Design and Analysis of Algorithm

UNIT – 1 : Analysis of Algorithm

What is Algorithm?

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.

Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

Search − Algorithm to search an item in a data structure.

Sort − Algorithm to sort items in a certain order.

Insert − Algorithm to insert item in a data structure.

Update − Algorithm to update an existing item in a data structure.

Delete − Algorithm to delete an existing item from a data structure.

How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example:

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

Step 1 – START

Step 2 − declare three integers a, b & c

Step 3 − define values of a & b

Step 4 − add values of a & b

Step 5 − store output of step 4 to c

Step 6 − print c

Step 7 – STOP

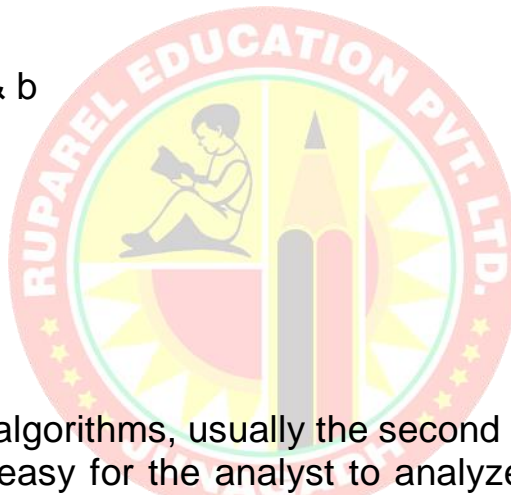Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as −

Step 1 − START ADD

Step 2 − get values of a & b

Step 3 − c ← a + b

Step 4 − display c

Step 5 – STOP

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing step numbers, is optional.

Use of the Algorithms:

Algorithms play an important role in various fields and have many applications. Some of the key areas where algorithms are used include:

Computer Science:

Algorithms form the basis of computer programming and are used to solve problems ranging from simple sorting and searching to complex tasks such as artificial intelligence and machine learning.

Mathematics:

Algorithms are used to solve mathematical problems, such as finding the optimal solution to a system of equations or finding the shortest path in a graph.

Operations Research:

Algorithms are used to optimize and make decisions in fields such as transportation, Information Technology etc.

Artificial Intelligence:

Algorithms are the foundation of artificial intelligence and machine learning, and are used to develop intelligent systems that can perform tasks such as image recognition, natural language processing, and decision-making.

Data Science:

Algorithms are used to analyze, process, and extract insights from large amounts of data in fields such as marketing, finance, and healthcare.

What is the need for algorithms?

- Algorithms are necessary for solving complex problems effectively.

- They help to automate processes and make them more reliable, faster, and easier to perform.

- Algorithms also enable computers to perform tasks that would be difficult or impossible for humans to do manually.

- They are used in various fields such as mathematics, computer science, engineering, finance, and many others to optimize processes, analyze data, make predictions, and provide solutions to problems.

Types of Algorithms:

There are several types of algorithms available. Some important algorithms are:

Brute Force Algorithm:

It is the simplest approach to a problem. A brute force algorithm is the first approach that comes to finding when we see a problem.

Recursive Algorithm:

A recursive algorithm is based on <u>recursion</u>. In this case, a problem is broken into several sub-parts and called the same function again and again.

Backtracking Algorithm:

The backtracking algorithm builds the solution by searching among all possible solutions. Using this algorithm, we keep on building the solution following criteria. Whenever a solution fails we trace back to the failure point build on the next solution and continue this process till we find the solution or all possible solutions are looked after.

Searching Algorithm:

Searching algorithms are the ones that are used for searching elements or groups of elements from a particular data structure. They can be of different types based on their approach or the data structure in which the element should be found.

Sorting Algorithm:

Sorting is arranging a group of data in a particular manner according to the requirement. The algorithms which help in performing this function are called sorting algorithms. Generally sorting algorithms are used to sort groups of data in an increasing or decreasing manner.

Hashing Algorithm:

Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.

Divide and Conquer Algorithm:

This algorithm breaks a problem into sub-problems, solves a single sub-problem, and merges the solutions to get the final solution. It consists of the following three steps: Divide, Solve, Combine

Greedy Algorithm:

In this type of algorithm, the solution is built part by part. The solution for the next part is built based on the immediate benefit of the next part. The one solution that gives the most benefit will be chosen as the solution for the next part.

Advantages of Algorithms:

- It is easy to understand.

- An algorithm is a step-wise representation of a solution to a given problem.

- In an Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.

- Understanding complex logic through algorithms can be very difficult.

- Branching and Looping statements are difficult to show in Algorithms(imp).

Efficiency of an algorithm:

Computer resources are limited that should be utilized efficiently. The efficiency of an algorithm is defined as the number of computational resources used by the algorithm.

An algorithm must be analysed to determine its resource usage. The efficiency of an algorithm can be measured based on the usage of different resources.

For maximum efficiency of algorithm, we wish to minimize resource usage. The important resources such as time and space complexity cannot be compared directly, so time and space complexity could be considered for an algorithmic efficiency.

Average, Best and Worst case analysis:

When analysing the performance of algorithms, we often consider three different cases: average case, best case, and worst case. Each case provides insights into how an algorithm is expected to behave under different conditions:

Best Case Analysis:

The best case analysis considers the scenario in which the algorithm performs at its optimal level. It assumes that the input is structured or behaves in a way that is most favourable for the algorithm.

Best-case time complexity represents the minimum amount of time or resources required to solve a problem of a given size.

Best-case analysis is useful for understanding the lower bounds of an algorithm's performance and for identifying situations in which the algorithm excels.

In practice, best-case scenarios are often theoretical and rarely encountered in real-world situations.

Worst Case Analysis:

The worst case analysis considers the scenario in which the algorithm performs at its least favourable level. It assumes that the input is structured or behaves in a way that is most challenging for the algorithm.

Worst-case time complexity represents the maximum amount of time or resources required to solve a problem of a given size.

Worst-case analysis is essential because it guarantees that the algorithm will perform no worse than this level under any input conditions.

In practice, worst-case scenarios help in determining the upper bounds of an algorithm's performance, ensuring that it does not exceed these bounds regardless of input.

Average Case Analysis:

The average case analysis considers the expected performance of the algorithm when inputs are randomly distributed or follow a specific probability distribution.

Average-case time complexity represents the expected amount of time or resources required to solve a problem of a given size when considering all possible inputs.

Average-case analysis provides a more realistic assessment of an algorithm's performance under typical or random input conditions.

However, conducting average-case analysis can be challenging and may require knowledge of probability theory and statistics.

In summary, best-case analysis gives us insights into the lower bounds of an algorithm's performance, worst-case analysis provides upper bounds, and average-case analysis offers a more practical view of expected performance under typical input conditions.

Depending on the specific problem and the algorithm's characteristics, one or more of these analyses may be used to evaluate and compare different algorithms.

Example of Average, Best and worst case analysis:

Problem: Given an unsorted array of integers, find the index of a specific target element if it exists in the array.

Algorithm: Linear Search

Here's how we can analyze the time complexity for each of the three cases:

Best Case (Target element is the first element of the array):

Best-case scenario occurs when the element we are searching for is found at the very beginning of the array.

In this case, the algorithm will find the target in the first comparison, and it will take only one comparison.

Worst Case (Target element is at the end of the array):

Worst-case scenario occurs when the element we are searching for is at the very end of the array.

In this case, the algorithm needs to compare the target element with all elements in the array until reaching the last element.

Average Case (Target element is not in the array):

In average-case analysis, we assume that the target element is equally likely to be at any position not in the array.

On average, the algorithm will need to examine all of the elements in the array before finding the target element.

Asymptotic Notations

Asymptotic notations are a set of mathematical notations used in computer science and mathematics to describe the behavior of functions as their inputs become very large (i.e., they approach infinity).

These notations help analyze the efficiency and performance of algorithms and describe how the runtime or resource usage of an algorithm scales with the size of the input.

The three most commonly used asymptotic notations are:

Big O Notation (O):

Big O notation, represented as O(n), describes the upper bound or worst-case scenario of the growth rate of an algorithm's runtime in relation to the input size (n).

It provides an upper limit on the running time, indicating that the algorithm's performance will not exceed a certain order of magnitude.

For example, if an algorithm has a time complexity of O(n), it means the runtime grows linearly with the input size.

Omega Notation (Ω):

Omega notation, represented as Ω(n), describes the lower bound or best-case scenario of the growth rate of an algorithm's runtime in relation to the input size (n).

It provides a lower limit on the running time, indicating that the algorithm's performance will not be faster than a certain order of magnitude.

