# UNIT- 4 Searching and Sorting

->Searching

Interpolation Search

->Sorting

Selection Sort

Insertion Sort

Bubble Sort

Quick Sort

Merge Sort

Radix Sort

## SEARCHING

*Definition*:

Searching technique refers to finding a key element among the list of elements. If the given element is present in the list, then the searching process is said to be successful. If the given element is not present in the list, then the searching process is said to be unsuccessful.

**INTERPOLATION SEARCH**

*Definition:*

- Interpolation search is an algorithm for searching for a key in a sorted array.
- It improves upon binary search by making a guess of where the element might be based on its value and the range of elements in the array.
- It uses the idea of the interpolation formula to estimate the probable location of the target element.
- It calculates the probable position using an interpolation formula that considers the range and values of the data elements.

**Interpolation Formula =**

**low + [(high - low) * (X - A[low])]/(A[high] - A[low])]**

*low - Left pointer*

*high - Right pointer*

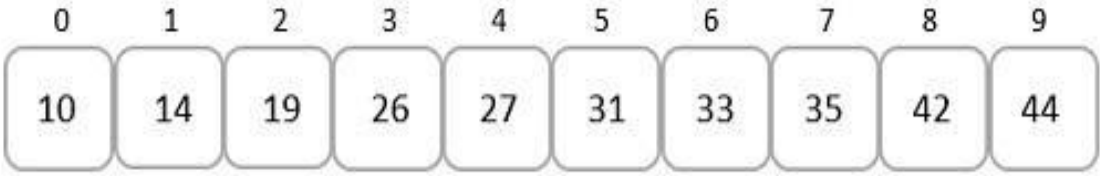
*A[low] - Element at the left pointer*

*A[high] - Element at the right pointer*

*X - Target element to be searched*

*Working of Interpolation Search:*

*Example:*




Let us search for the element 19.

*Solution*

- Unlike binary search, the middle point in this approach is chosen using the formula − **mid=Lo+(Hi−Lo)∗(X−A[Lo])/A[Hi]−A[Lo]**

So in this given array input,

Lo = 0, A[Lo] = 10

Hi = 9, A[Hi] = 44

X = 19

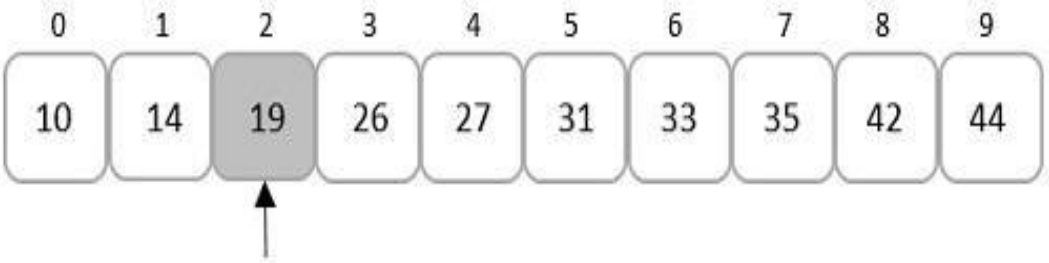
- Applying the formula to find the middle point in the list, we get

mid=0+(9−0)∗(19−10)/44−10

mid=9∗9/34

mid=81/34=2.38

- Since, mid is an index value, we only consider the integer part of the decimal. That is, mid = 2.




- Comparing the key element given, that is 19, to the element present in the mid index, it is found that both the elements match.
- Therefore, the element is found at index 2.

**Interpolation Search Algorithm:**

```
int interpolationSearch(int arr[], int n, int x) {
   int low = 0, high = n - 1;
   while (low <= high && x >= arr[low] && x <= arr[high]) {
      // Estimate the position
      int pos = low + (((double)(high - low) / (arr[high] - arr[low])) * (x - arr[low]));
      // If the element is found
```

```
        if (arr[pos] == x)
            return pos;
        // If x is larger, ignore left half
        if (arr[pos] < x)
            low = pos + 1;
        // If x is smaller, ignore right half
        else
            high = pos - 1;
    }
    return -1; // If element is not found
}
```

## SORTING

- Sorting is the process of arranging data into meaningful order so that you can analyze it more effectively.
- In this we have 5 types of sorting techniques they are
    1. Selection Sort
    2. Bubble Sort
    3. Merge Sort
    4. Quick Sort
    5. Radix Sort

## BUBBLE SORT

*Definition:*

Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are in the intended order.

