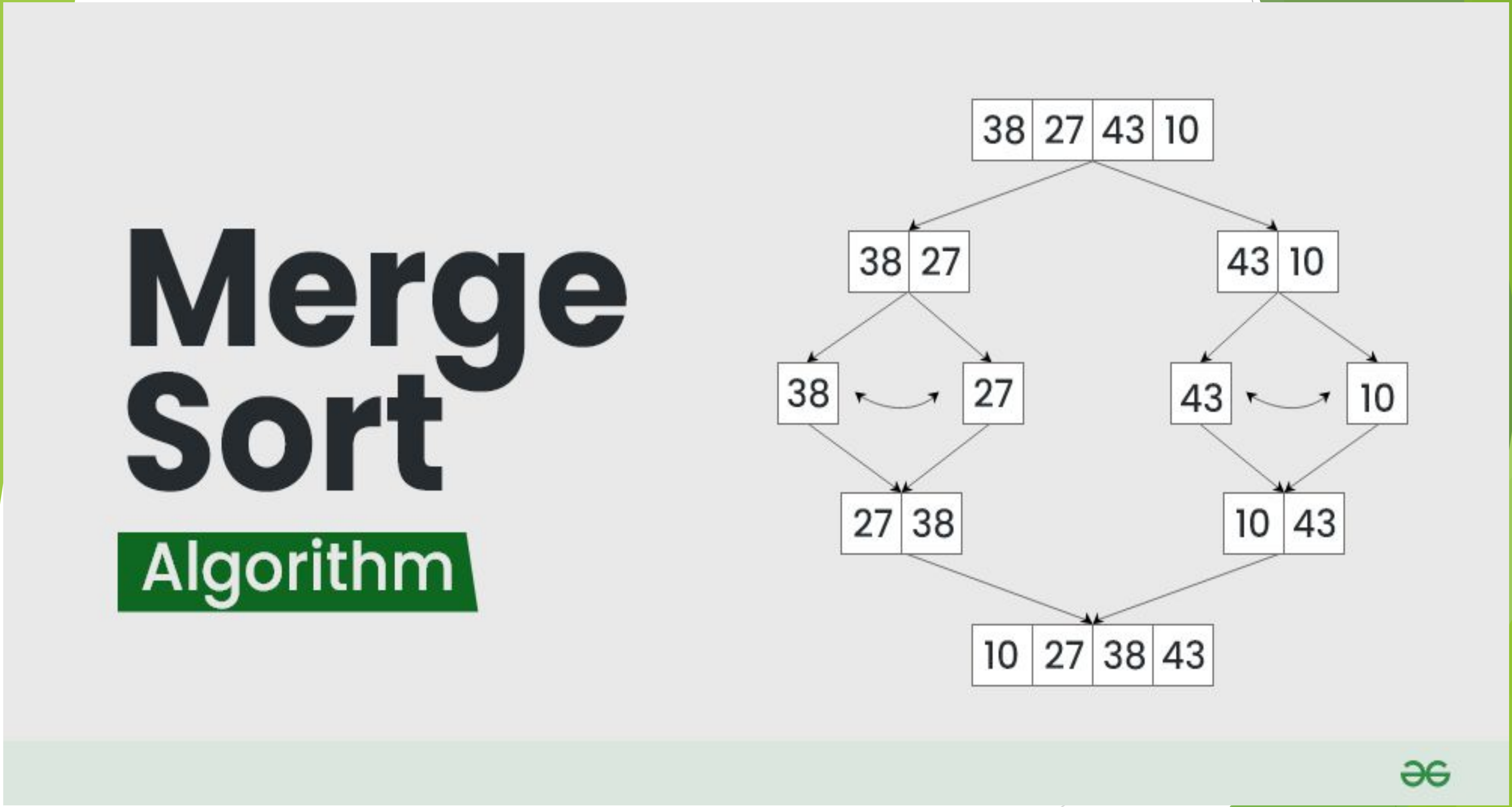
UNIT IV


Merge Sort
Algorithm
38 27 43 10
38 27
43 10
38
27
43
10
27 38
10 43
10 27 38 43

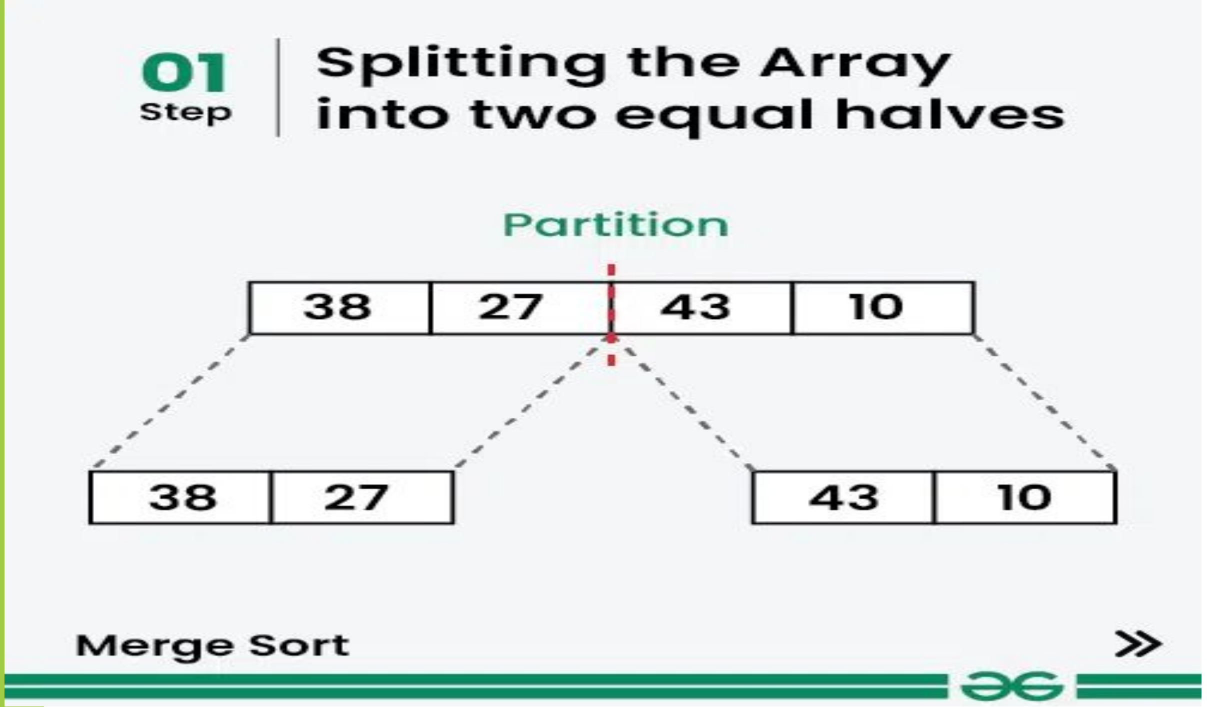
How does Merge Sort work?

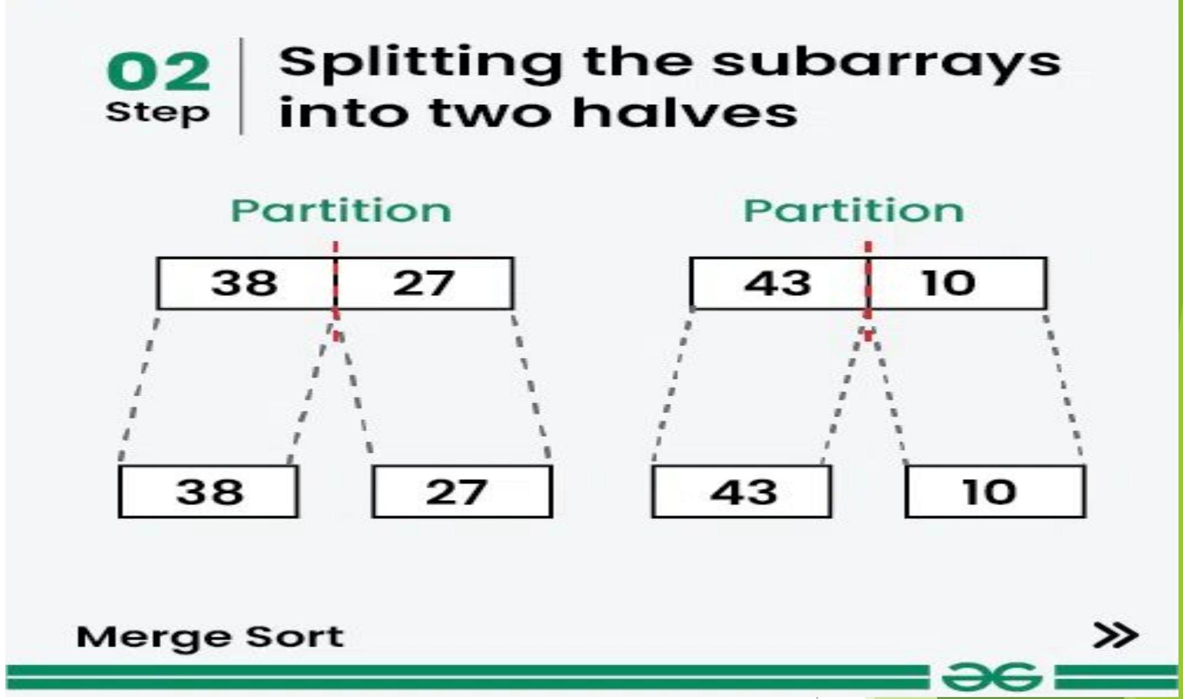
Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

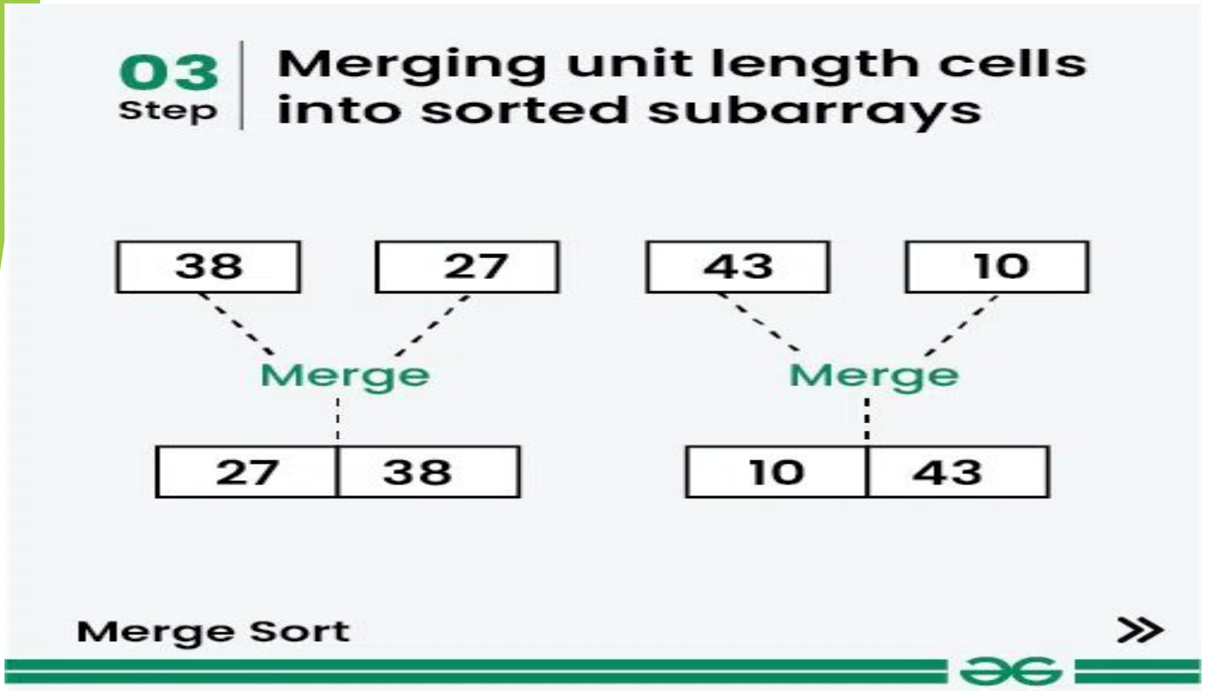
Illustration:

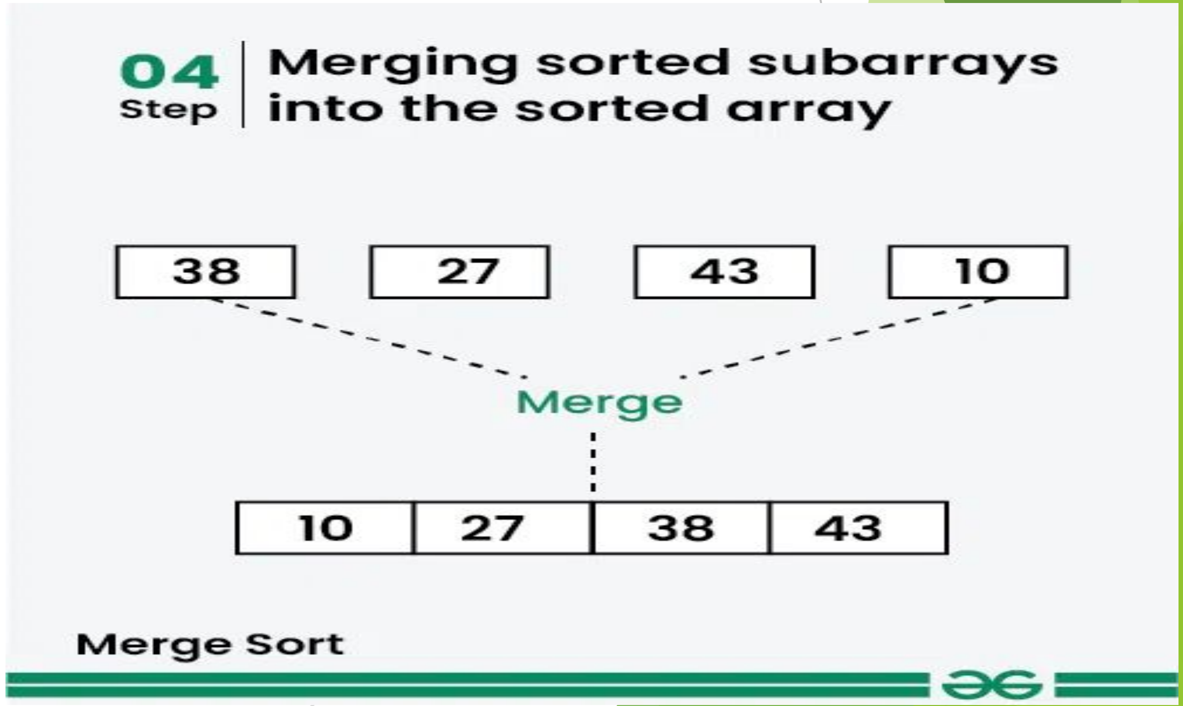
Lets consider an array **arr[] = {38, 27, 43, 10}**

- These sorted subarrays are merged together, and we get bigger sorted subarrays.
- This merging process is continued until the sorted array is built from the smaller subarrays.
- Initially divide the array into two equal halves:
- These subarrays are further divided into two halves. Now they become array of unit length that can no longer be divided and array of unit length are always sorted.
- The following diagram shows the complete merge sort process for an example array {38, 27, 43, 10}.