For example, if an algorithm has a time complexity of Ω(n), it means the runtime grows at least linearly with the input size.

Theta Notation (Θ):

Theta notation, represented as Θ(n), describes both the upper and lower bounds of an algorithm's runtime.

It indicates that the algorithm's performance grows at the same rate as the function f(n) within constant factors.

For example, if an algorithm has a time complexity of Θ(n), it means the runtime grows linearly with the input size, and it's neither faster nor slower.

These asymptotic notations are valuable for comparing and analysing the efficiency of algorithms without getting into the specifics of hardware or constant factors.

They help algorithm designers make informed decisions about which algorithm to use for a particular problem based on its expected performance as the input size grows.

Analysing control statement

Control Structures are just a way to specify flow of control in programs.

Any algorithm or program can be more clear and understood if they use self-contained modules called as logic or control structures.

It basically analyses and chooses in which direction a program flows based on certain parameters or conditions.

There are three basic types of logic, or flow of control, known as:

Sequence logic, or sequential flow
Selection logic, or conditional flow
Iteration logic, or repetitive flow

Let us see them in detail:

Sequential Logic (Sequential Flow):

Sequential logic as the name suggests follows a serial or sequential flow in which the flow depends on the series of instructions given to the computer.

Unless new instructions are given, the modules are executed in the obvious sequence. The sequences may be given, by means of numbered steps explicitly.
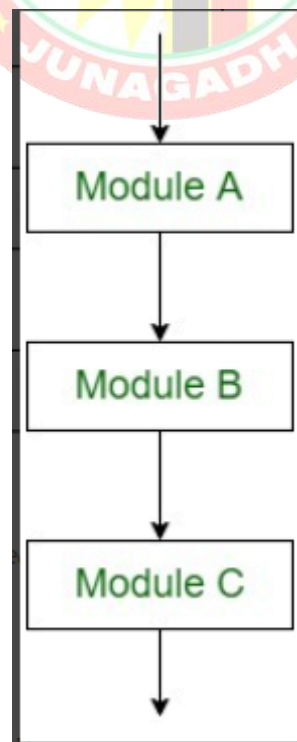
Selection Logic (Conditional Flow):

Selection Logic simply involves a number of conditions or parameters which decides one out of several written modules.

The structures which use these type of logic are known as Conditional Structures. These structures can be of three types:

Single Alternative:

This structure has the form:

If (condition) then:

[Module A]

[End of If structure]
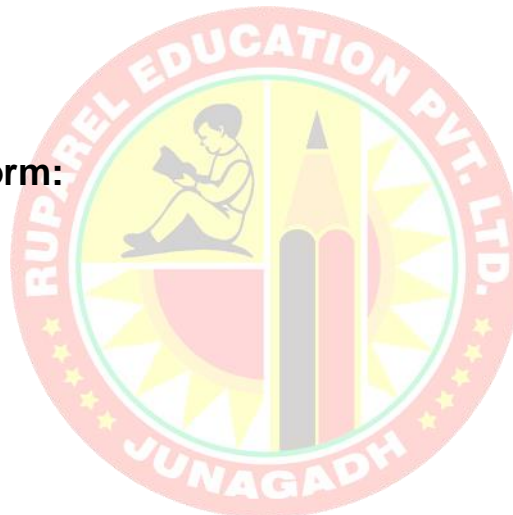
Double Alternative:

This structure has the form:

If (Condition), then:

[Module A]

Else:

[Module B]

[End if structure]

Multiple Alternatives:

This structure has the form:

If (condition A), then:

[Module A]

Else if (condition B), then:
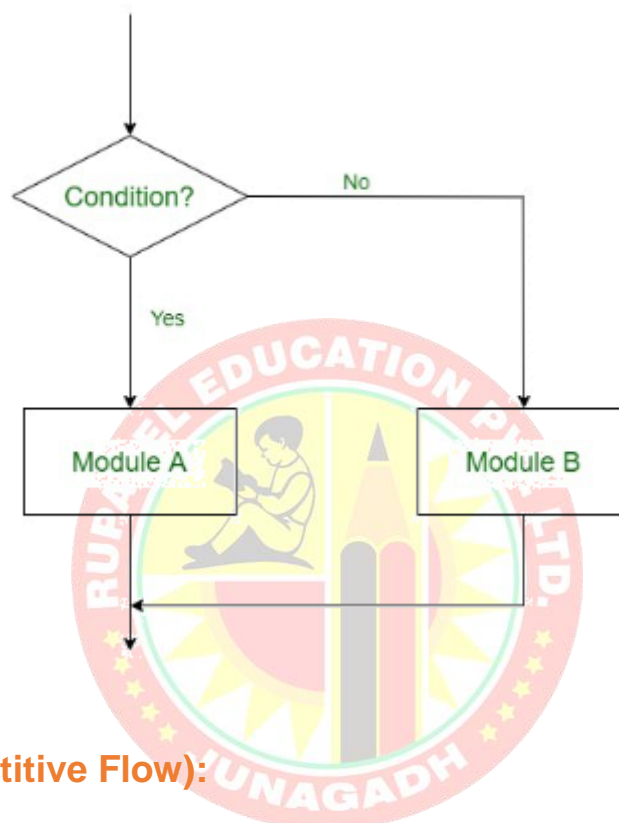
[Module B]
..
..

Else if (condition N), then:

    [Module N]

[End If structure]

In this way, the flow of the program depends on the set of conditions that are written.

This can be more understood by the following flow charts:

Iteration Logic (Repetitive Flow):

The Iteration logic employs a loop which involves a repeat statement followed by a module known as the body of a loop.

The two types of these structures are:

Repeat-For Structure:

This structure has the form:

    Repeat for i = A to N by I:

        [Module]

    [End of loop]

Here, A is the initial value, N is the end value and I is the increment. The loop ends when A>B.
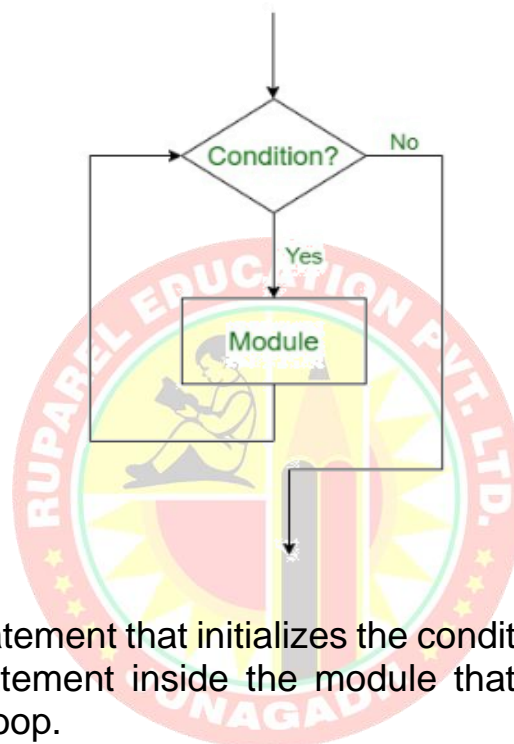
Repeat-While Structure:

It also uses a condition to control the loop.

This structure has the form:

Repeat While condition:

[Module]

[End of Loop]

In this, there requires a statement that initializes the condition controlling the loop, and there must also be a statement inside the module that will change this condition leading to the end of the loop.

Sorting Algorithms and analysis:

Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order.

For example, consider an array A = {A1, A2, A3, A4, ??, An}, the array is called to be in ascending order if element of A is arranged like A1 > A2 > A3 > A4 > A5 > ? > An.

Consider an array;

int A [10] = {5, 4, 10, 2, 30, 45, 34, 14, 18, 9}

The Array sorted in ascending order will be given as:

A [ ] = {2, 4, 5, 9, 10, 14, 18, 30, 34, 45}

There are many techniques by using which, sorting can be performed. we will discuss each method in detail.

Sorting Algorithms

Sorting algorithms are described in the following table along with the description.

