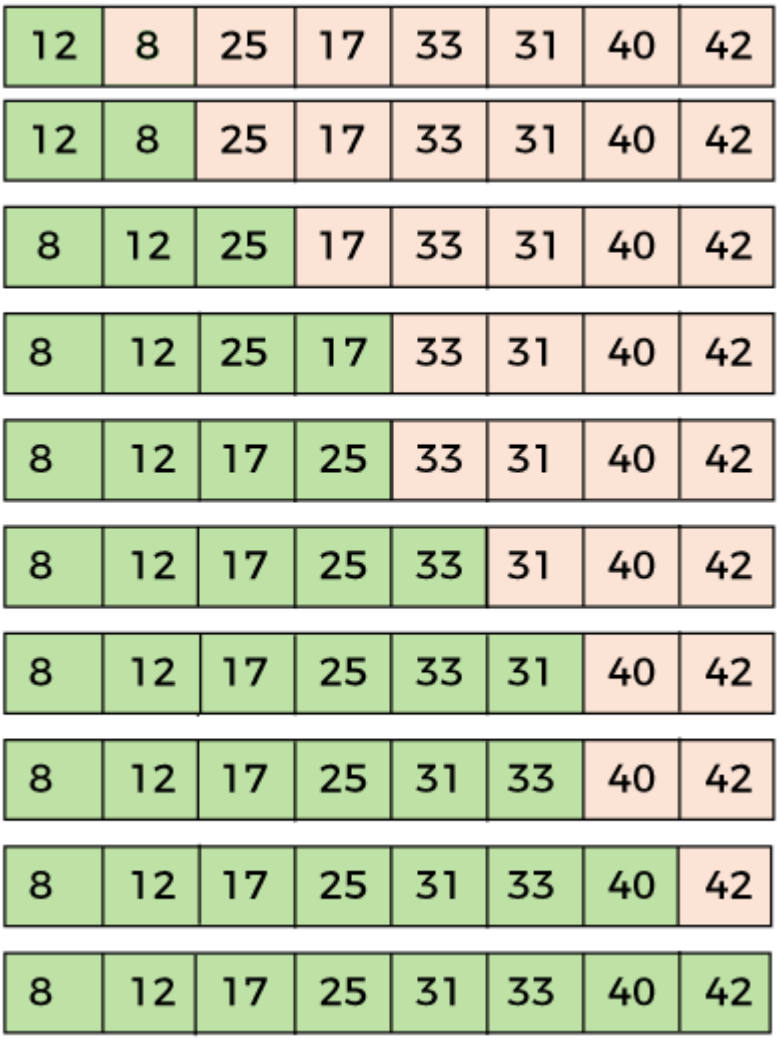
Now, we are taking the interval of **2** to sort the rest of the array. With an interval of 2, two sublists will be generated - {12, 25, 33, 40}, and {17, 8, 31, 42}.



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	$O(n^2)$

Compiled by: Er. Sandesh S Poudel

- **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

Now, let's see the programs of Shell sort in different programming languages.

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

```
#include <stdio.h>
/* function to implement shellSort */
int shell(int a[], int n)
{
   /* Rearrange the array elements at n/2, n/4, ..., 1 intervals */
   for (int interval = n/2; interval > 0; interval /= 2)
   {
      for (int i = interval; i < n; i += 1)
      {
         /* store a[i] to the variable temp and make the ith position empty */
         int temp = a[i];
         int j;
         for (j = i; j >= interval && a[j - interval] > temp; j -= interval)
            a[j] = a[j - interval];
```

```
            // put temp (the original a[i]) in its correct position
            a[j] = temp;
        }
    }
    return 0;
}
void printArr(int a[], int n) /* function to print the array elements */
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}
int main()
{
    int a[] = { 33, 31, 40, 8, 12, 17, 25, 42 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    shell(a, n);
    printf("\nAfter applying shell sort, the array elements are - \n");
    printArr(a, n);
    return 0;
}
```

Heap Sort Algorithm

In this article, we will discuss the Heapsort Algorithm. 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.