01 Step | Splitting the Array into two equal halves


02 Step | Splitting the subarrays into two halves


03 Step | Merging unit length cells into sorted subarrays


04 Step | Merging sorted subarrays into the sorted array

Complexity Analysis of Merge Sort :

Time Complexity: O(N log(N)), Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \theta(n)$$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is θ(Nlog(N)).

The time complexity of Merge Sort is θ(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Auxiliary Space: O(N), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

Applications of Merge Sort:

- **Sorting large datasets:** Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
- **External sorting:** Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
- **Custom sorting:** Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.
- Inversion Count Problem .

QuickSort :

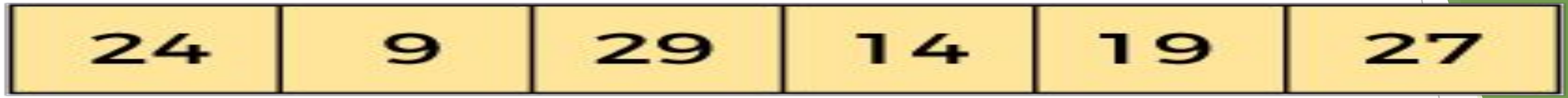
QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How does QuickSort work?

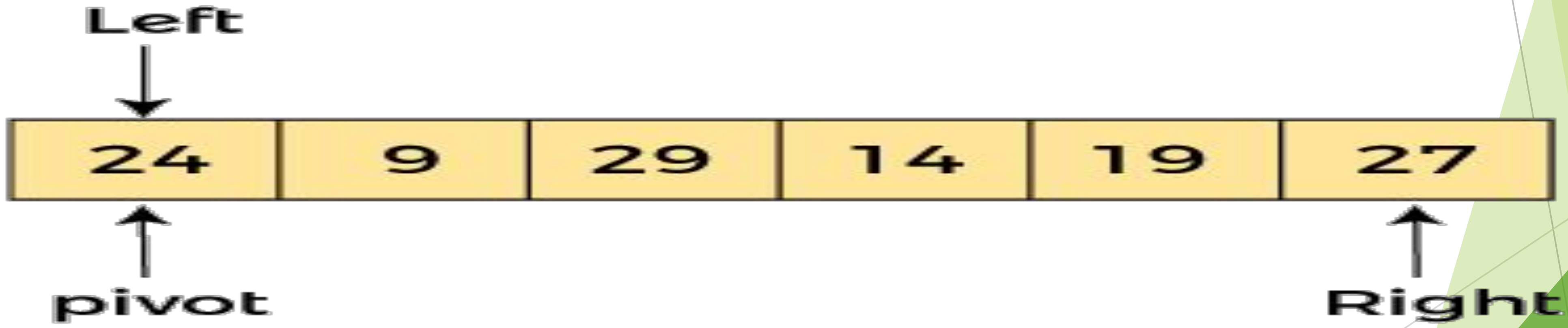
The key process in **quickSort** is a **partition()**. The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

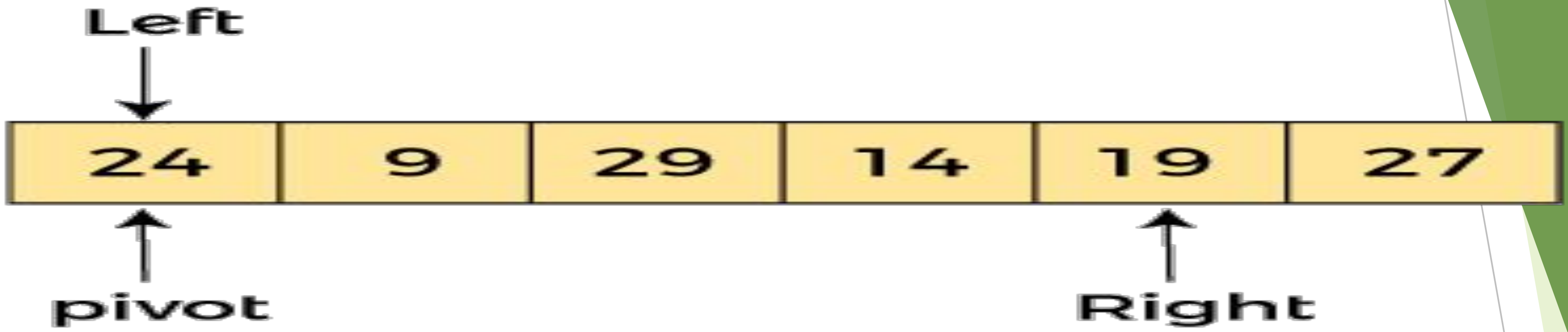
Let the elements of array are -



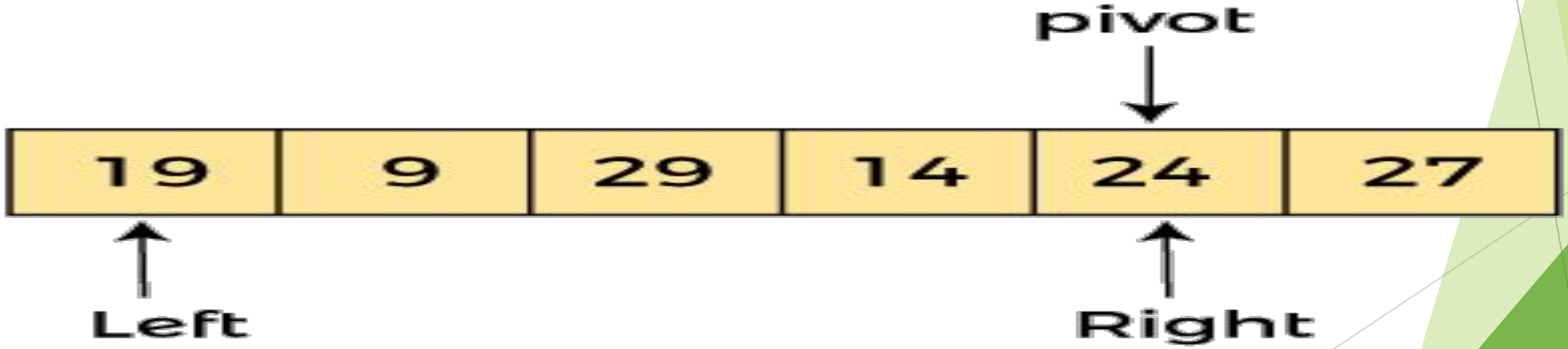
In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.
Since, pivot is at left, so algorithm starts from right and move towards left.



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

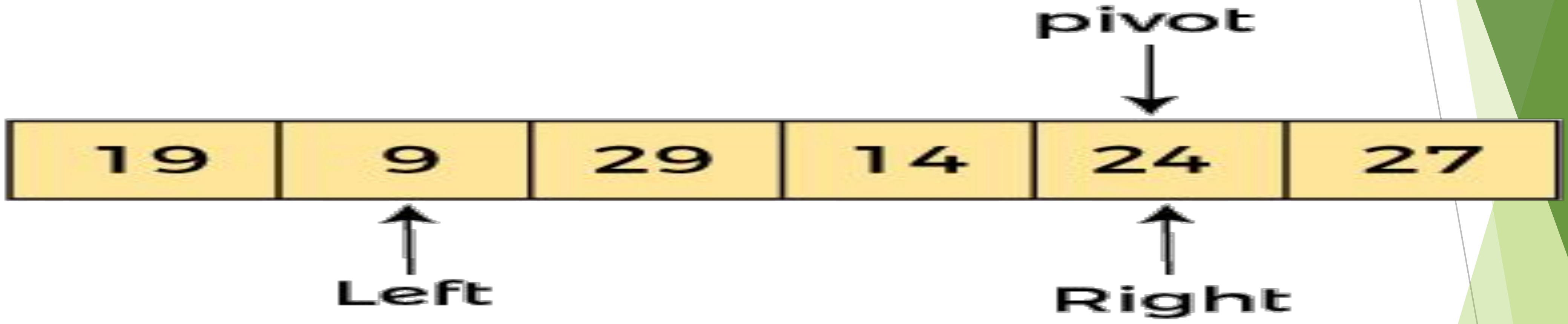


Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.
Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

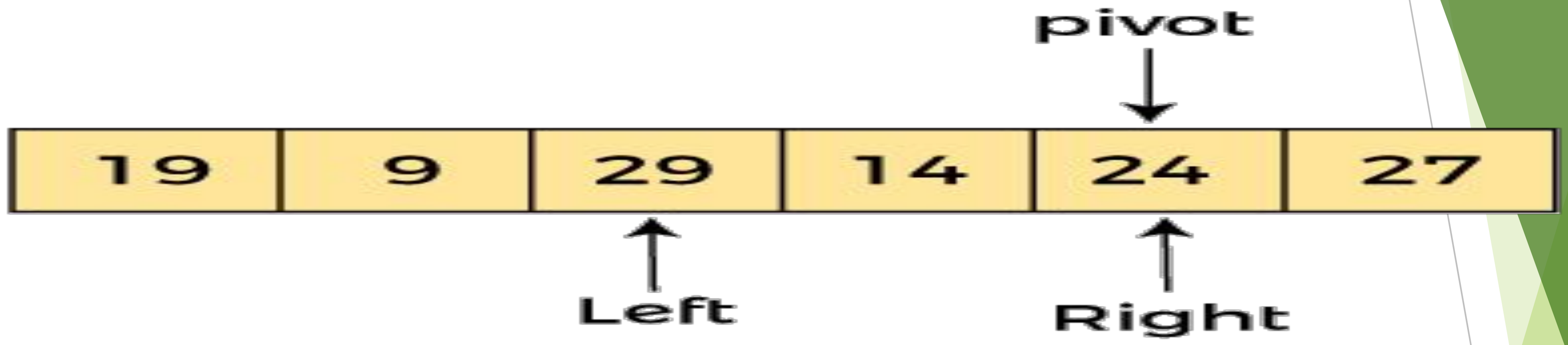


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

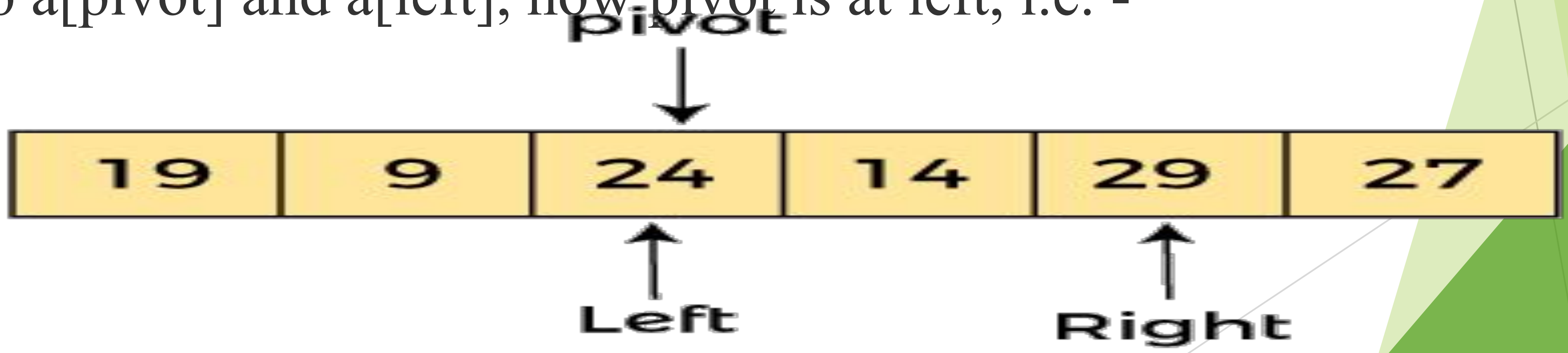
As a[pivot] > a[left], so algorithm moves one position to right a -