Sorting Algorithms	Description
Bubble Sort	It is the simplest sort method which performs sorting by repeatedly moving the largest element to the highest index of the array. It comprises of comparing each element to its adjacent element and replace them accordingly.
Selection Sort	Selection sort finds the smallest element in the array and place it on the first place on the list, then it finds the second smallest element in the array and place it on the second place. This process continues until all the elements are moved to their correct ordering. It carries running time O(n2) which is worse than insertion sort.
Insertion Sort	As the name suggests, insertion sort inserts each element of the array to its proper place. It is a very simple sort method which is used to arrange the deck of cards while playing bridge.
Shell Sort	Shell sort is the generalization of insertion sort which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.
Heap Sort	In the heap sort, Min heap or max heap is maintained from the array elements depending upon the choice and the elements are sorted by deleting the root element of the heap.
Bucket Sort	Bucket sort is also known as bin sort. It works by distributing the element into the array also called buckets. In this sorting algorithms, Buckets are sorted individually by using different sorting algorithm.
Radix Sort	In Radix sort, the sorting is done as we do sort the names according to their alphabetical order. It is the linear sorting algorithm used for Integers.
Counting Sort	It is a sorting technique based on the keys i.e. objects are collected according to keys which are small integers. Counting sort calculates the number of occurrence of objects and stores its key values. New array is formed by adding previous key elements and assigning to objects.

Bubble Sort Algorithm:

The working procedure of bubble sort is simplest.

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water.

Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.

Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets.

The average and worst-case complexity of Bubble sort is **O(n^2),** where **n** is a number of items.

Bubble short is majorly used where –

- o complexity does not matter

- o simple and short code is preferred

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).**

Let the elements of array are –

| 13 | 32 | 26 | 35 | 10 |

First Pass:

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

| 13 | 32 | 26 | 35 | 10 |

Here, 32 is greater than 13 (32 > 13), so it is already sorted.

Now, compare 32 with 26.

| 13 | 32 | 26 | 35 | 10 |

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

| 13 | 26 | 32 | 35 | 10 |

Now, compare 32 and 35.

| 13 | 26 | 32 | 35 | 10 |

Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.
Now, the comparison will be in between 35 and 10.

| 13 | 26 | 32 | 35 | 10 |

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

| 13 | 26 | 32 | 10 | 35 |

Now, move to the second iteration.

Second Pass:

The same process will be followed for second iteration.

| 13 | 26 | 32 | 10 | 35 |

| 13 | 26 | 32 | 10 | 35 |

| 13 | 26 | 32 | 10 | 35 |

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

| 13 | 26 | 10 | 32 | 35 |

| 13 | 26 | 10 | 32 | 35 |

Now, move to the third iteration.

Third Pass:

The same process will be followed for third iteration.

| 13 | 26 | 10 | 32 | 35 |

| 13 | 26 | 10 | 32 | 35 |

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

| 13 | 10 | 26 | 32 | 35 |

| 13 | 10 | 26 | 32 | 35 |

| 13 | 10 | 26 | 32 | 35 |

Now, move to the fourth iteration.

Fourth pass:

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

| 10 | 13 | 26 | 32 | 35 |

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

Bubble Sort Complexity:

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

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(n²).**

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(n²).**

Optimized Bubble Sort Algorithm:

In the bubble sort algorithm, comparisons are made even when the array is already sorted. Because of that, the execution time increases.

To solve it, we can use an extra variable *swapped.* It is set to **true** if swapping requires; otherwise, it is set to **false.**

It will be helpful, as suppose after an iteration, if there is no swapping required, the value of variable **swapped** will be **false.** It means that the elements are already sorted, and no further iterations are required.

This method will reduce the execution time and also optimizes the bubble sort.

Program: Write a program to implement Bubble Sort in C language.

```c
#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 ("\n After sorting array elements are - \n");
    print (a, n);
}
```

Output:

```
Before sorting array elements are -
10 35 32 13 26
After sorting array elements are -
10 13 26 32 35
```

Selection Sort Algorithm

The working procedure of selection sort is also simple.

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part.

Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

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

The average and worst-case complexity of selection sort is **O(n²)**, where **n** is the number of items. Due to this, it is not suitable for large data sets.

Selection sort is generally used when –

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

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

Working of Selection Sort Algorithm:

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

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

Let the elements of array are –

| 12 | 29 | 25 | 8 | 32 | 17 | 40 |

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

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

| 12 | 29 | 25 | 8 | 32 | 17 | 40 |

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.
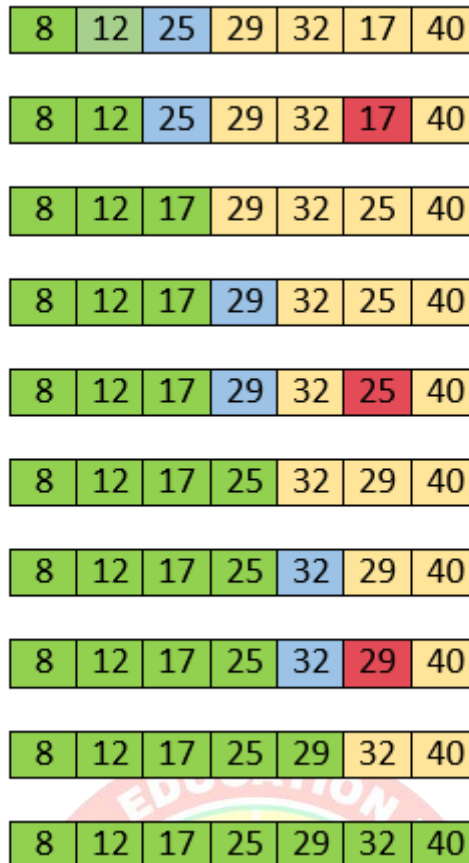
| 8 | 29 | 25 | 12 | 32 | 17 | 40 |

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

| 8 | 29 | 25 | 12 | 32 | 17 | 40 |

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.

| 8 | 12 | 25 | 29 | 32 | 17 | 40 |

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

| 8 | 12 | 25 | 29 | 32 | 17 | 40 |

| 8 | 12 | 25 | 29 | 32 | 17 | 40 |

| 8 | 12 | 17 | 29 | 32 | 25 | 40 |

| 8 | 12 | 17 | 29 | 32 | 25 | 40 |

| 8 | 12 | 17 | 29 | 32 | 25 | 40 |

| 8 | 12 | 17 | 25 | 32 | 29 | 40 |

| 8 | 12 | 17 | 25 | 32 | 29 | 40 |

| 8 | 12 | 17 | 25 | 32 | 29 | 40 |

| 8 | 12 | 17 | 25 | 29 | 32 | 40 |

| 8 | 12 | 17 | 25 | 29 | 32 | 40 |

Now, the array is completely sorted.

Selection Sort Complexity:

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(n²)**.

Average Case Complexity:
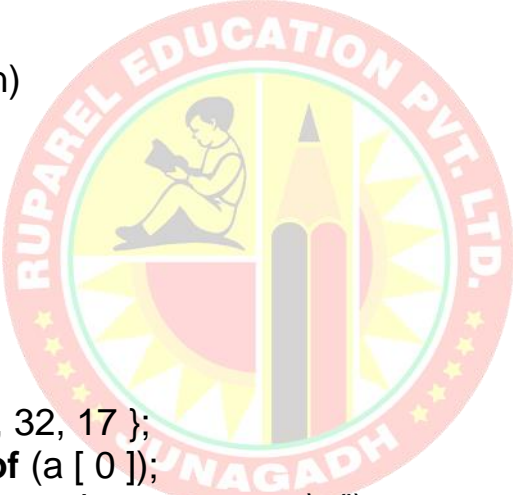
It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **O(n²)**.

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(n²)**.

Program: **Write a program to implement Selection Sort in C language.**

```c
#include <stdio.h>
void selection (int arr [ ], int n)
{
      int i, j, small;
      for (i = 0; i < n-1; i++)
      {
            small = i;
            for (j = i+1; j < n; j++)
            {
                  if (arr [ j ] < arr [ small ])
                  small = j;
            }
            int temp = arr [ small ];
            arr [ small ] = arr [ i ];
            arr [ i ] = temp;
      }
}
void printArr (int a [ ], int n)
{
      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 ("\n After sorting array elements are - \n");
   printArr (a, n);
   return 0;
}
```

Output:

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

Insertion Sort Algorithm

The working procedure of insertion sort is also simple.

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

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is **O(n²)**, where n is the number of items.

Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as –

- o   Simple implementation

- o   Efficient for small data sets

- o   Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

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

Algorithm

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

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

Step 2   :   Pick the next element, and store it separately in a **key.**

Step 3   :   Now, compare the **key** with all elements in the sorted array.

Step 4   :   If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5   :   Insert the value.

Step 6   :   Repeat until the array is sorted.