Before knowing more about the heap sort, let's first see a brief description of **Heap.**

What is a 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.

What is 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.

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

Algorithm

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

BuildMaxHeap(arr)

```
BuildMaxHeap(arr)
   heap_size(arr) = length(arr)
   for i = length(arr)/2 to 1
MaxHeapify(arr,i)
```

End

MaxHeapify(arr,i)

```
MaxHeapify(arr,i)
L = left(i)
R = right(i)
if L ? heap_size[arr] and arr[L] > arr[i]
largest = L
else
largest = i
if R ? heap_size[arr] and arr[R] > arr[largest]
largest = R
if largest != i
swap arr[i] with arr[largest]
MaxHeapify(arr,largest)
End
```

Working of Heap sort Algorithm

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

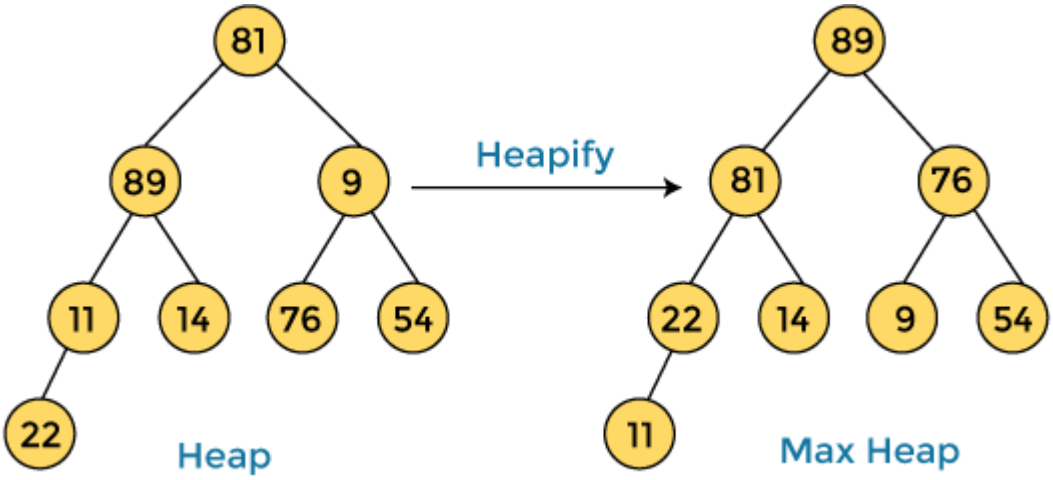
In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

- 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

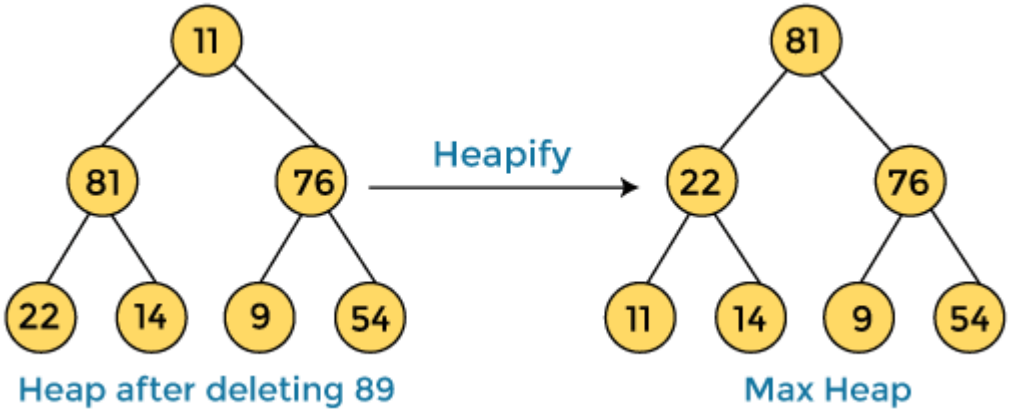
First, we have to construct a heap from the given array and convert it into max heap.