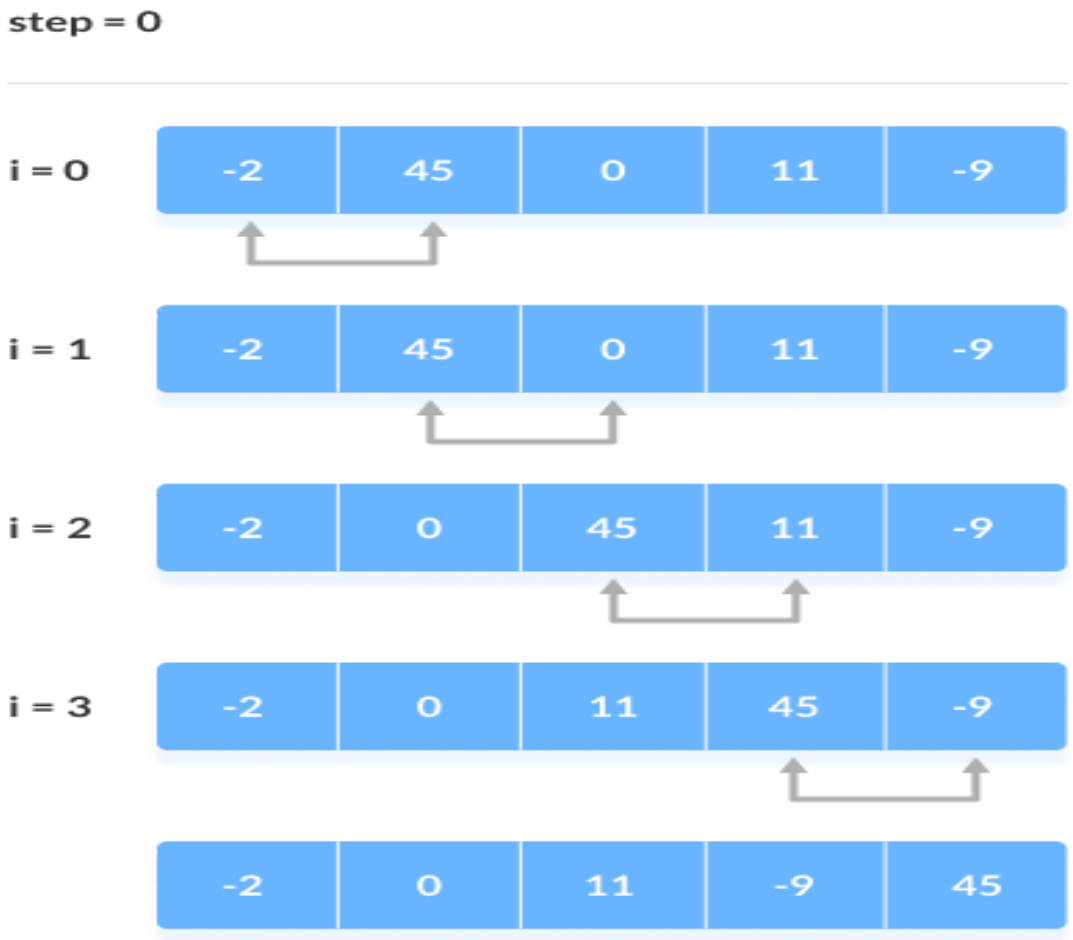
*Working of Bubble Sort:*

Suppose we are trying to sort the elements in ascending order.

*1.First Iteration (Compare and Swap):*

- Starting from the first index, compare the first and the second elements.
- If the first element is greater than the second element, they are swapped.
- Now, compare the second and the third elements. Swap them if they are not in order.