Working of Insertion Sort Algorithm:

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

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

Let the elements of array are –

| 12 | 31 | 25 | 8 | 32 | 17 |

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

| 12 | 31 | 25 | 8 | 32 | 17 |

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.

| 12 | 31 | 25 | 8 | 32 | 17 |

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

| 12 | 31 | 25 | 8 | 32 | 17 |

| 12 | 31 | 25 | 8 | 32 | 17 |

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.

| 12 | 25 | 31 | 8 | 32 | 17 |

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

| 12 | 25 | 31 | 8 | 32 | 17 |

| 12 | 25 | 31 | 8 | 32 | 17 |

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

| 12 | 25 | 8 | 31 | 32 | 17 |

After swapping, elements 25 and 8 are unsorted.

| 12 | 25 | 8 | 31 | 32 | 17 |

So, swap them.

| 12 | 8 | 25 | 31 | 32 | 17 |

Now, elements 12 and 8 are unsorted.

| 12 | 8 | 25 | 31 | 32 | 17 |

So, swap them too.

| 8 | 12 | 25 | 31 | 32 | 17 |

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

| 8 | 12 | 25 | 31 | 32 | 17 |

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

| 8 | 12 | 25 | 31 | 32 | 17 |

Move to the next elements that are 32 and 17.

| 8 | 12 | 25 | 31 | 32 | 17 |

17 is smaller than 32. So, swap them.

| 8 | 12 | 25 | 31 | 17 | 32 |

| 8 | 12 | 25 | 31 | 17 | 32 |

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

| 8 | 12 | 25 | 17 | 31 | 32 |

| 8 | 12 | 25 | 17 | 31 | 32 |

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

| 8 | 12 | 17 | 25 | 31 | 32 |

Now, the array is completely sorted.

Insertion Sort Complexity:

Best Case Complexity:

It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion 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 insertion sort is **O(n²)**.

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 insertion sort is **O(n²)**.
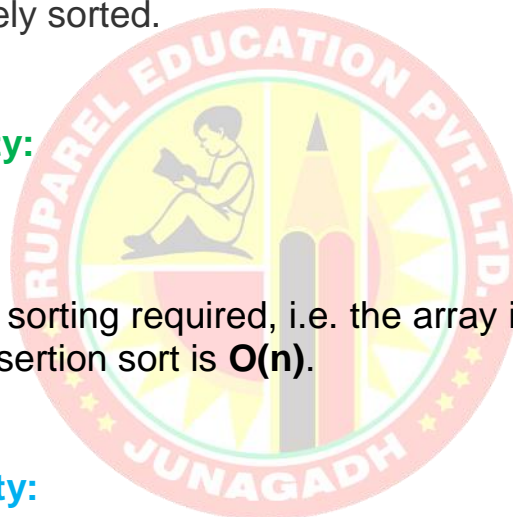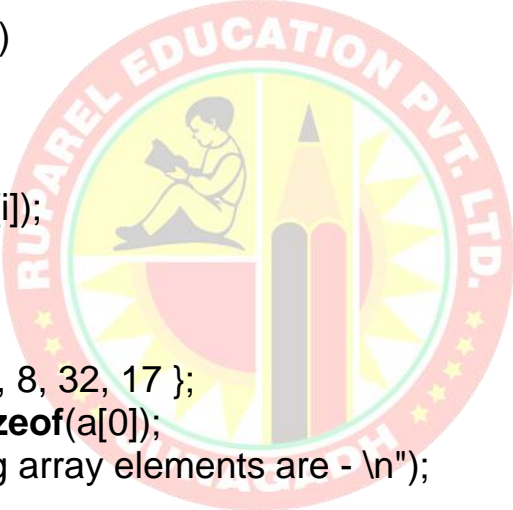
Program: Write a program to implement Insertion Sort in C language.

```c
#include <stdio.h>
void insert (int a[ ], int n)
{
      int i, j, temp;
      for (i = 1; i < n; i++)
      {
            temp = a[i];
            j = i - 1;
            while(j>=0 && temp <= a[j])
            {
                  a[j+1] = a[j];
                  j = j-1;
            }
            a[j+1] = temp;
      }
}
void printArr (int a[ ], int n)
{
      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 ("\n After sorting array elements are - \n");
      printArr (a, n);
      return 0;
}
```

Output:

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

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.**

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 0th position will be compared with the element at 4th position. If the 0th element is greater, it will be swapped with the element at 4th position. Otherwise, it remains the same. This process will continue for the remaining elements.

At the interval of 4, the sublists are {33, 12}, {31, 17}, {40, 25}, {8, 42}.
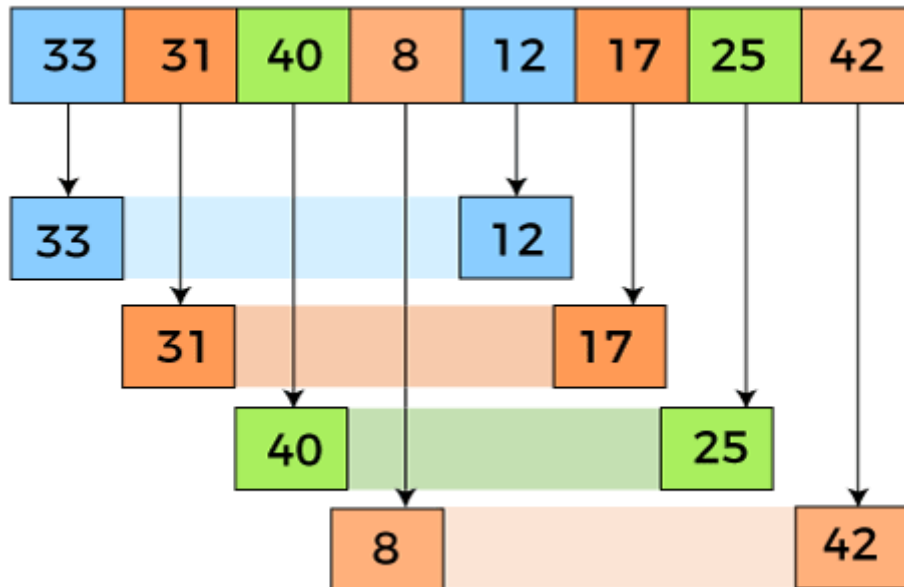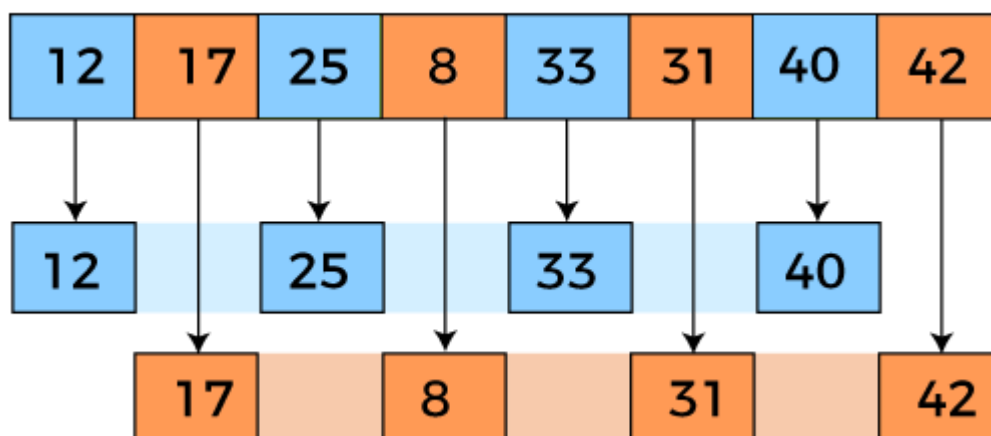
Now, we have to compare the values in every sub-list. After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows –

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

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

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

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

Now, the array is sorted in ascending order.

Shell Sort Complexity:

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).**
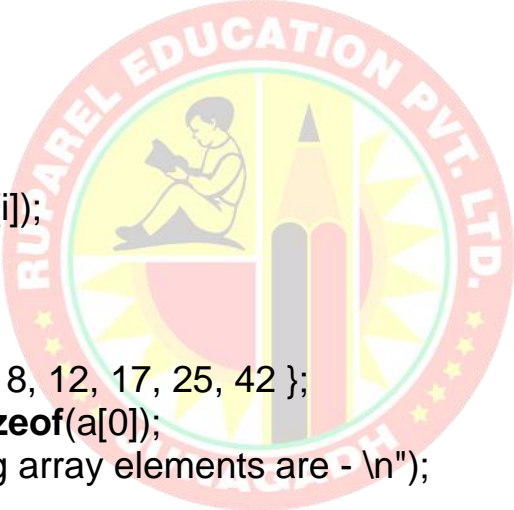
Worst Case Complexity:

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(n²).**

Program: **Write a program to Implement Shell Sort in C language.**

```c
#include <stdio.h>
int shell (int a[], int n)
{
        for (int interval = n/2; interval > 0; interval /= 2)
        {
                for (int i = interval; i < n; i += 1)
                {
                        int temp = a[i];
                        int j;
                        for (j = i; j >= interval && a[j - interval] > temp; j -= interval)
                        a[j] = a[j - interval];
                        a[j] = temp;
                }
        }
        return 0;
}
void printArr (int a[], int n)
{
        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;
}
```

Output:

```
Before sorting array elements are -
33 31 40 8 12 17 25 42
After applying shell sort, the array elements are -
8 12 17 25 31 33 40 42
```

Heap Sort 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.