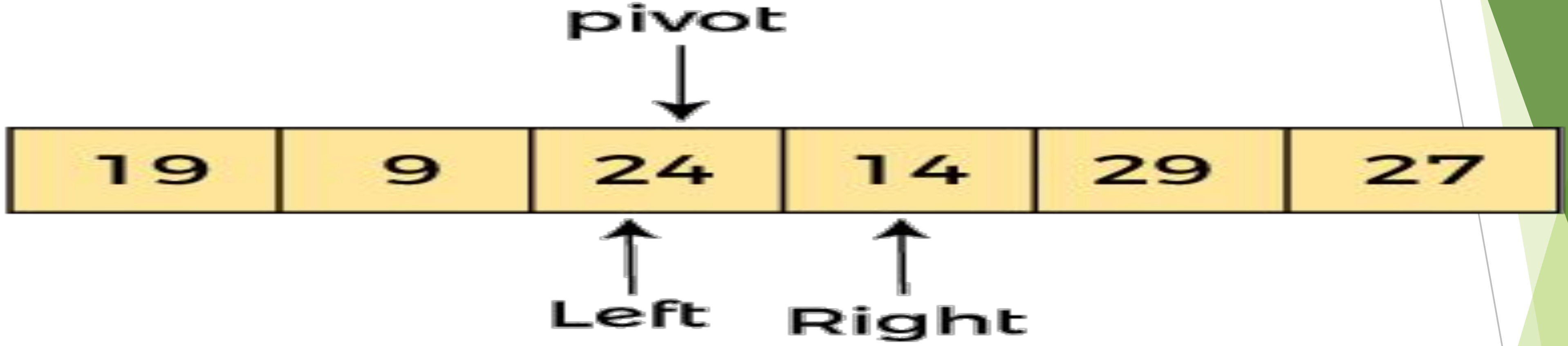
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



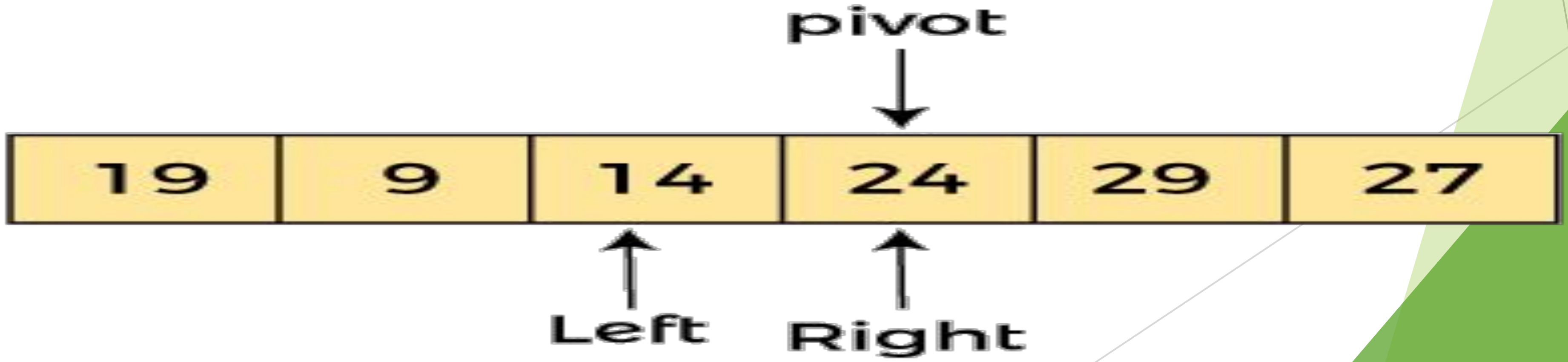
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



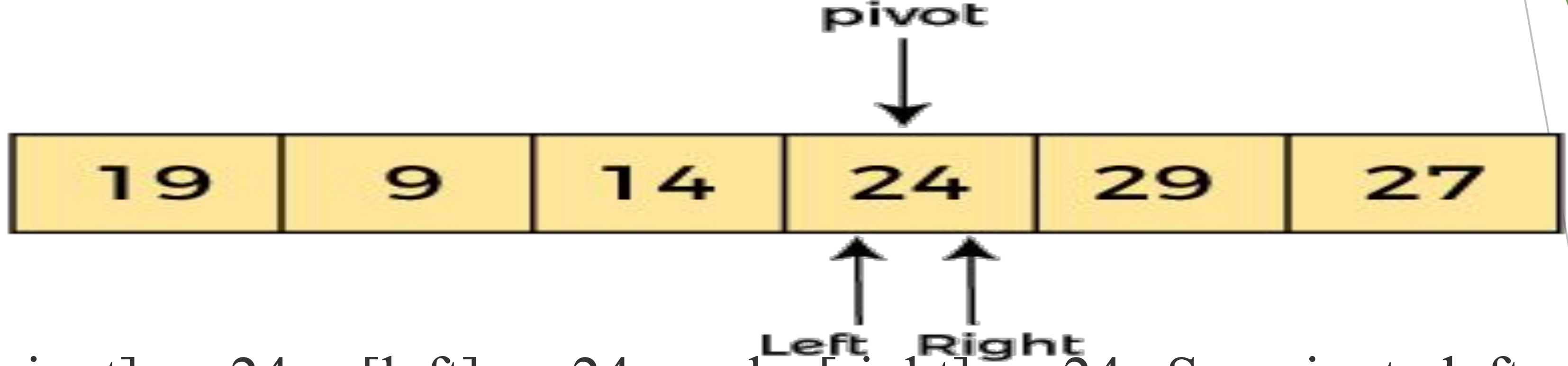
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



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. -



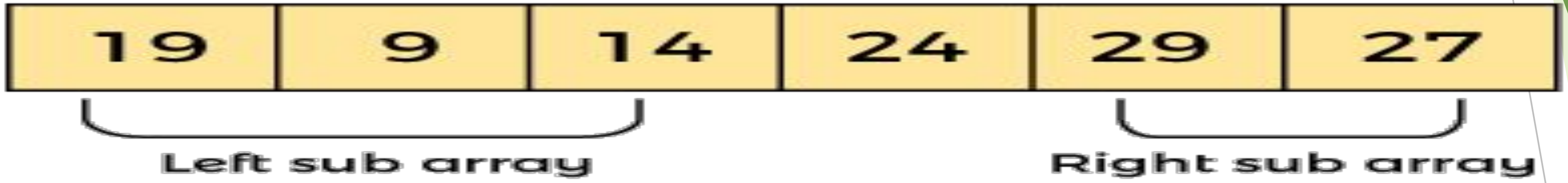
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



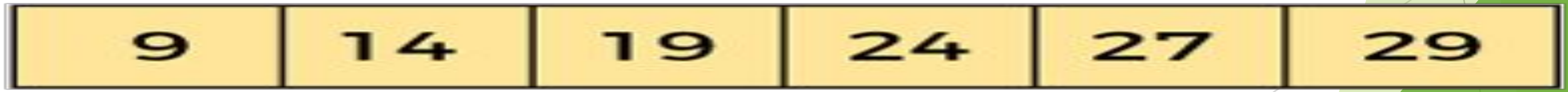
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.
Element 24, which is the pivot element is placed at its exact position.

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.

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.

Time Complexity

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

Abstract data types :

An **abstract data type** (ADT) is a conceptual model that defines data in terms of a set of possible values and a set of operations that can be carried out on that data, regardless of a specific implementation.

The term "abstract" is used because the details of how the data type is implemented is abstracted so that the data type can be studied without having to worry about how it is implemented. There are often many ways to implement an abstract data type, depending on the programming language being used.

ADT	Description
Queue	An ADT that follows the First-In-First-Out (FIFO) principle, where the first element added to the queue is the first one to be removed. Queues are often used to model real-life situations.
Stack	An ADT that follows the Last-In-First-Out (LIFO) principle, where the last element added to the stack is the first one to be removed. Stacks are often used for specialist tasks in computer science such as managing the call stack when a program is running.
Graph	An ADT that consists of a finite set of nodes (vertices) connected by edges (arcs). Graphs are used to model connections or relationships between objects and to represent data that does not have a linear structure
Tree	An ADT that represents a specific type of graph where all nodes are connected and there are no cycles (loops). Trees are often used for very specialist tasks in computer science such as Abstract Syntax Trees.
Hash table	An ADT that stores key-value pairs, allowing efficient lookup and retrieval of values based on their associated keys. Hash tables are often used to speed up searching for data and are often found in database applications.
Vector	An ADT that is used to represent properties that have both direction and magnitude (as opposed to a scalar that only has magnitude). Vectors are commonly used to represent movement. For example, the distance between two points can be represented as a vector, as can the velocity (speed and direction of movement) of an object.

Radix sort :

Radix sort is a step-wise sorting algorithm that starts the sorting from the least significant digit of the input elements. The sorting starts with the least significant digit of each element. These least significant digits are all considered individual elements and sorted first; followed by the second least significant digits. This process is continued until all the digits of the input elements are sorted.

Note – If the elements do not have same number of digits, find the maximum number of digits in an input element and add leading zeroes to the elements having less digits. It does not change the values of the elements but still makes them k-digit numbers.

Radix Sort Algorithm :

The radix sort algorithm makes use of the counting sort algorithm while sorting in every phase. The detailed steps are as follows −

Step 1 − Check whether all the input elements have same number of digits. If not, check for numbers that have maximum number of digits in the list and add leading zeroes to the ones that do not.

Step 2 − Take the least significant digit of each element.

Step 3 − Sort these digits using counting sort logic and change the order of elements based on the output achieved. For example, if the input elements are decimal numbers, the possible values each digit can take would be 0-9, so index the digits based on these values.

Step 4 − Repeat the Step 2 for the next least significant digits until all the digits in the elements are sorted.

Step 5 − The final list of elements achieved after kth loop is the sorted output.

Example

For the given unsorted list of elements, 236, 143, 26, 42, 1, 99, 765, 482, 3, 56, we need to perform the radix sort and obtain the sorted output list −

Step 1

Check for elements with maximum number of digits, which is 3. So we add leading zeroes to the numbers that do not have 3 digits. The list we achieved would be −

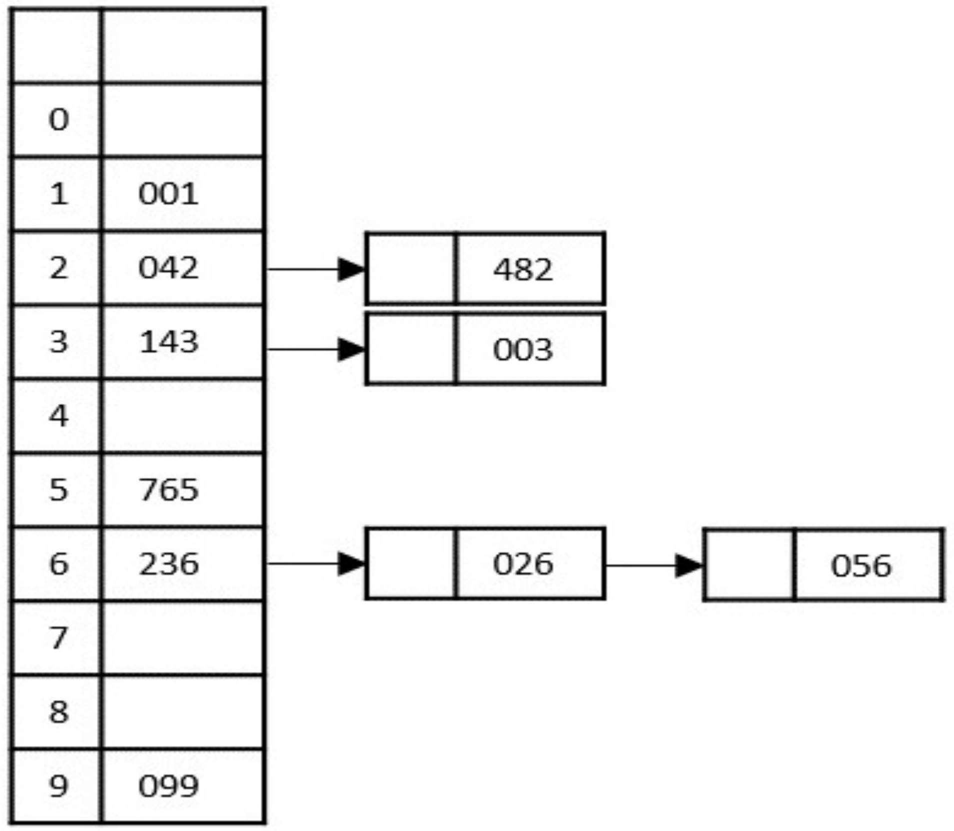
236, 143, 026, 042, 001, 099, 765, 482, 003, 056

Step 2

Construct a table to store the values based on their indexing. Since the inputs given are decimal numbers, the indexing is done based on the possible values of these digits, i.e., 0-9.