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.

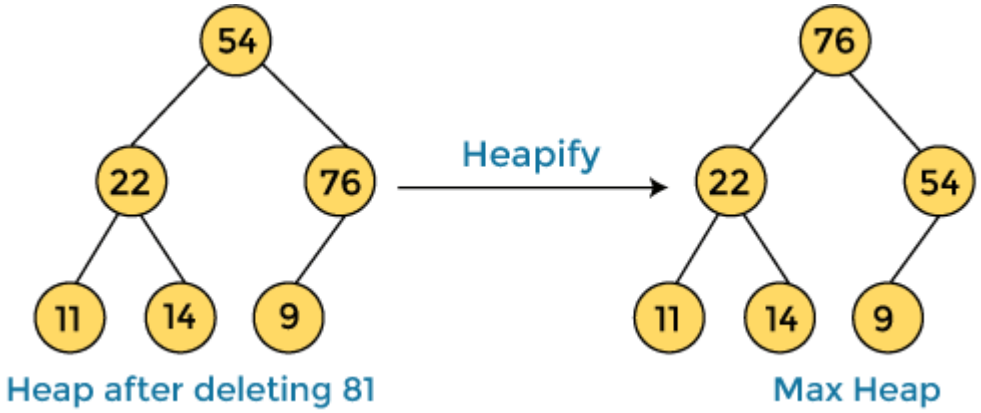


After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -

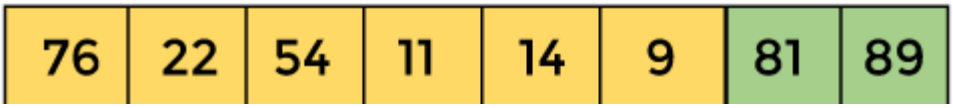
81	22	76	11	14	9	54	89

Compiled by: Er. Sandesh S Poudel

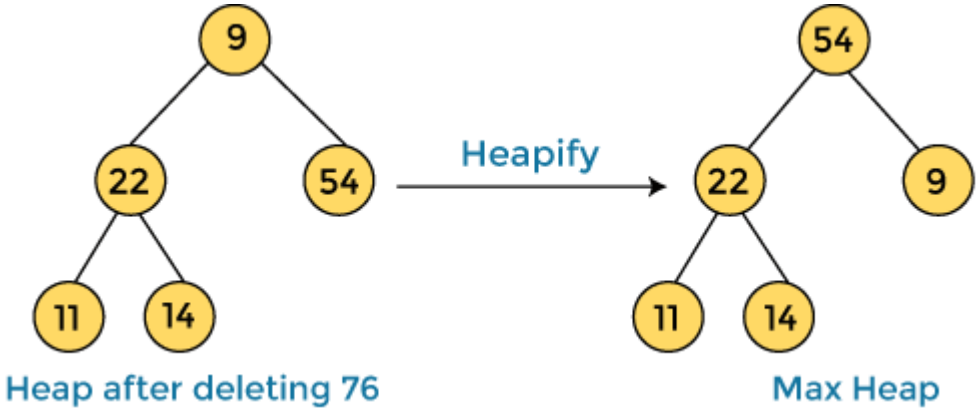
In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