Working of Heap Sort Algorithm:

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

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

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

- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

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

| 81	89	9	11	14	76	54	22

First, we have to construct a heap from the given array and convert it into max heap.
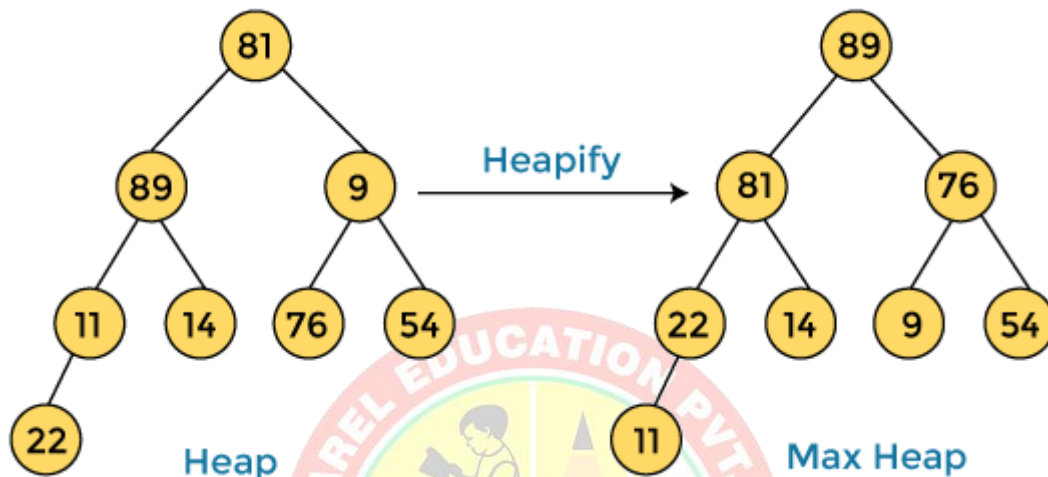
After converting the given heap into max heap, the array elements are –

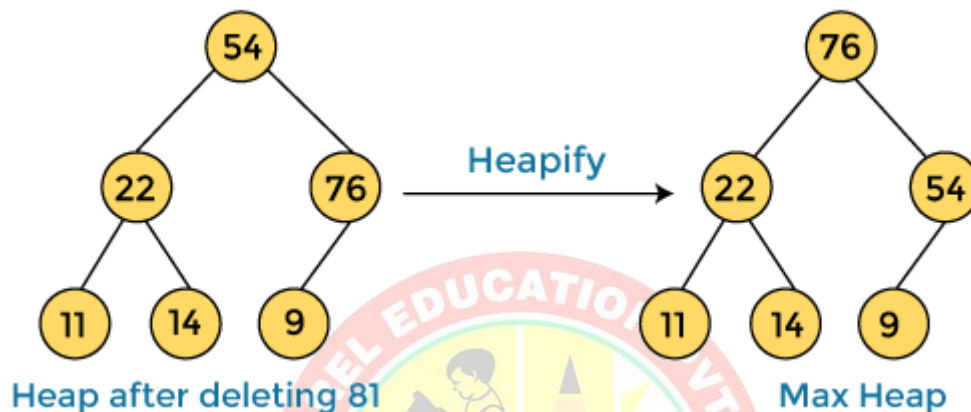| 89	81	76	22	14	9	54	11

Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.
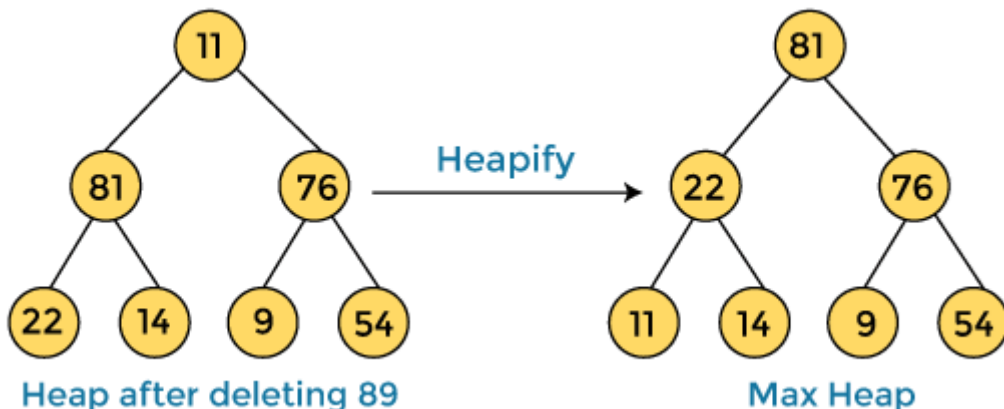
After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are –