Step 3

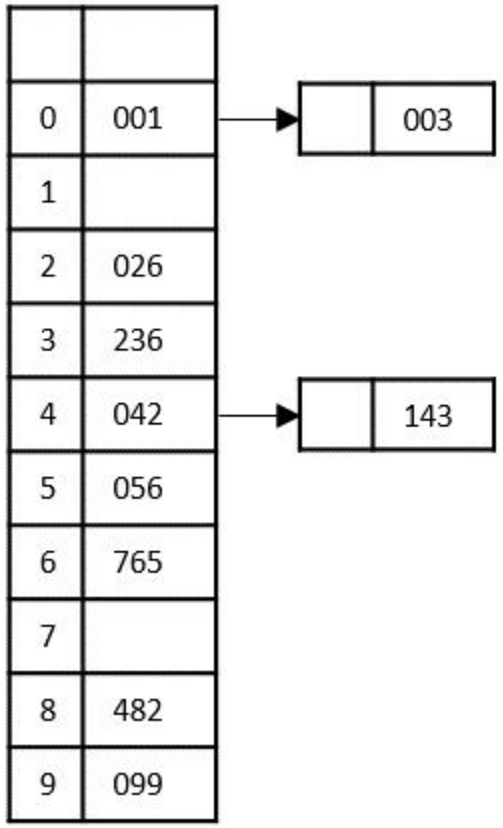
Based on the least significant digit of all the numbers, place the numbers on their respective indices.



The elements sorted after this step would be 001, 042, 482, 143, 003, 765, 236, 026, 056, 099.

Step 4

The order of input for this step would be the order of the output in the previous step. Now, we perform sorting using the second least significant digit.



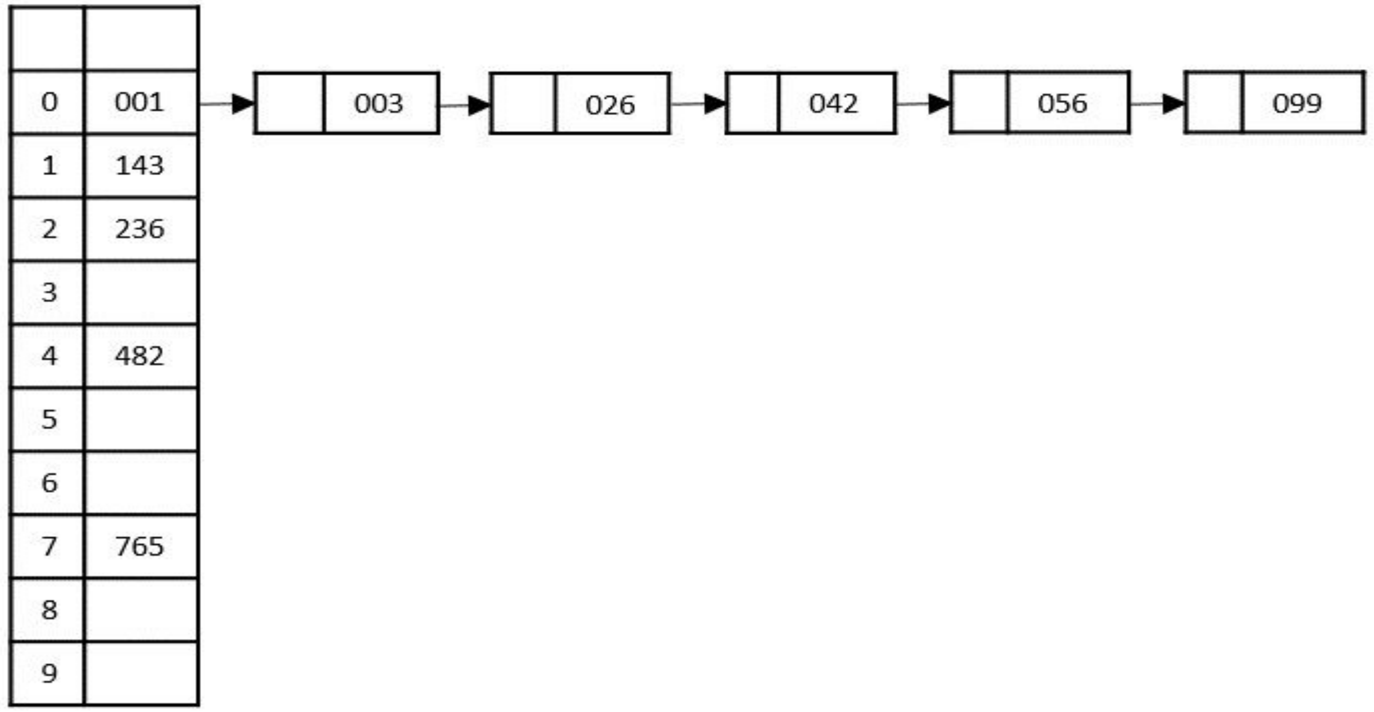
The order of the output achieved is 001, 003, 026, 236, 042, 143, 056, 765, 482, 099.

Step 5

The input list after the previous step is rearranged as −

001, 003, 026, 236, 042, 143, 056, 765, 482, 099

Now, we need to sort the last digits of the input elements.



Since there are no further digits in the input elements, the output achieved in this step is considered as the final output.

The final sorted output is −

1, 3, 26, 42, 56, 99, 143, 236, 482, 765

```
#include <iostream>
using namespace std;
void countsort(int a[], int n, int pos){
  int output[n + 1];
  int max = (a[0] / pos) % 10;
  for (int i = 1; i < n; i++) {
    if (((a[i] / pos) % 10) > max)
      max = a[i];
  }
  int count[max + 1];
  for (int i = 0; i < max; ++i)
    count[i] = 0;
  for (int i = 0; i < n; i++)
    count[(a[i] / pos) % 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] / pos) % 10] - 1] = a[i];
    count[(a[i] / pos) % 10]--;
  }
for (int i = 0; i < n; i++)
      a[i] = output[i];
}
void radixsort(int a[], int n){
   int max = a[0];
   for (int i = 1; i < n; i++)
      if (a[i] > max)
         max = a[i];
   for (int pos = 1; max / pos > 0; pos *= 10)
      countsort(a, n, pos);
}
```

```
int main(){
   int a[] = {236, 15, 333, 27, 9, 108, 76, 498};
   int n = sizeof(a) / sizeof(a[0]);
   cout <<"Before sorting array elements are: ";
   for (int i = 0; i < n; ++i) {
      cout <<a[i] << " ";
   }
   radixsort(a, n);
   cout <<"\nAfter sorting array elements are: ";
   for (int i = 0; i < n; ++i) {
      cout << a[i] << " ";
   }
   cout << "\n";
}
```

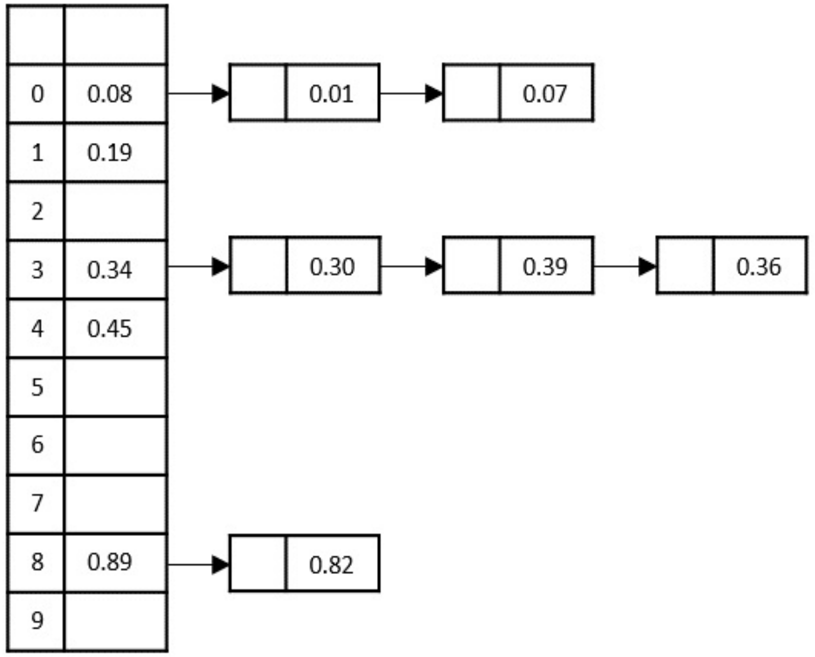
Output

```
Before sorting array elements are: 236 15 333 27 9 108 76 498
After sorting array elements are: 9 15 27 76 108 236 333 498
```

Bucket sort algorithm :

Bucket sort assumes that the input elements are drawn from a uniform distribution over the interval [0, 1).Hence, the bucket sort algorithm divides the interval [0, 1) into n equal parts, and the input elements are added to indexed *buckets* where the indices based on the lower bound of the (n element) value. Since the algorithm assumes the values as the independent numbers evenly distributed over a small range, not many elements fall into one bucket only.

For example, let us look at an input list of elements, 0.08, 0.01, 0.19, 0.89, 0.34, 0.07, 0.30, 0.82, 0.39, 0.45, 0.36. The bucket sort would look like −



Bucket Sort Algorithm

Let us look at how this algorithm would proceed further below −

Step 1 − Divide the interval in n equal parts, each part being referred to as a bucket. Say if n is 10, then there are 10 buckets; otherwise more.

Step 2 − Take the input elements from the input array A and add them to these output buckets B based on the computation formula, $B[i]= \lfloor n.A[i] \rfloor$

Step 3 − If there are any elements being added to the already occupied buckets, created a linked list through the corresponding bucket.

Step 4 − Then we apply insertion sort to sort all the elements in each bucket.

Step 5 − These buckets are concatenated together which in turn is obtained as the output.

Example

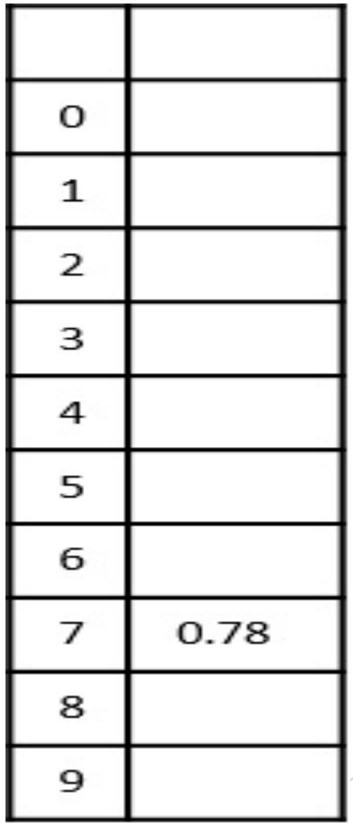
Consider, an input list of elements, 0.78, 0.17, 0.93, 0.39, 0.26, 0.72, 0.21, 0.12, 0.33, 0.28, to sort these elements using bucket sort −

Solution

Step 1

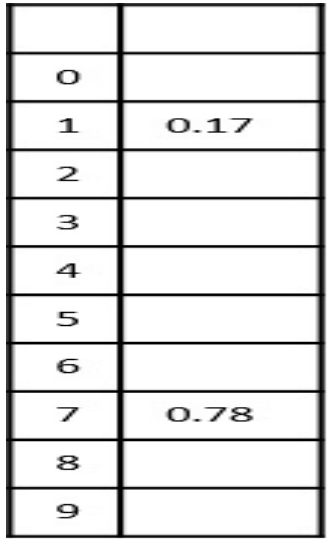
Linearly insert all the elements from the index 0 of the input array. That is, we insert 0.78 first followed by other elements sequentially. The position to insert the element is obtained using the

formula − $B[i]= \lfloor n.A[i]\rfloor$, i.e, $\lfloor 10\ 0.78\rfloor=7$



0	
1	
2	
3	
4	
5	
6	
7	0.78
8	
9	

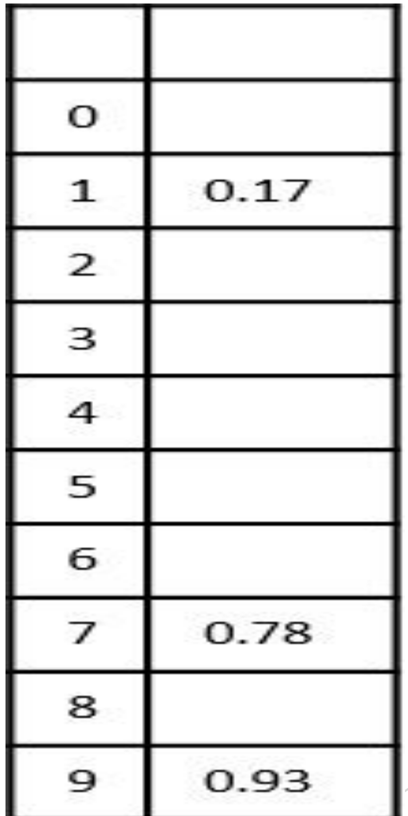
Now, we insert 0.17 at index ⌊10 0.17⌋=1