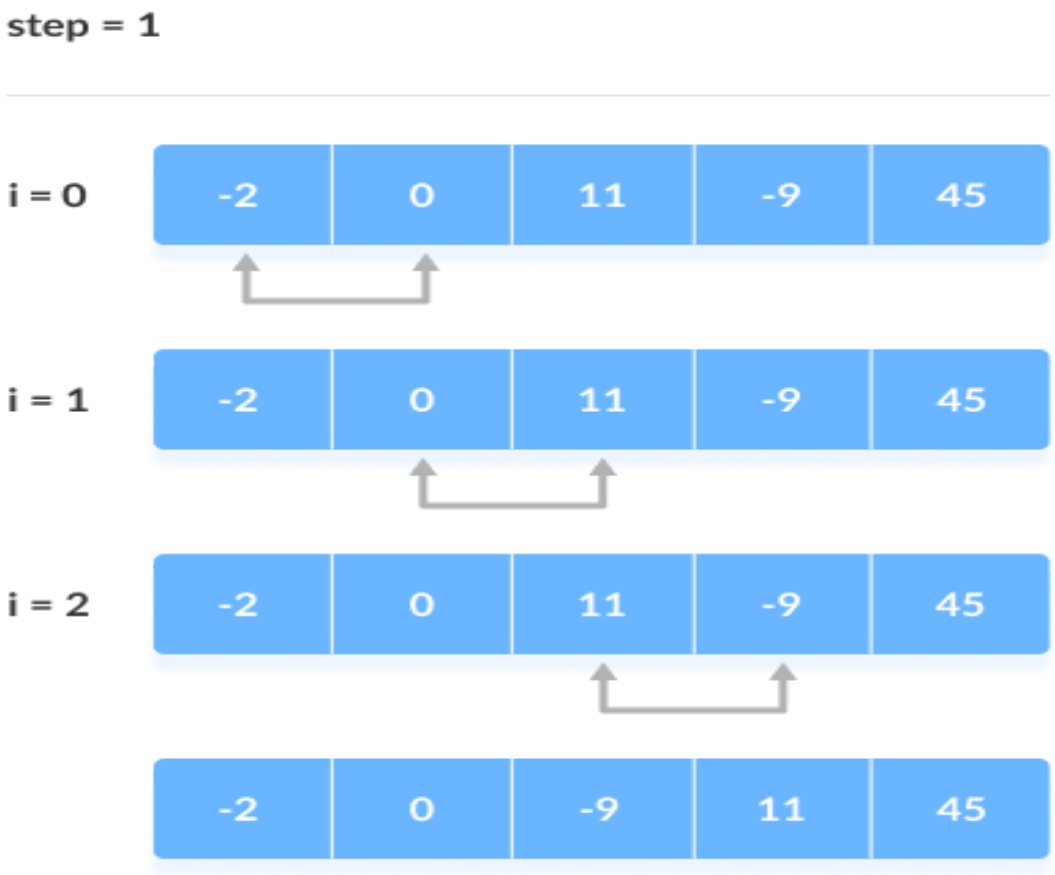
- The above process goes on until the last element.