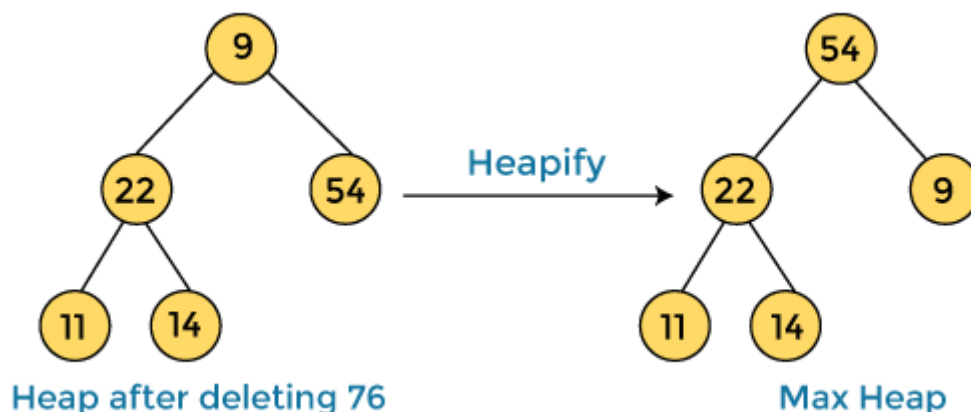| 81	22	76	11	14	9	54	89

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

Heap after deleting 81 → Heapify → Max Heap

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

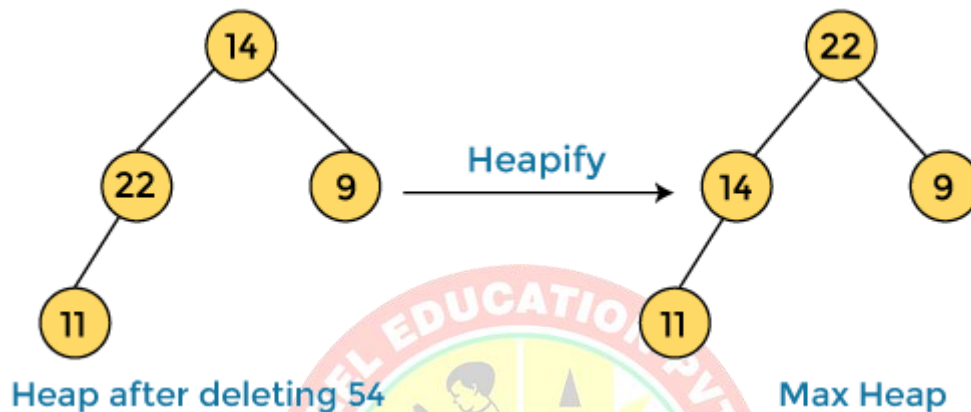| 76	22	54	11	14	9	81	89

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

Heap after deleting 76 → Heapify → Max Heap

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

54	22	9	11	14	76	81	89

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

Heap after deleting 54   Max Heap

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

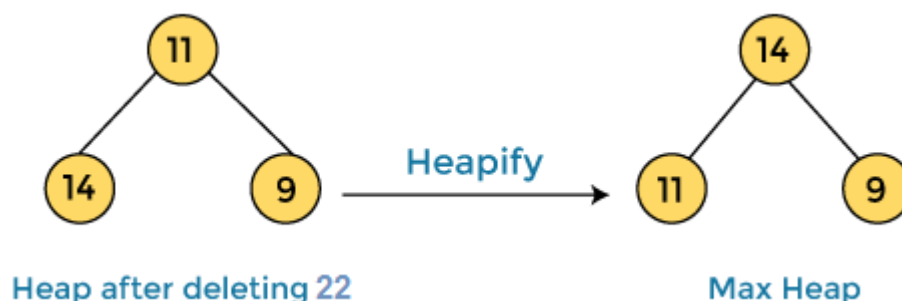| 22	14	9	11	54	76	81	89

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

Heap after deleting 22   Max Heap

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

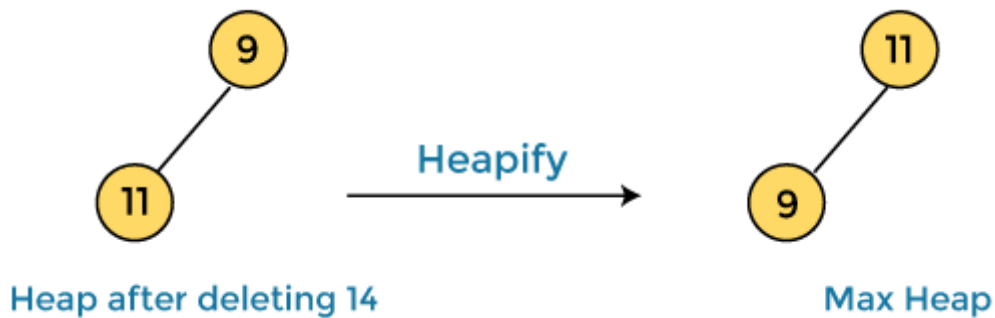| 14 | 11 | 9 | 22 | 54 | 76 | 81 | 89 |

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

Heap after deleting 14 → Heapify → Max Heap

After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are –
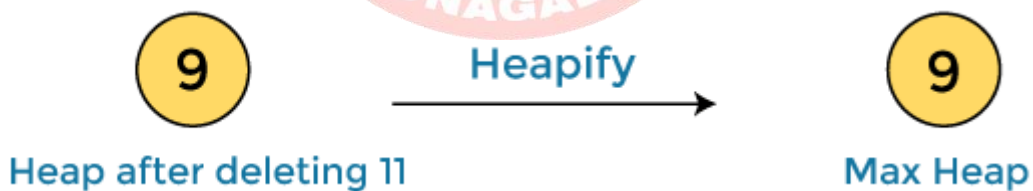
| 11 | 9 | 14 | 22 | 54 | 76 | 81 | 89 |

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

Heap after deleting 11 → Heapify → Max Heap

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

| 9 | 11 | 14 | 22 | 54 | 76 | 81 | 89 |

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

9 → Remove 9 → Empty

After completion of sorting, the array elements are –

9	11	14	22	54	76	81	89

Now, the array is completely sorted.

Heap Sort Complexity:

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).**

Program: Write a program to implement Heap Sort in C language.

```
#include <stdio.h>
void heapify (int a[ ], int n, int i)
{
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;
        if (left < n && a[left] > a[largest])
                largest = left;
        if (right < n && a[right] > a[largest])
                largest = right;
        if (largest != i)
        {
                int temp = a[i];
```

```c
            a[i] = a[largest];
            a[largest] = temp;
            heapify (a, n, largest);
        }
}
void heapSort (int a[], int n)
{
        for (int i = n / 2 - 1; i >= 0; i--)
        heapify (a, n, i);
        for (int i = n - 1; i >= 0; i--)
        {
                int temp = a[0];
                a[0] = a[i];
                a[i] = temp;
                heapify (a, i, 0);
        }
}
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;
}
```
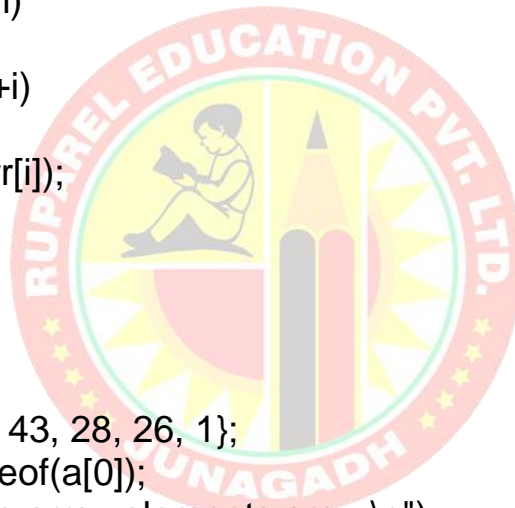
Output:

```
Before sorting array elements are -
48 10 23 43 28 26 1
After sorting array elements are -
1 10 23 26 28 43 48
```

Bucket Sort Algorithm

The data items in the bucket sort are distributed in the form of buckets. In coding or technical interviews for software engineers, sorting algorithms are widely asked. So, it is important to discuss the topic.

Bucket sort is a sorting algorithm that separate the elements into multiple groups said to be buckets.

Elements in bucket sort are first uniformly divided into groups called buckets, and then they are sorted by any other sorting algorithm. After that, elements are gathered in a sorted manner.

The basic procedure of performing the bucket sort is given as follows –

- o First, partition the range into a fixed number of buckets.
- o Then, toss every element into its appropriate bucket.
- o After that, sort each bucket individually by applying a sorting algorithm.
- o And at last, concatenate all the sorted buckets.

The advantages of bucket sort are –

- o Bucket sort reduces the no. of comparisons.
- o It is asymptotically fast because of the uniform distribution of elements.

The limitations of bucket sort are –

- o It may or may not be a stable sorting algorithm.
- o It is not useful if we have a large array because it increases the cost.
- o It is not an in-place sorting algorithm, because some extra space is required to sort the buckets.

Bucket sort is commonly used –

- o With floating-point values.
- o When input is distributed uniformly over a range.

Scatter-gather approach:

We can understand the Bucket sort algorithm via scatter-gather approach. Here, the given elements are first scattered into buckets. After scattering, elements in each bucket are sorted using a stable sorting algorithm. At last, the sorted elements will be gathered in order.

Let's take an unsorted array to understand the process of bucket sort. It will be easier to understand the bucket sort via an example.

Let the elements of array are –

Now, create buckets with a range from **0 to 25**. The buckets range are **0-5, 5-10, 10-15, 15-20, 20-25**.

Elements are inserted in the buckets according to the bucket range.

Suppose the value of an item is 16, so it will be inserted in the bucket with the range 15-20. Similarly, every item of the array will insert accordingly.

This phase is known to be the **scattering of array elements**.

Now, **sort** each bucket individually. The elements of each bucket can be sorted by using any of the stable sorting algorithms.

At last, **gather** the sorted elements from each bucket in order

Now, the array is completely sorted.

Bucket Sort Complexity:

Best Case Complexity:

It occurs when there is no sorting required, i.e. the array is already sorted. In Bucket sort, best case occurs when the elements are uniformly distributed in the buckets.

The complexity will be better if the elements are already sorted in the buckets. If we use the insertion sort to sort the bucket elements, the overall complexity will be linear, i.e., O(n + k), where O(n) is for making the buckets, and O(k) is for sorting the bucket elements using algorithms with linear time complexity at best case. The best-case time complexity of bucket sort is **O(n + k)**.

Average Case Complexity:

It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. Bucket sort runs in the linear time, even when the elements are uniformly distributed. The average case time complexity of bucket sort is **O(n + K)**.

Worst Case Complexity:

In bucket sort, worst case occurs when the elements are of the close range in the array, because of that, they have to be placed in the same bucket. So, some buckets have more number of elements than others. The complexity will get worse when the elements are in the reverse order. The worst-case time complexity of bucket sort is **O(n²)**.