0	
1	0.17
2	
3	
4	
5	
6	
7	0.78
8	
9	

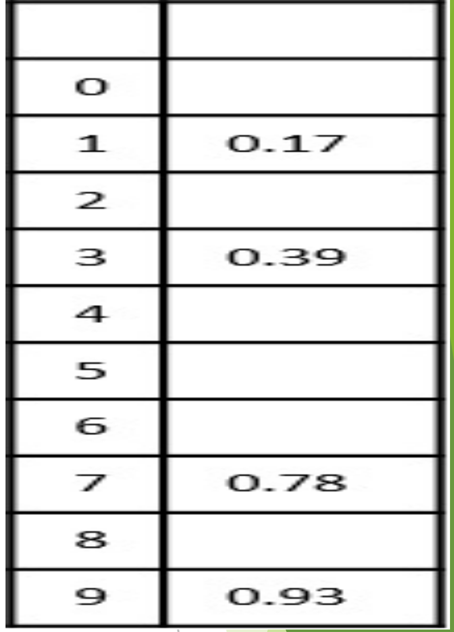
Step 3

Inserting the next element, 0.93 into the output buckets at ⌊10 0.93⌋=9



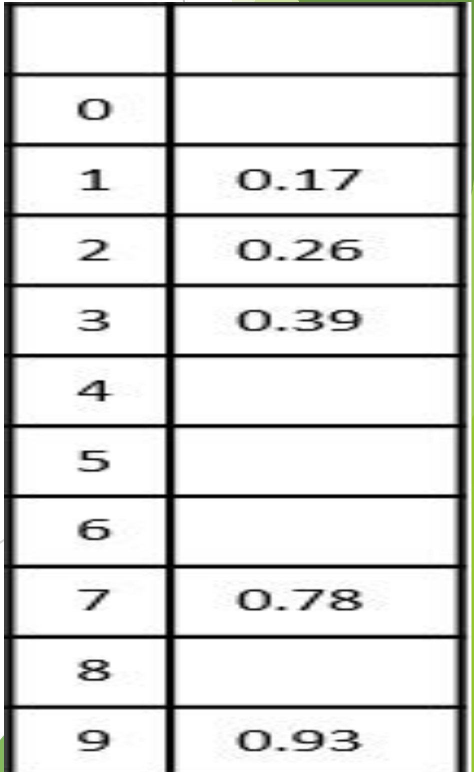
0	
1	0.17
2	
3	
4	
5	
6	
7	0.78
8	
9	0.93

Step 4 : Insert 0.39 at index 3 using the formula ⌊10 0.39⌋=3



0	
1	0.17
2	
3	0.39
4	
5	
6	
7	0.78
8	
9	0.93

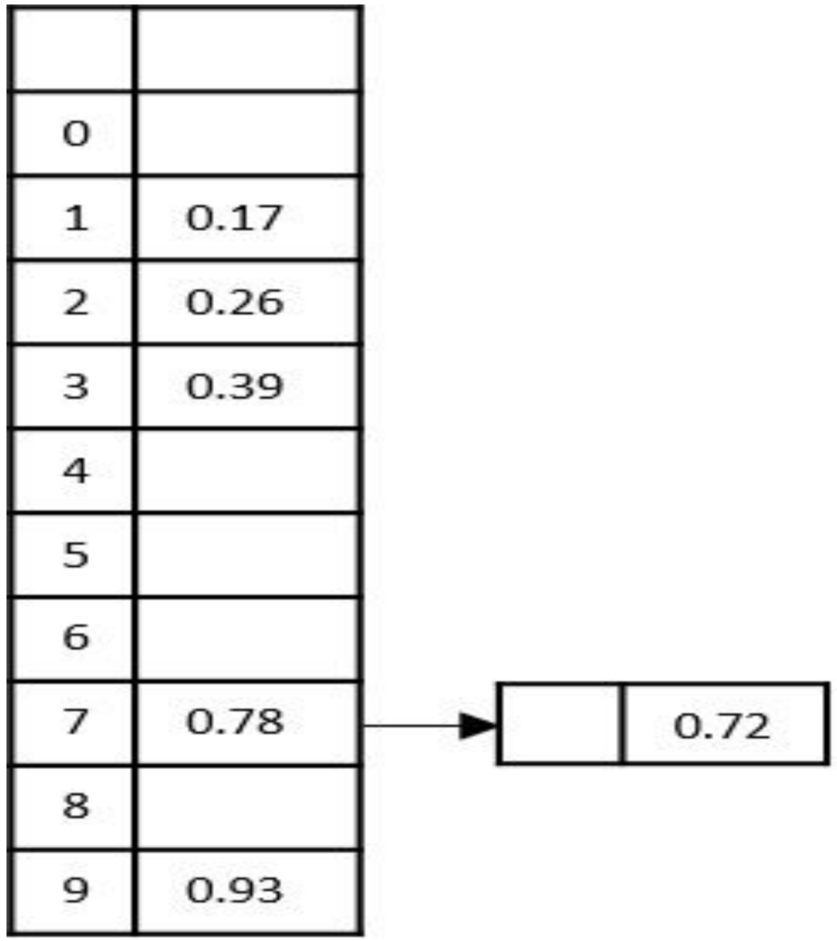
Step 5: Inserting the next element in the input array, 0.26, at position ⌊10 0.26⌋=2



0	
1	0.17
2	0.26
3	0.39
4	
5	
6	
7	0.78
8	
9	0.93

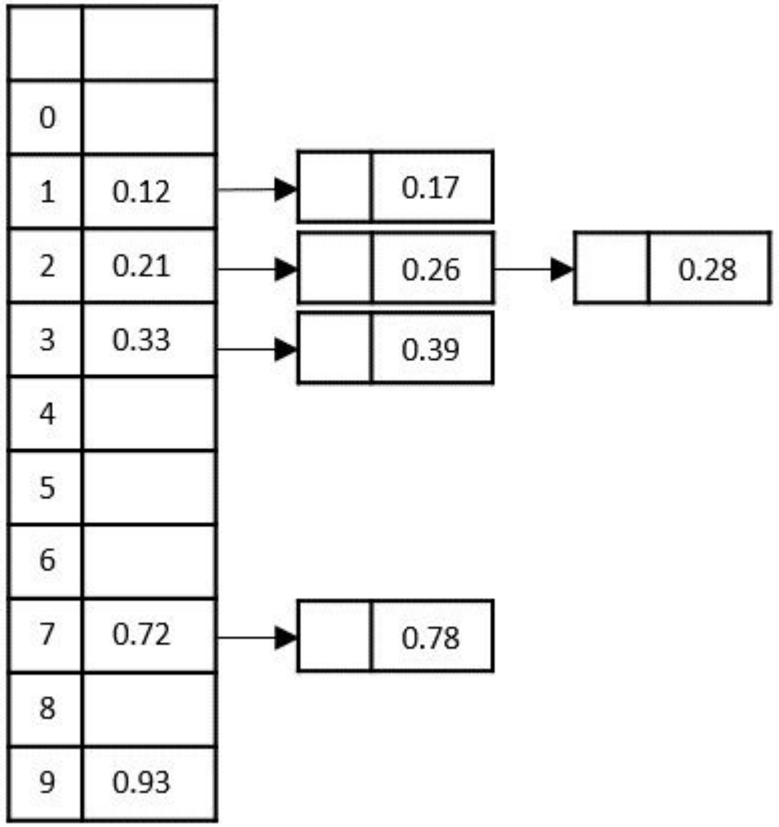
Step 6 :

Here is where it gets tricky. Now, the next element in the input list is 0.72 which needs to be inserted at index 7 using the formula ⌊10 0.72⌋=7. But there is already a number in the 7th bucket. So, a link is created from the 7th index to store the new number like a linked list, as shown below –



Step 7

Add the remaining numbers to the buckets in the similar manner by creating linked lists from the desired buckets. But while inserting these elements as lists, we apply insertion sort, i.e., compare the two elements and add the minimum value at the front as shown below −



Step 8

Now, to achieve the output, concatenate all the buckets together.

0.12, 0.17, 0.21, 0.26, 0.28, 0.33, 0.39, 0.72, 0.78, 0.93

```
#include <stdio.h>
void bucketsort(int a[], int n){ // function to implement bucket sort
  int max = a[0]; // get the maximum element in the array
  for (int i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];
  int b[max], i;
  for (int i = 0; i <= max; i++) {
    b[i] = 0;
  }
  for (int i = 0; i < n; i++) {
    b[a[i]]++;
  }
  for (int i = 0, j = 0; i <= max; i++) {
    while (b[i] > 0) {
      a[j++] = i;
      b[i]--;
    }  } }
```

```
int main(){
  int a[] = {12, 45, 33, 87, 56, 9, 11, 7, 67};
  int n = sizeof(a) / sizeof(a[0]); // n is the size of array
  printf("Before sorting array elements are: \n");
  for (int i = 0; i < n; ++i)
    printf("%d ", a[i]);
  bucketsort(a, n);
  printf("\nAfter sorting array elements are: \n");
  for (int i = 0; i < n; ++i)
    printf("%d ", a[i]);
}
```

Output :

```
Before sorting array elements are:
12 45 33 87 56 9 11 7 67
After sorting array elements are:
7 9 11 12 33 45 56 67 87
```

Selection sort algorithm :

Selection sort is a simple sorting algorithm. This sorting algorithm, like insertion sort, is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundaries by one element to the right.

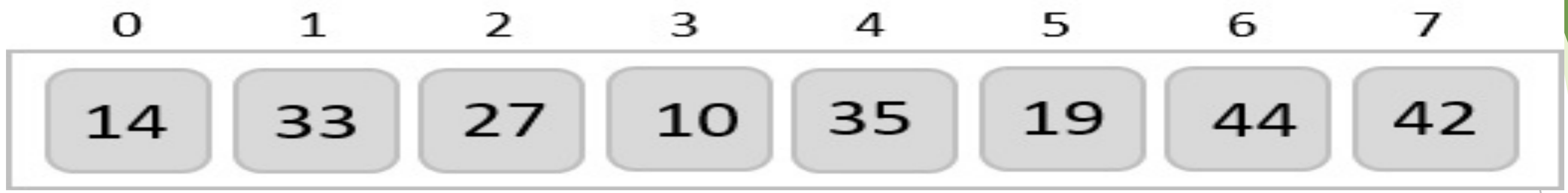
This type of sorting is called Selection Sort as it works by repeatedly sorting elements. That is: we first find the smallest value in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and we continue the process in this way until the entire array is sorted.

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

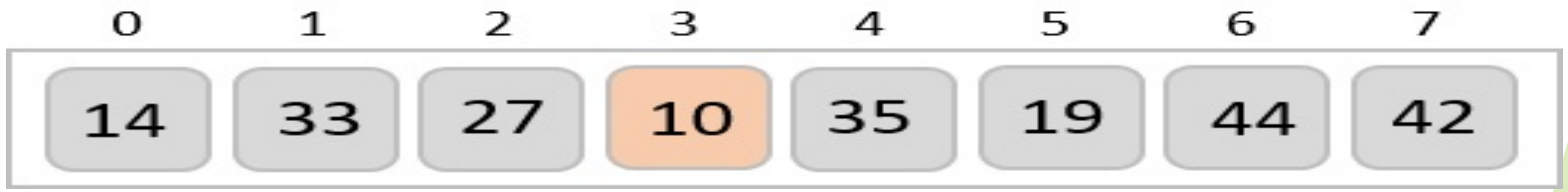
In the worst case, this could be quadratic, but in the average case, this quantity is **O(n log n)**. It implies that the **running time of Selection sort is quite insensitive to the input**.

Example

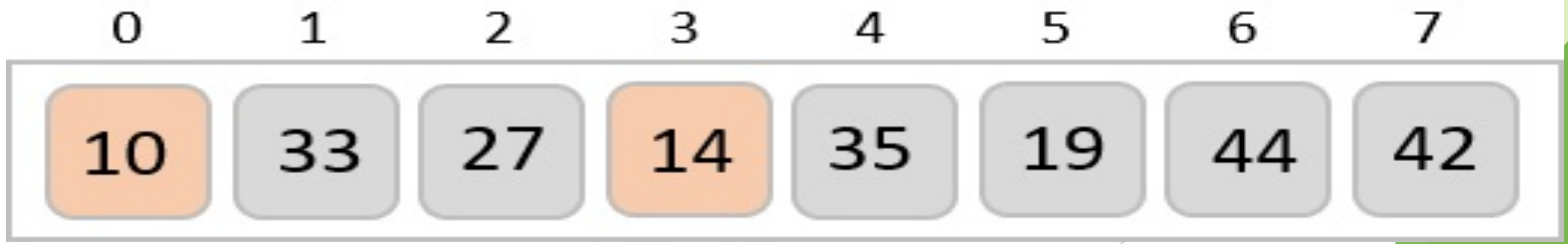
Consider the following depicted array as an example.