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

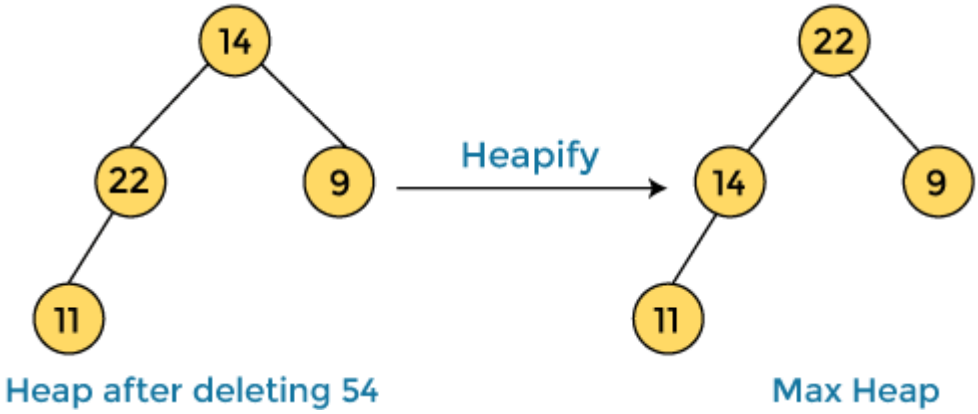


In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

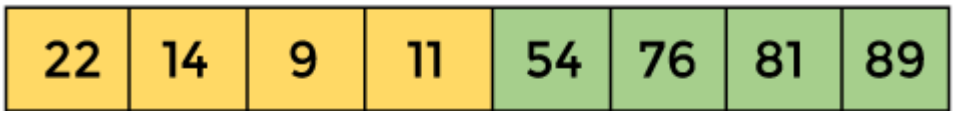


After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

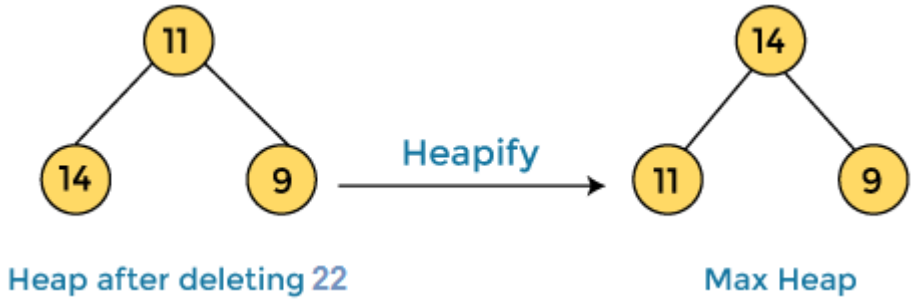
In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



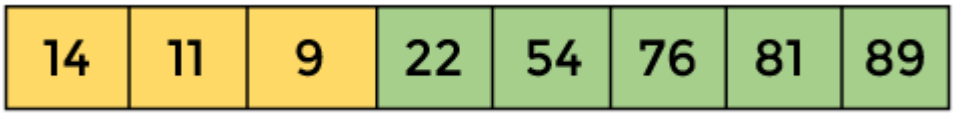
After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -



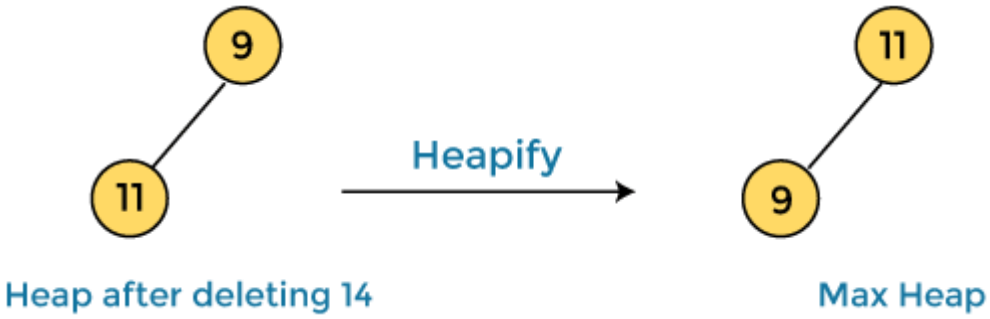
In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



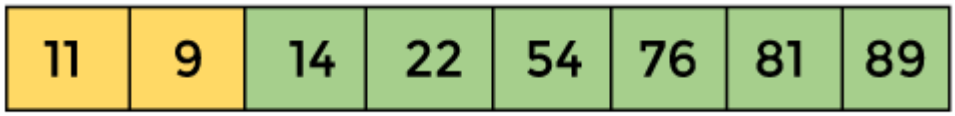
After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -



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.