Program: Write a program to implement Bucket Sort in C language.
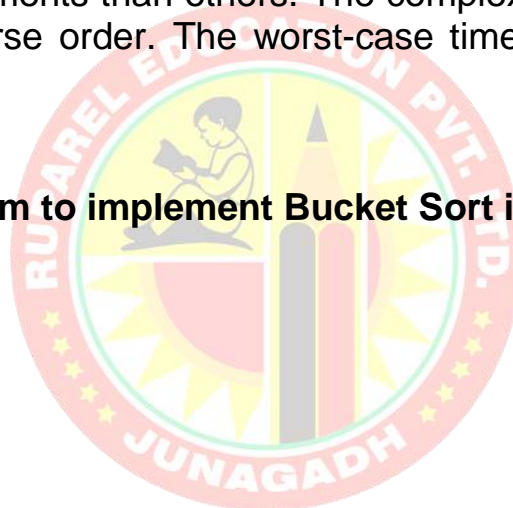
```c
#include <stdio.h>

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;
}

void bucket (int a[ ], int n)
{
        int max = getMax (a, n);
        int bucket[max], i;

        for (int i = 0; i <= max; i++)
        {
                bucket[i] = 0;
        }
```

```c
    for (int i = 0; i < n; i++)
    {
        bucket[ a[ i ] ]++;
    }

    for (int i = 0, j = 0; i <= max; i++)
    {
        while (bucket[i] > 0)
        {
            a[j++] = i;
            bucket[i]--;
        }
    }
}

void printArr (int a[ ], int n)
{
    for (int i = 0; i < n; ++i)
    printf ("%d  ", a[i]);
}

int main ()
{
    int a[ ] = {54, 12, 84, 57, 69, 41, 9, 5};
    int n = sizeof(a) / sizeof(a[0]);

    printf ("Before sorting array elements are - \n");
    printArr (a, n);
    bucket (a, n);
    printf ("\n After sorting array elements are - \n");
    printArr (a, n);
}
```

Output:

```
Before sorting array elements are -
54  12  84  57  69  41  9  5
After sorting array elements are -
5  9  12  41  54  57  69  84
```

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.

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.

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.
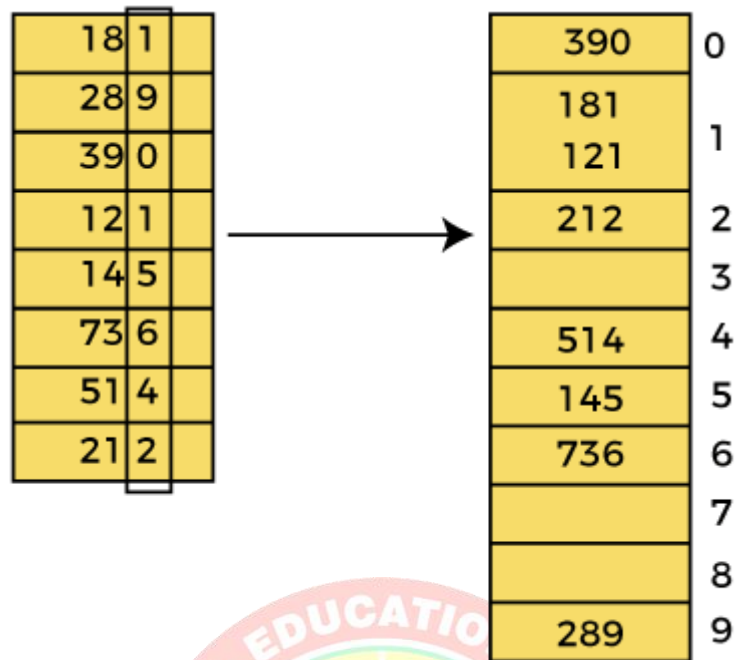
181	289	390	121	145	736	514	212

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:

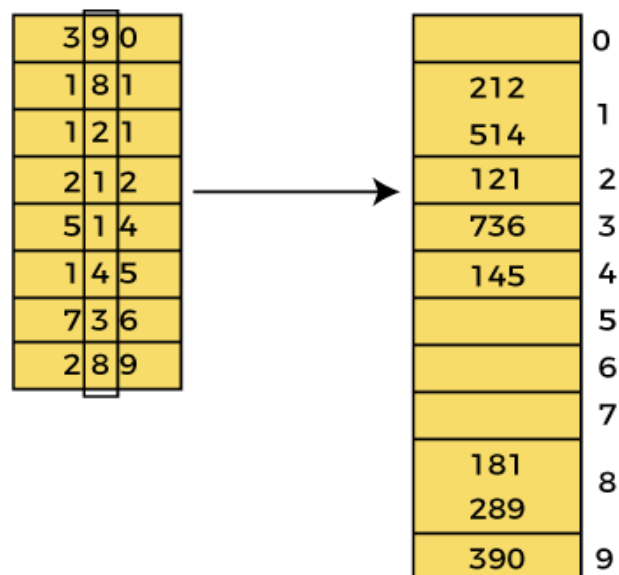In the first pass, the list is sorted on the basis of the digits at 0's place.

18	1			390	0
28	9			181	1
39	0			121	
12	1	→		212	2
14	5				3
73	6			514	4
51	4			145	5
21	2			736	6
				7
				8
			289	9

After the first pass, the array elements are –

390	181	121	212	514	145	736	289

Pass 2:

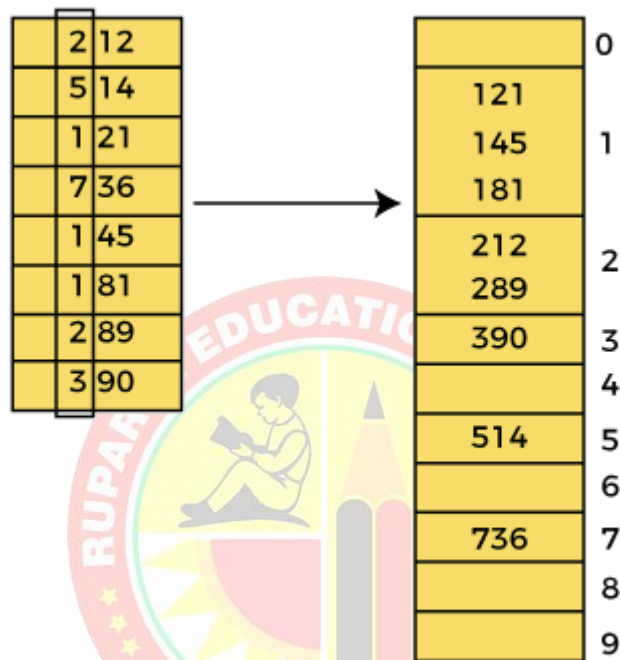In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10th place).

3	9	0			0
1	8	1		212	1
1	2	1		514	
2	1	2	→	121	2
5	1	4		736	3
1	4	5		145	4
7	3	6			5
2	8	9			6
			7
		181	8
		289	
		390	9

After the second pass, the array elements are –

| 212 | 514 | 121 | 736 | 145 | 181 | 289 | 390 |

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 –

| 121 | 145 | 181 | 212 | 289 | 390 | 514 | 736 |

Now, the array is sorted in ascending order.

Radix Sort Complexity:

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)**.

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

```c
#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;
}


void countingSort (int a[], int n, int place)
{
        int output [n + 1];
        int count [10] = {0};

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

        for (int i = 1; i < 10; i++)
                count[i] += count[i - 1];
```

```
        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];
}

void radixsort (int a[ ], int n)
{
        int max = getMax(a, n);

        for (int place = 1; max / place > 0; place *= 10)
                countingSort(a, n, place);
}

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);
}
```
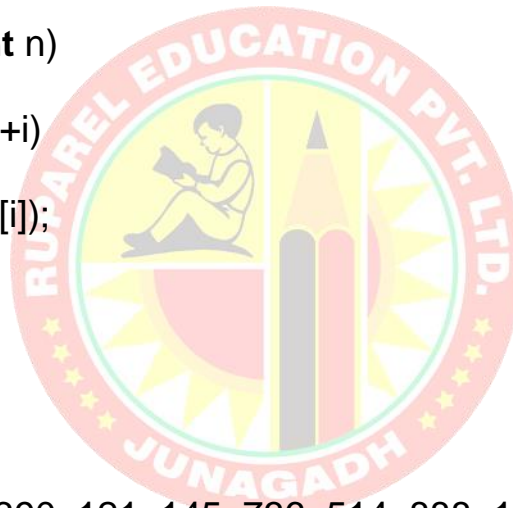
Output:

```
Before sorting array elements are -
181  289  390  121  145  736  514  888  122
After applying Radix sort, the array elements are -
121  122  145  181  289  390  514  736  888
```

Counting Sort Algorithm:

Counting sort is a sorting technique that is based on the keys between specific ranges.

This sorting technique doesn't perform sorting by comparing elements. It performs sorting by counting objects having distinct key values like hashing.

After that, it performs some arithmetic operations to calculate each object's index position in the output sequence. Counting sort is not used as a general-purpose sorting algorithm.