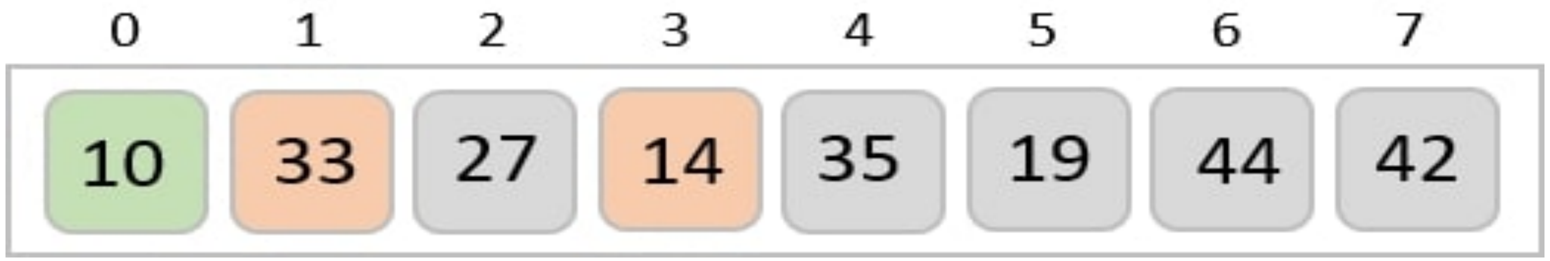
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



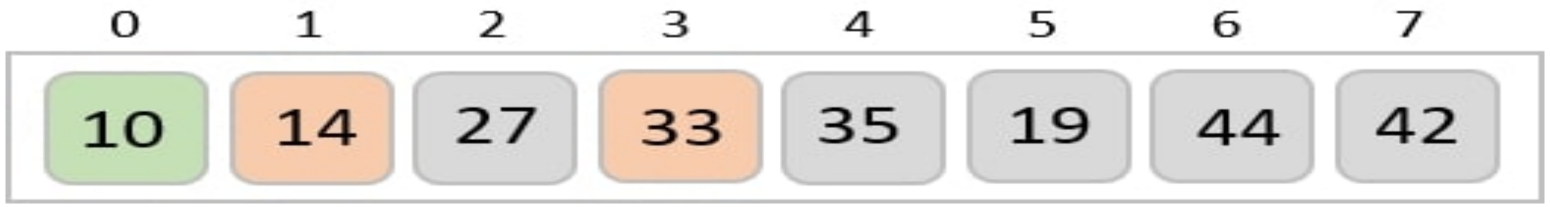
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



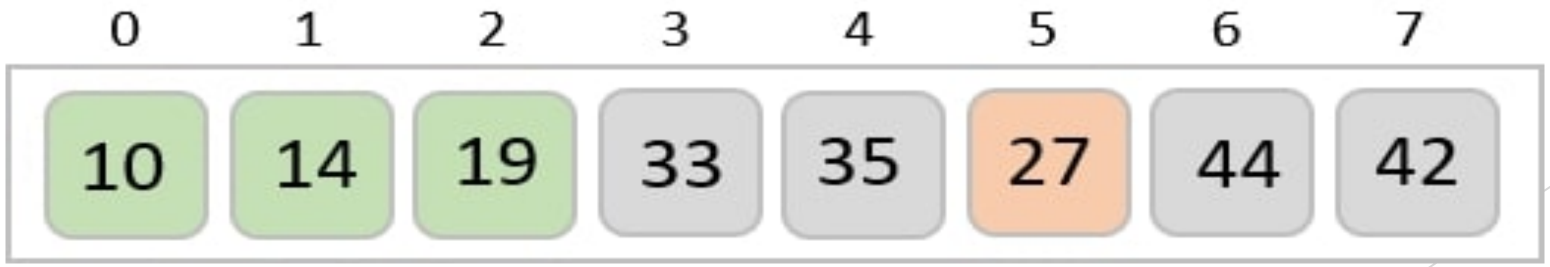
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

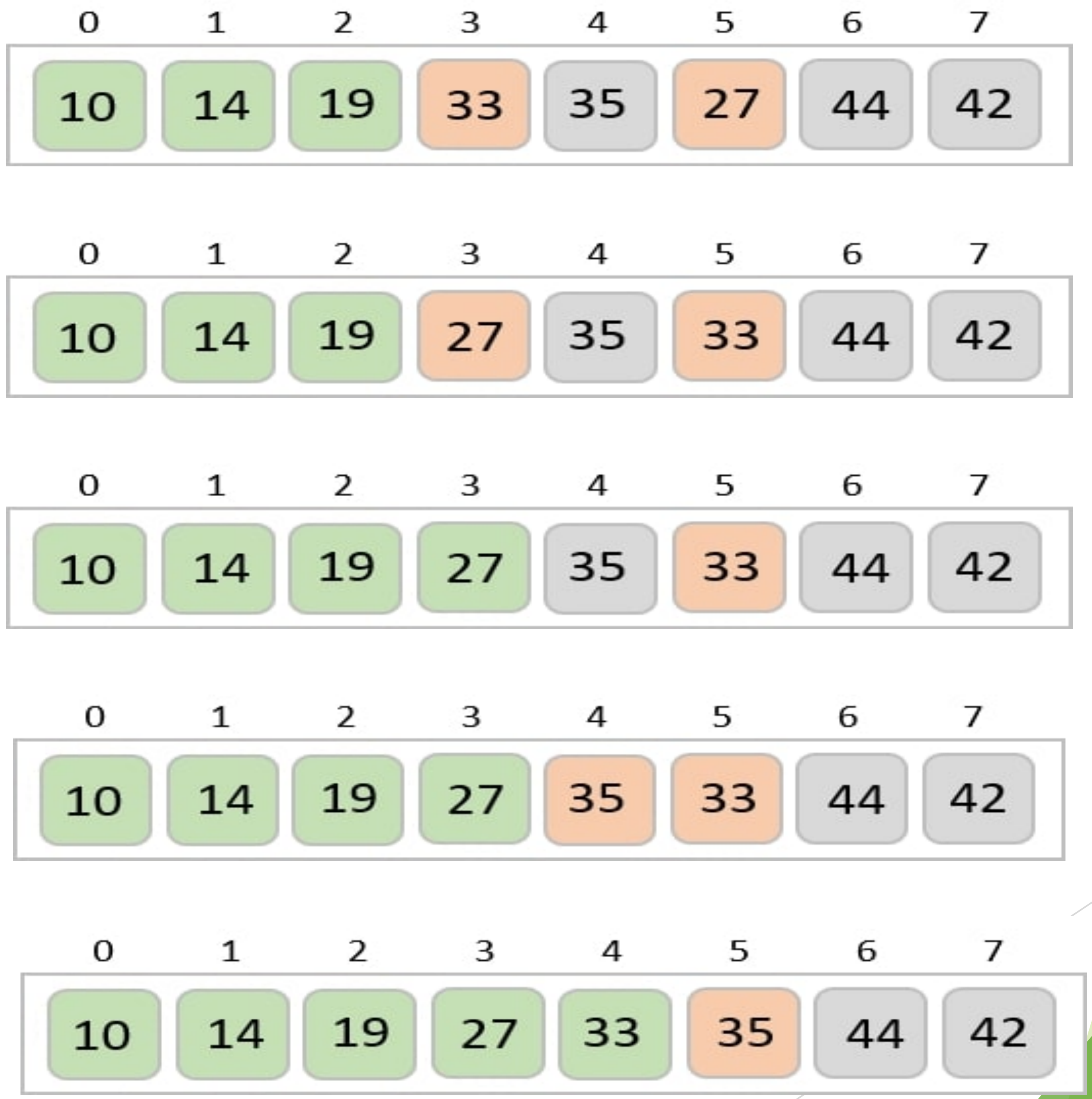


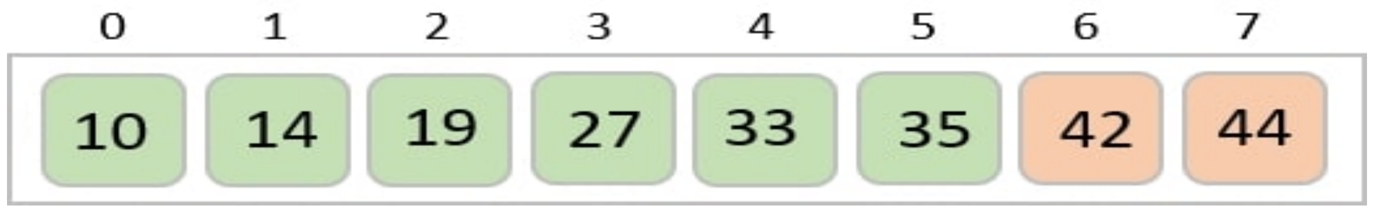
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

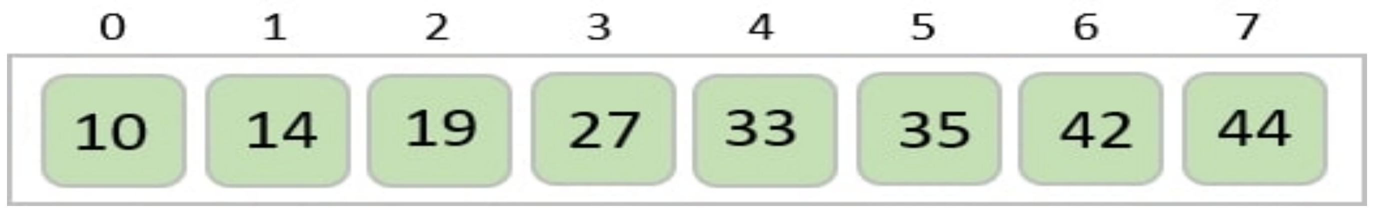


After two iterations, two least values are positioned at the beginning in a sorted manner.




0 1 2 3 4 5 6 7
10 14 19 33 35 27 44 42
0 1 2 3 4 5 6 7
10 14 19 27 35 33 44 42
0 1 2 3 4 5 6 7
10 14 19 27 35 33 44 42
0 1 2 3 4 5 6 7
10 14 19 27 35 33 44 42
0 1 2 3 4 5 6 7
10 14 19 27 33 35 44 42


0 1 2 3 4 5 6 7
10 14 19 27 33 35 42 44


0 1 2 3 4 5 6 7
10 14 19 27 33 35 42 44

```
#include <stdio.h>
void selectionSort(int array[], int size){
  int i, j, imin;
  for(i = 0; i<size-1; i++) {
    imin = i; //get index of minimum data
    for(j = i+1; j<size; j++)
      if(array[j] < array[imin])
        imin = j;

    //placing in correct position
    int temp;
    temp = array[i];
    array[i] = array[imin];
    array[imin] = temp;
  }
}
```

```
int main(){
   int n;
   n = 5;
   int arr[5] = {12, 19, 55, 2, 16}; // initialize the array
   printf("Array before Sorting: ");
   for(int i = 0; i<n; i++)
      printf("%d ",arr[i]);
   printf("\n");
   selectionSort(arr, n);
   printf("Array after Sorting: ");
   for(int i = 0; i<n; i++)
      printf("%d ", arr[i]);
   printf("\n");
}
```

Output

```
Array before Sorting: 12 19 55 2 16
Array after Sorting: 2 12 16 19 55
```

Bubble Sort Algorithm :

Bubble Sort is an elementary sorting algorithm, which works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.

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

Step 1 – Check if the first element in the input array is greater than the next element in the array.

Step 2 – If it is greater, swap the two elements; otherwise move the pointer forward in the array.

Step 3 – Repeat Step 2 until we reach the end of the array.

Step 4 – Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.

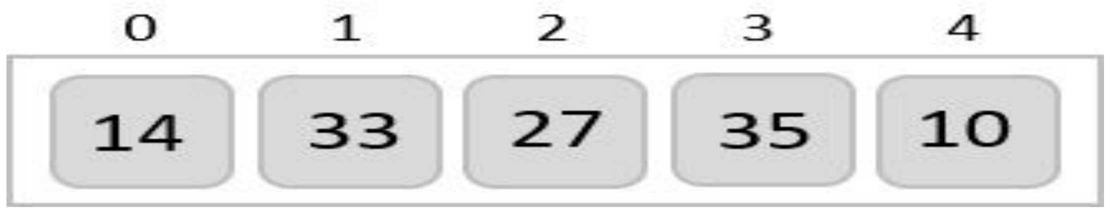
Step 5 – The final output achieved is the sorted array.

Pseudocode of bubble sort algorithm can be written as follows –

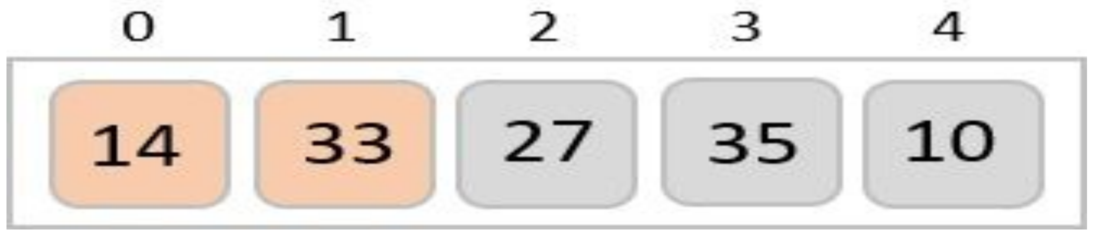
```
voidbubbleSort(int numbers[], intarray_size){
  inti, j, temp;
  for (i = (array_size - 1); i>= 0; i--)
  for (j = 1; j <= i; j++)
  if (numbers[j-1] > numbers[j]){
    temp = numbers[j-1];
    numbers[j-1] = numbers[j];
    numbers[j] = temp;
  }
}
```

Example

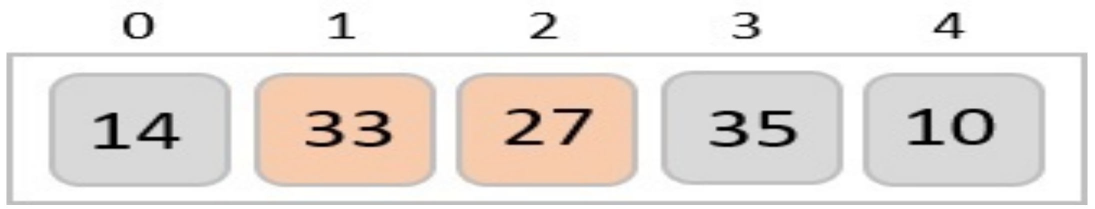
We take an unsorted array for our example. Bubble sort takes (n^2) time so we're keeping it short and precise.