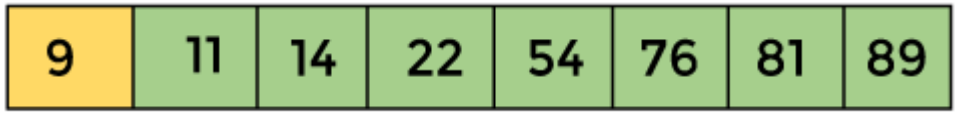


After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -



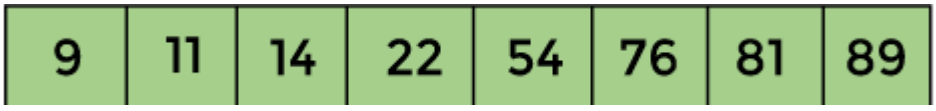
In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

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.

After completion of sorting, the array elements are -



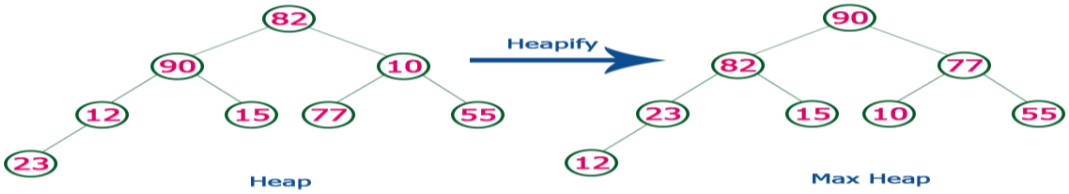
Now, the array is completely sorted.

Example 2:

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

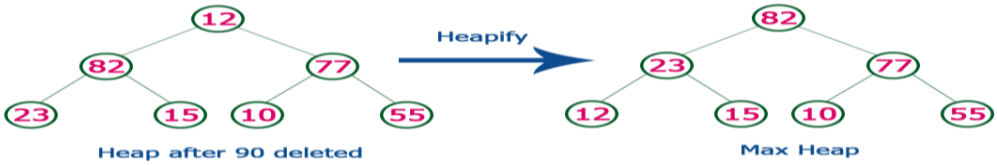
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, 90

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, 82, 90

Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, 77, 82, 90

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 55 with 15.

12, 15, 10, 23, 55, 77, 82, 90

Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 23 with 12.

12, 15, 10, 23, 55, 77, 82, 90

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Compiled by: Er. Sandesh S Poudel

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	N0

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

Implementation of Heapsort

Now, let's see the programs of Heap sort in different programming languages.

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

```
#include <stdio.h>
  /* function to heapify a subtree. Here 'i' is the index of root node in array a[], and 'n' is the size of heap. */
  void heapify(int a[], int n, int i)
        {
           int largest = i; // Initialize largest as root
           int left = 2 * i + 1; // left child
```

```
    int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && a[left] > a[largest])
        largest = left;
    // If right child is larger than root
    if (right < n && a[right] > a[largest])
        largest = right;
    // If root is not largest
    if (largest != i) {
        // swap a[i] with a[largest]
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;
        heapify(a, n, largest);
}
}
/*Function to implement the heap sort*/
void heapSort(int a[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(a, n, i);
    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        /* Move current root element to end*/
        // swap a[0] with a[i]
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;

        heapify(a, i, 0);
    }
}
/* function to print the array elements */
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
    {
        printf("%d", arr[i]);
        printf(" ");
    }

}
int main()
{
    int a[] = {48, 10, 23, 43, 28, 26, 1};
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    heapSort(a, n);
    printf("\nAfter sorting array elements are - \n");
```

```
    printArr(a, n);
    return 0;
}
```

Sorting Comparison:

Sort	Worst Case	Average Case	Best Case	Comments
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	(*Unstable)
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	
Merge Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	Requires Memory
Heap Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	*Large constants
Quick Sort	$\Theta(n^2)$	$\Theta(n\log n)$	$\Theta(n\log n)$	*Small constants