Counting sort is effective when range is not greater than number of objects to be sorted. It can be used to sort the negative input values.
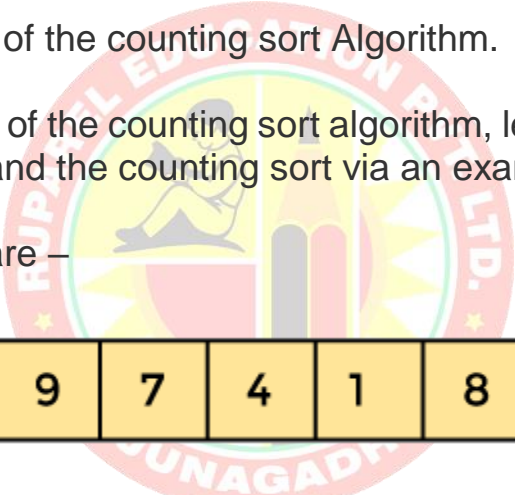
Now, let's see the algorithm of counting sort.

Working of Counting Sort Algorithm:

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

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

Let the elements of array are –

2	9	7	4	1	8	4

STEP-1   Find the maximum element from the given array. Let **max** be the maximum element.

max

9

2	7	4	1	8	4

STEP-2   Now, initialize array of length **max + 1** having all 0 elements. This array will be used to store the count of the elements in the given array.

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

Count array

STEP-3   Now, we have to store the count of each array element at their corresponding index in the count array.

The count of an element will be stored as - Suppose array element '4' is appeared two times, so the count of element 4 is 2.

Hence, 2 is stored at the 4th position of the count array. If any element is not present in the array, place 0, i.e. suppose element '3' is not present in the array, so, 0 will be stored at 3rd position.

Given array	2	9	7	4	1	8	4

	0	1	2	3	4	5	6	7	8	9
Count array	0	1	1	0	2	0	0	1	1	1

Count of each stored element

Now, store the cumulative sum of **count** array elements. It will help to place the elements at the correct index of the sorted array.

0	1	2	3	4	5	6	7	8	9
0	1	2	0	2	0	0	1	1	1

1+1=2

0	1	2	3	4	5	6	7	8	9
0	1	2	2	2	0	0	1	1	1

2+0=2

Similarly, the cumulative count of the count array is –

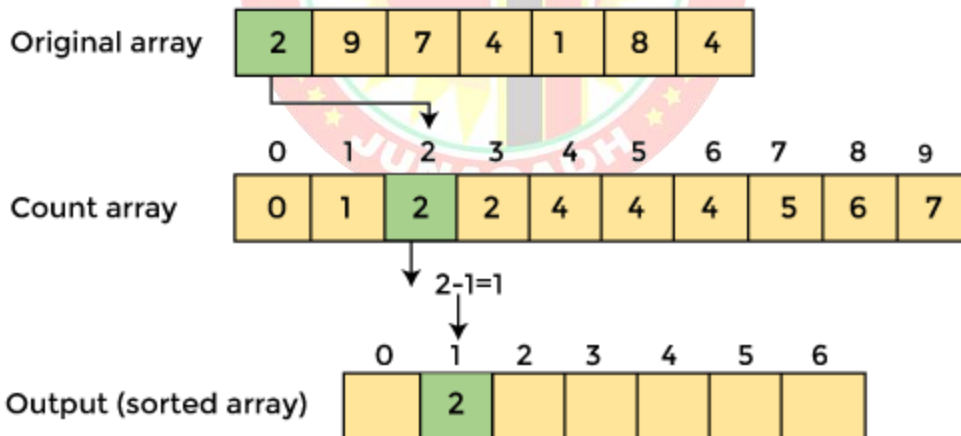| 0	1	2	3	4	5	6	7	8	9
0	1	2	2	4	4	4	5	6	7

Cumulative count

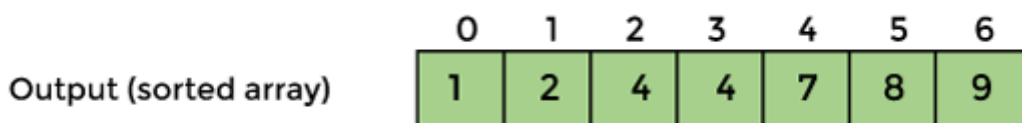**STEP-4** Now, find the index of each element of the original array

For element 9
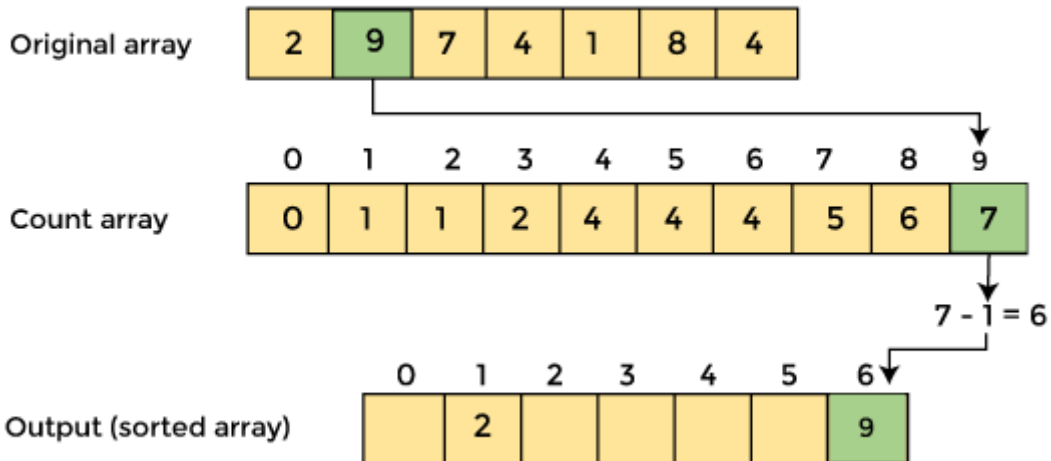
After placing element at its place, decrease its count by one. Before placing element 2, its count was 2, but after placing it at its correct position, the new count for element 2 is 1.

For element 2

Similarly, after sorting, the array elements are –

Now, the array is completely sorted.

Counting Sort Complexity:

Best Case Complexity:

It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of counting sort is **O(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 counting sort is **O(n + k)**.

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 counting sort is **O(n + k)**.

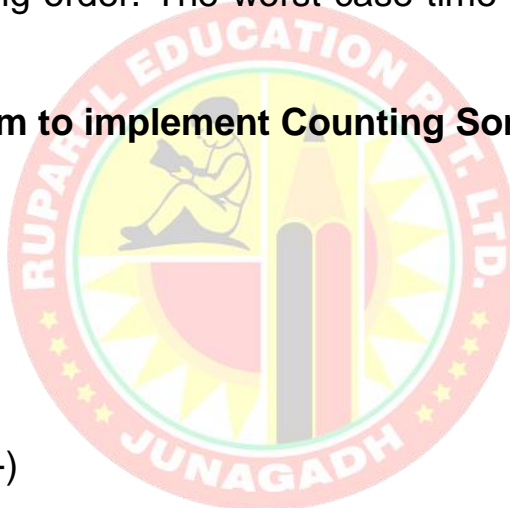**Program: Write a program to implement Counting Sort in C language.**

```c
#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;
}

void countSort (int a[ ], int n)
{
        int output [n+1];
        int max = getMax (a, n);
        int count [max+1];

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

```c
    for (int i = 0; i < n; i++)
    {
            count[a[i]]++;
    }

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

    for (int i = n - 1; i >= 0; i--)
    {
            output[count[a[i]] - 1] = a[i];
            count[a[i]]--;
    }

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

void printArr (int a[ ], int n)
{
    int i;

    for (i = 0; i < n; i++)
            printf ("%d ", a[i]);
}

int main ()
{
    int a[ ] = { 11, 30, 24, 7, 31, 16 };
    int n = sizeof(a)/sizeof(a[0]);
    printf ("Before sorting array elements are - \n");
    printArr (a, n);
    countSort (a, n);
    printf ("\n After sorting array elements are - \n");
    printArr (a, n);
    return 0;
}
```
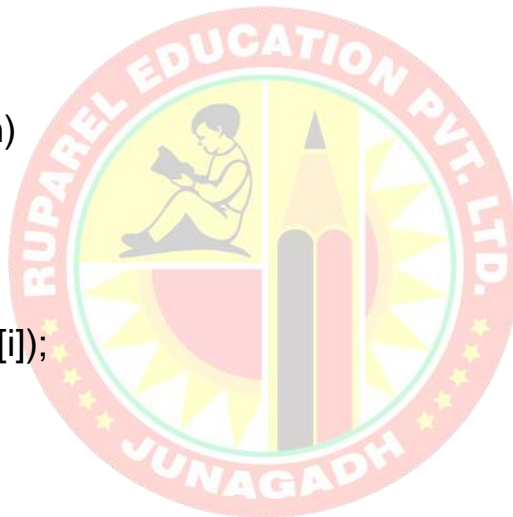
Output:

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