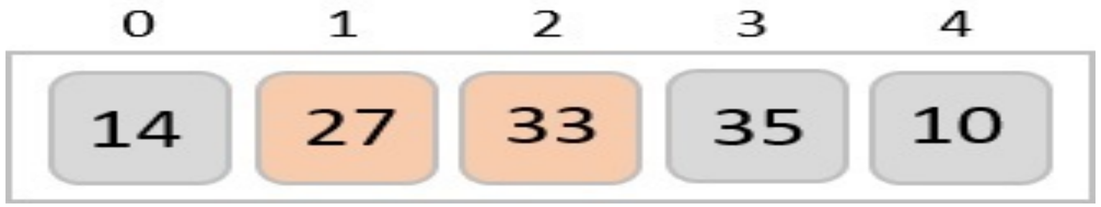
Bubble sort starts with very first two elements, comparing them to check which one is greater.



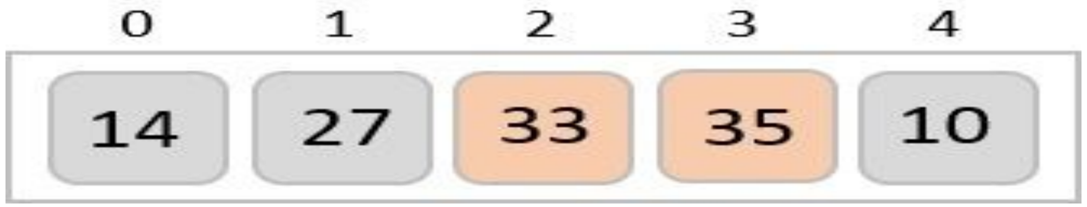
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



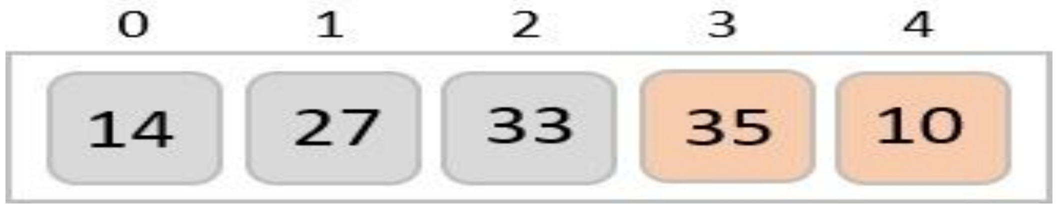
We find that 27 is smaller than 33 and these two values must be swapped.



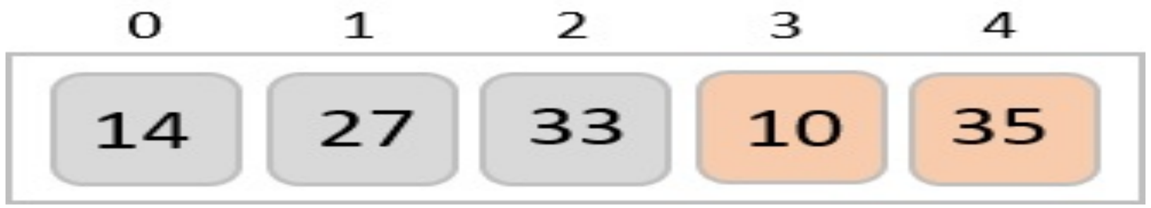
Next we compare 33 and 35. We find that both are in already sorted positions.



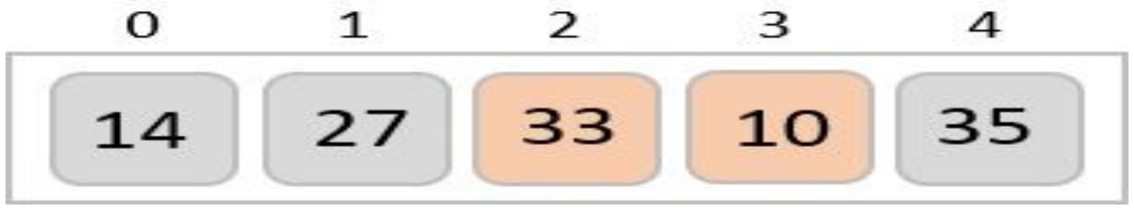
Then we move to the next two values, 35 and 10.

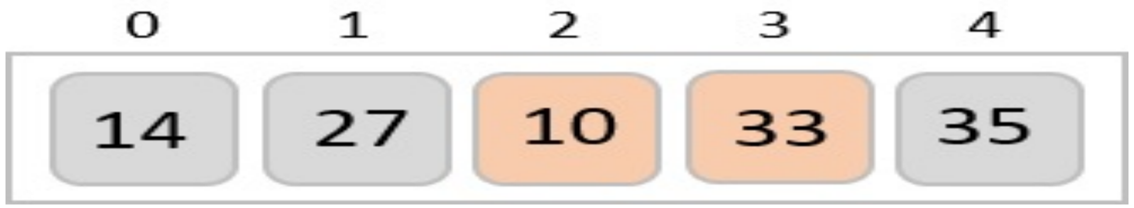


We know then that 10 is smaller 35. Hence they are not sorted. We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

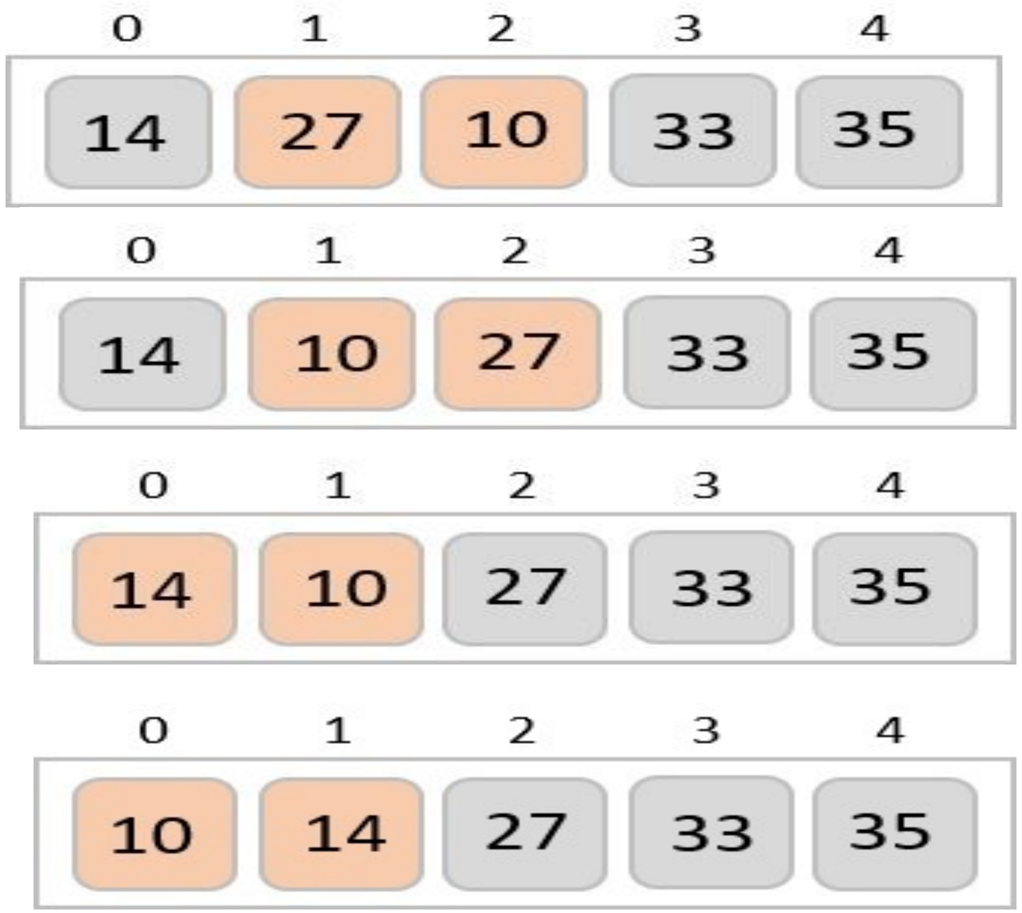


To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

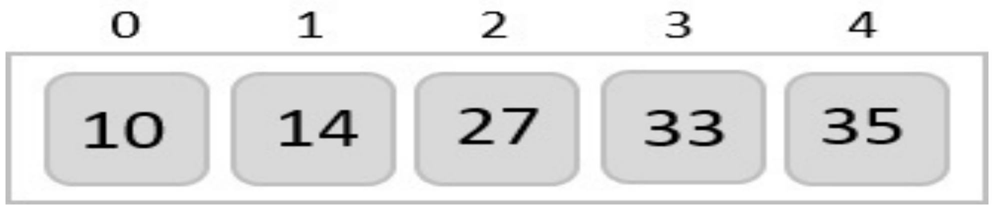




Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sort learns that an array is completely sorted.



```
#include<iostream>
using namespace std;
void bubbleSort(int *array, int size){
   for(int i = 0; i<size; i++) {
      int swaps = 0; //flag to detect any swap is there or not
      for(int j = 0; j<size-i-1; j++) {
         if(array[j] > array[j+1]) { //when the current item is bigger than next
            int temp;
            temp = array[j];
            array[j] = array[j+1];
            array[j+1] = temp;
            swaps = 1; //set swap flag
         }
      }
      if(!swaps)
         break; // No swap in this pass, so array is sorted
   }
}
```

```
int main(){
    int n;
    n = 5;
    int arr[5] = {67, 44, 82, 17, 20}; //initialize an array
    cout << "Array before Sorting: ";
    for(int i = 0; i<n; i++)
        cout << arr[i] << " ";
    cout << endl;
    bubbleSort(arr, n);
    cout << "Array after Sorting: ";
    for(int i = 0; i<n; i++)
        cout << arr[i] << " ";
    cout << endl;
}
```

Output

```
Array before Sorting: 67 44 82 17 20
Array after Sorting: 17 20 44 67 82
```

Linear Search Algorithm :

Linear search is a type of sequential searching algorithm. In this method, every element within the input array is traversed and compared with the key element to be found. If a match is found in the array the search is said to be successful; if there is no match found the search is said to be unsuccessful and gives the worst-case time complexity.

Linear Search Algorithm :

The algorithm for linear search is relatively simple. The procedure starts at the very first index of the input array to be searched.

Step 1 – Start from the 0th index of the input array, compare the key value with the value present in the 0th index.

Step 2 – If the value matches with the key, return the position at which the value was found.

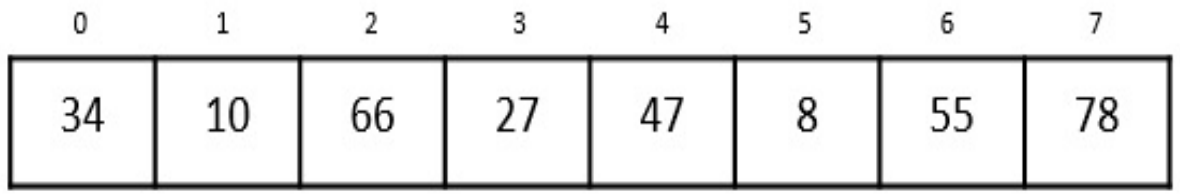
Step 3 – If the value does not match with the key, compare the next element in the array.

Step 4 – Repeat Step 3 until there is a match found. Return the position at which the match was found.

Step 5 – If it is an unsuccessful search, print that the element is not present in the array and exit the program.

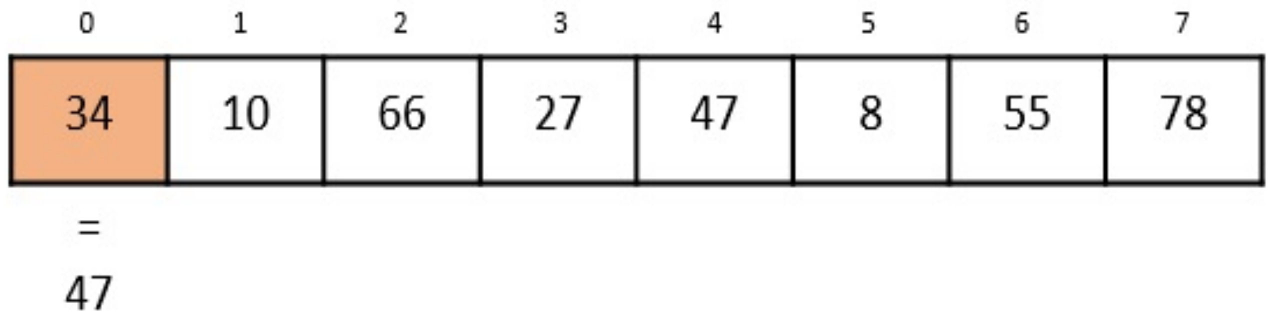
Example

Let us look at the step-by-step searching of the key element (say 47) in an array using the linear search method.



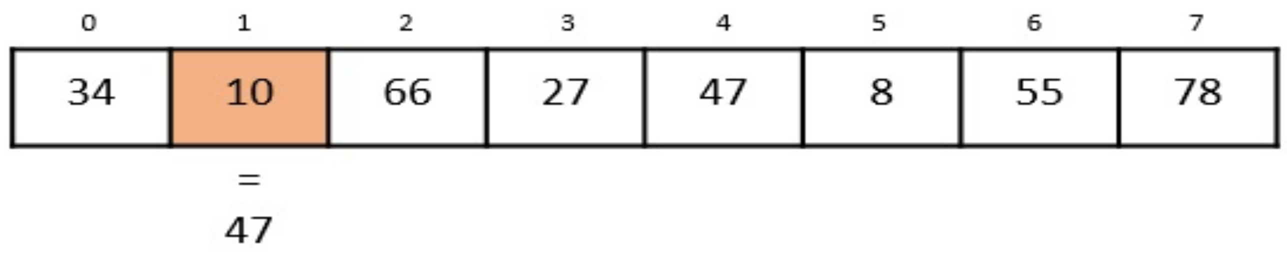
Step 1

The linear search starts from the 0th index. Compare the key element with the value in the 0th index, 34.



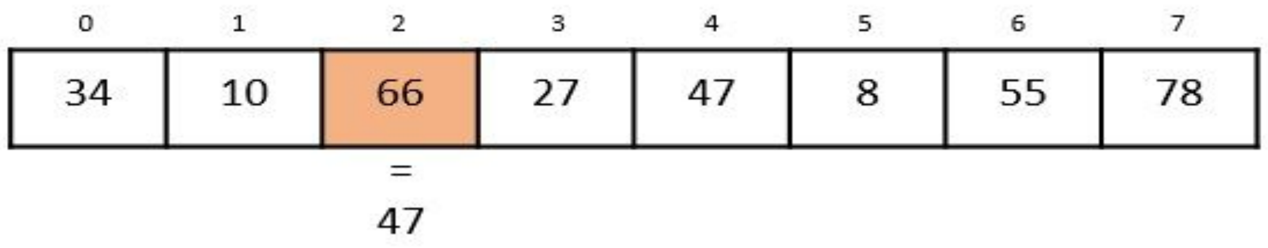
However, 47 34. So it moves to the next element.

Step 2 : Now, the key is compared with value in the 1st index of the array.



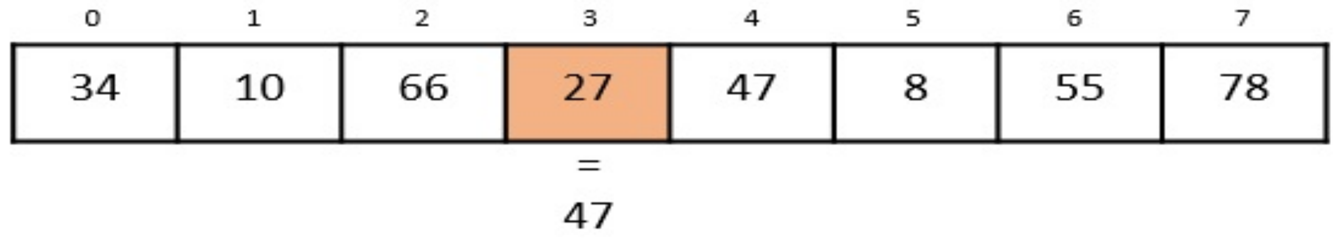
Still, 47 10, making the algorithm move for another iteration. Still, 47 10, making the algorithm move for another iteration.

Step 3 : The next element 66 is compared with 47. They are both not a match so the algorithm compares the further elements.



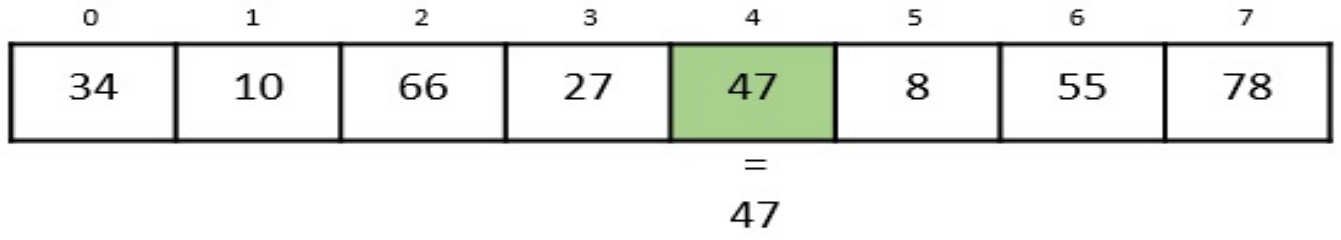
Step 4

Now the element in 3rd index, 27, is compared with the key value, 47. They are not equal so the algorithm is pushed forward to check the next element.



Step 5

Comparing the element in the 4th index of the array, 47, to the key 47. It is figured that both the elements match. Now, the position in which 47 is present, i.e., 4 is returned.

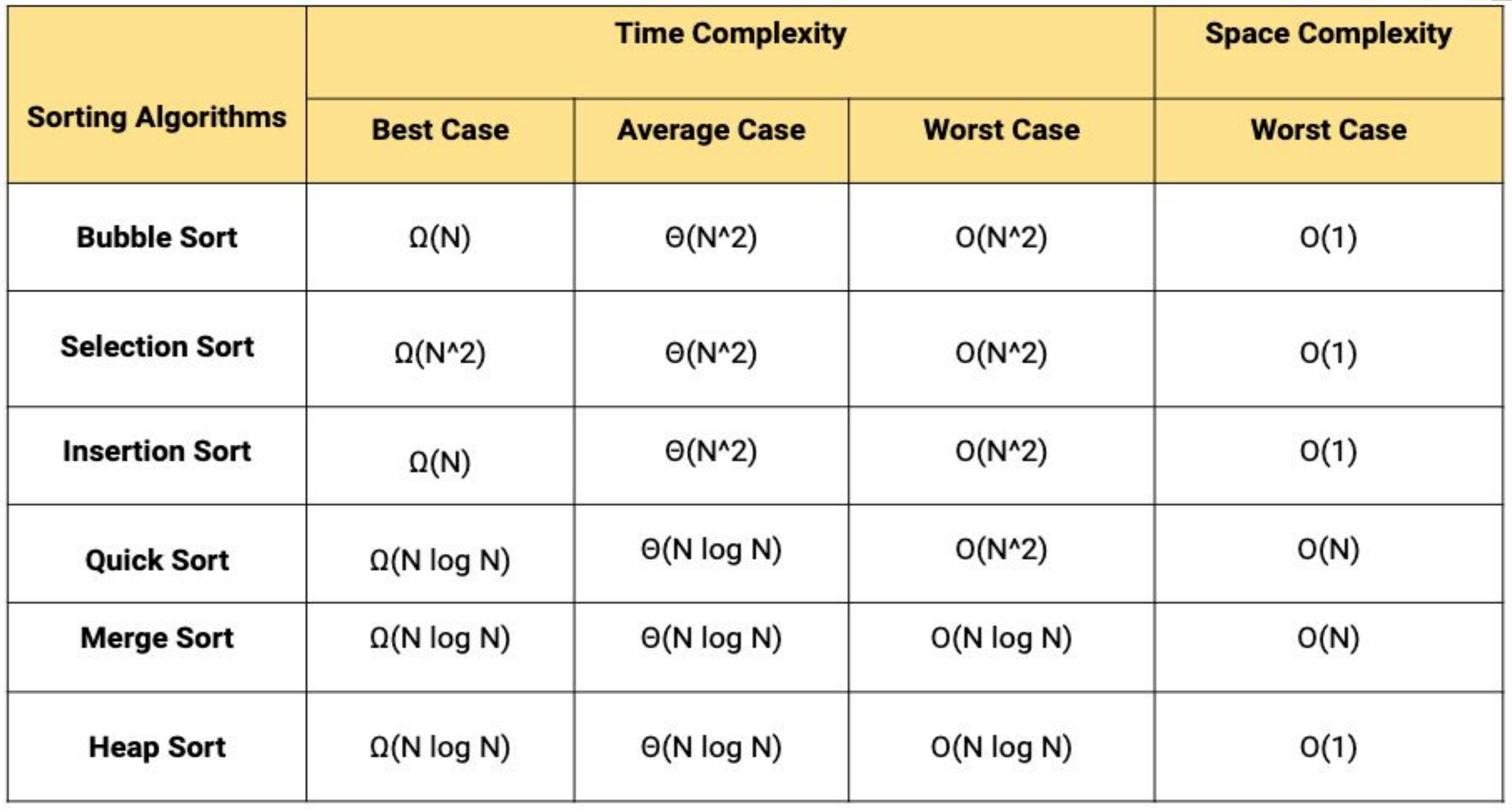


Comparison based sorting algorithms

In a comparison based sorting algorithms, we compare elements of an array with each other to determines which of two elements should occur first in the final sorted list.

All comparison-based sorting algorithms have a complexity lower bound of **nlogn. (Think!)**

Here is the comparison of time and space complexities of some popular comparison based sorting algorithms:

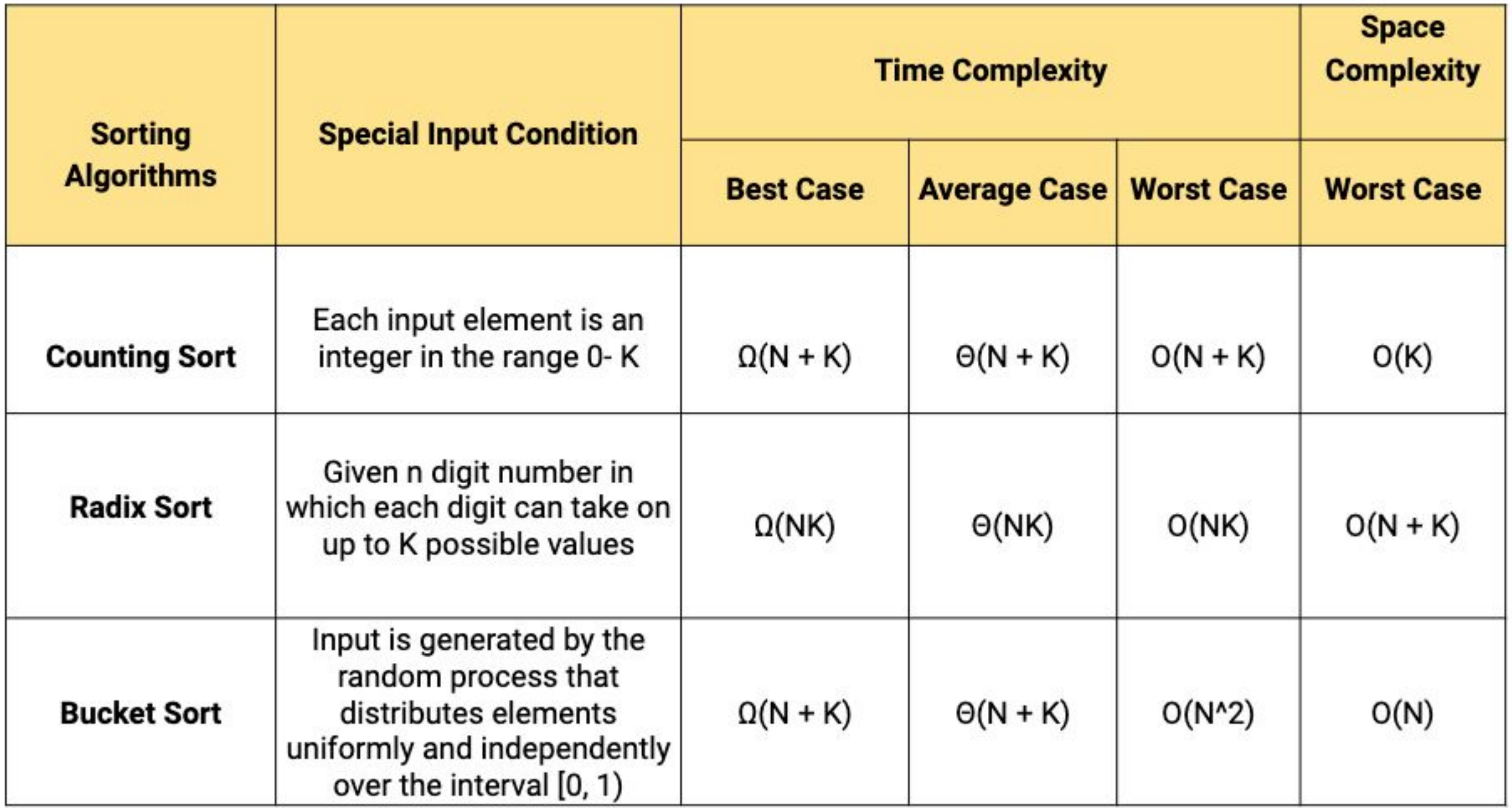


Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)

Non-comparison based sorting algorithm

There are sorting algorithms that use special information about the keys and operations other than comparison to determine the sorted order of elements.

Consequently, **nlogn** lower bound does not apply to these sorting algorithms.



Sorting Algorithms	Special Input Condition	Time Complexity			Space Complexity
		Best Case	Average Case	Worst Case	Worst Case
Counting Sort	Each input element is an integer in the range 0- K	Ω(N + K)	Θ(N + K)	O(N + K)	O(K)
Radix Sort	Given n digit number in which each digit can take on up to K possible values	Ω(NK)	Θ(NK)	O(NK)	O(N + K)
Bucket Sort	Input is generated by the random process that distributes elements uniformly and independently over the interval [0, 1)	Ω(N + K)	Θ(N + K)	O(N^2)	O(N)

END