Compare the Adjacent Elements

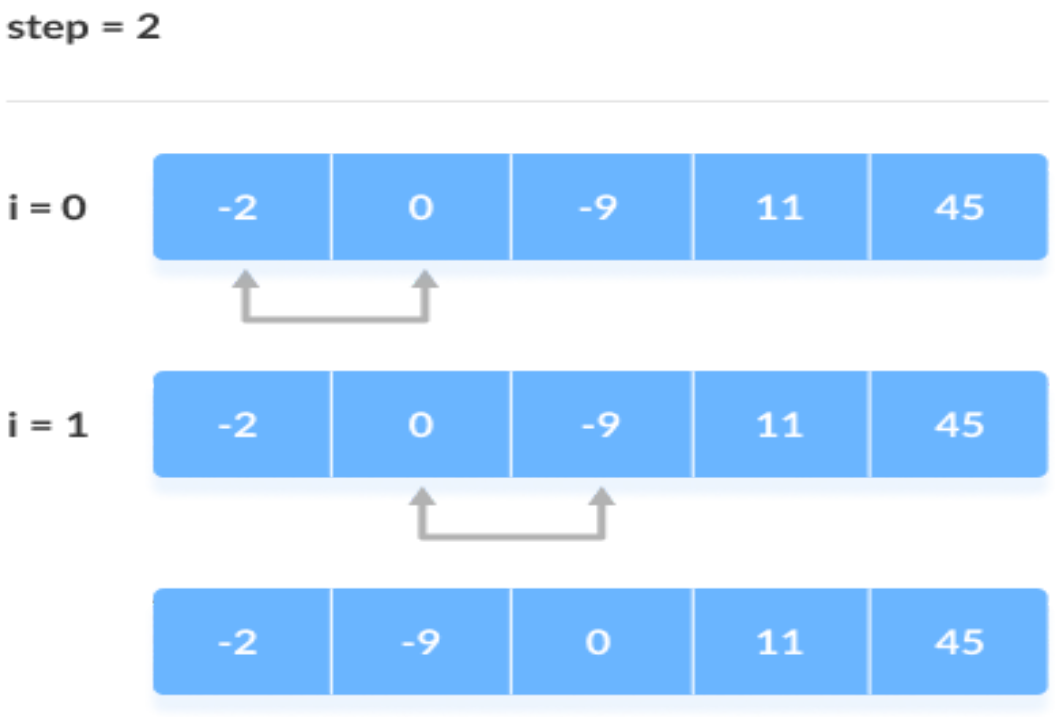
*2. Remaining Iteration:*

- The same process goes on for the remaining iterations.
- After each iteration, the largest element among the unsorted elements is placed at the end.




Put the largest element at the end

*3 .In each iteration, the comparison takes place up to the last unsorted element.*




Compare the adjacent elements

*4. The array is sorted when all the unsorted elements are placed at their correct positions.*


The array is sorted if all elements are kept in the right order.

**Bubble Sort Algorithm:**

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

**Bubble Sort Complexity**

*Time Complexity:*

Best - O(n)

Worst - O(n2)

Average - O(n2)

Space Complexity - O(1)

Stability - Yes

*Advantages of Bubble Sort:*

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

*Disadvantages of Bubble Sort:*

- Bubble sort has a time complexity of O(N2) which makes it very slow for large data sets.
- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

## Selection Sort

*Definition:*

Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

*Working of Selection Sort:*

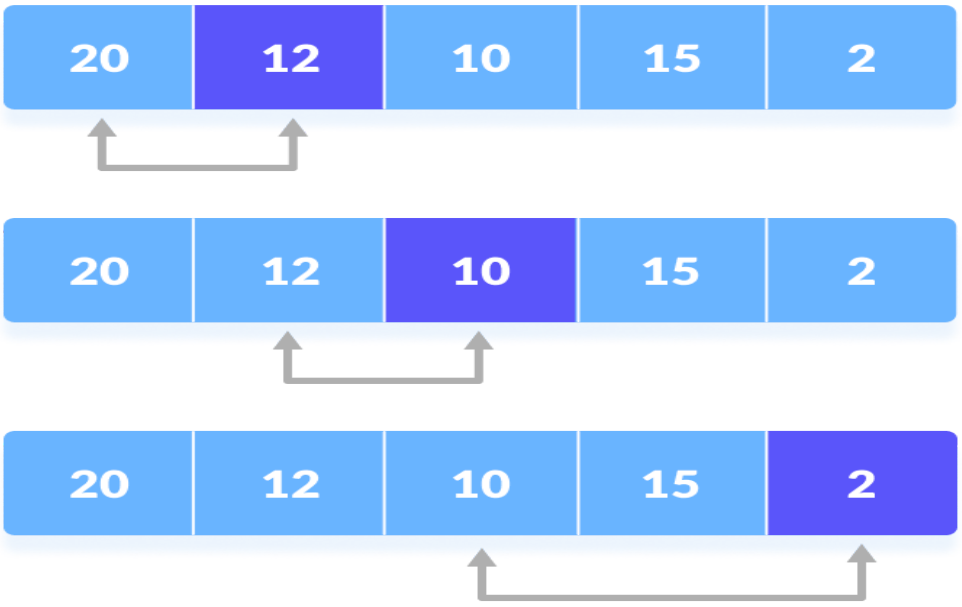
*1. Set the first element as minimum.*


Select first element as minimum

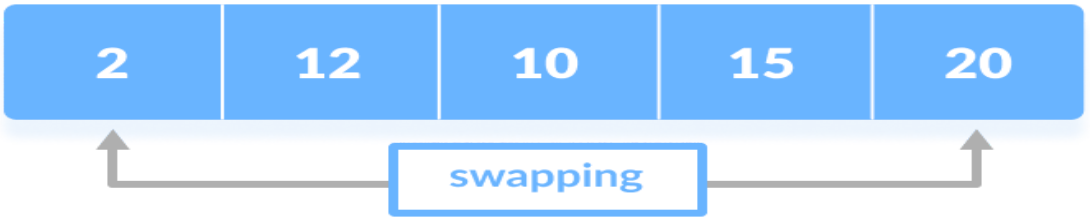
*2.Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.*

- Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.

Compare minimum with the remaining elements

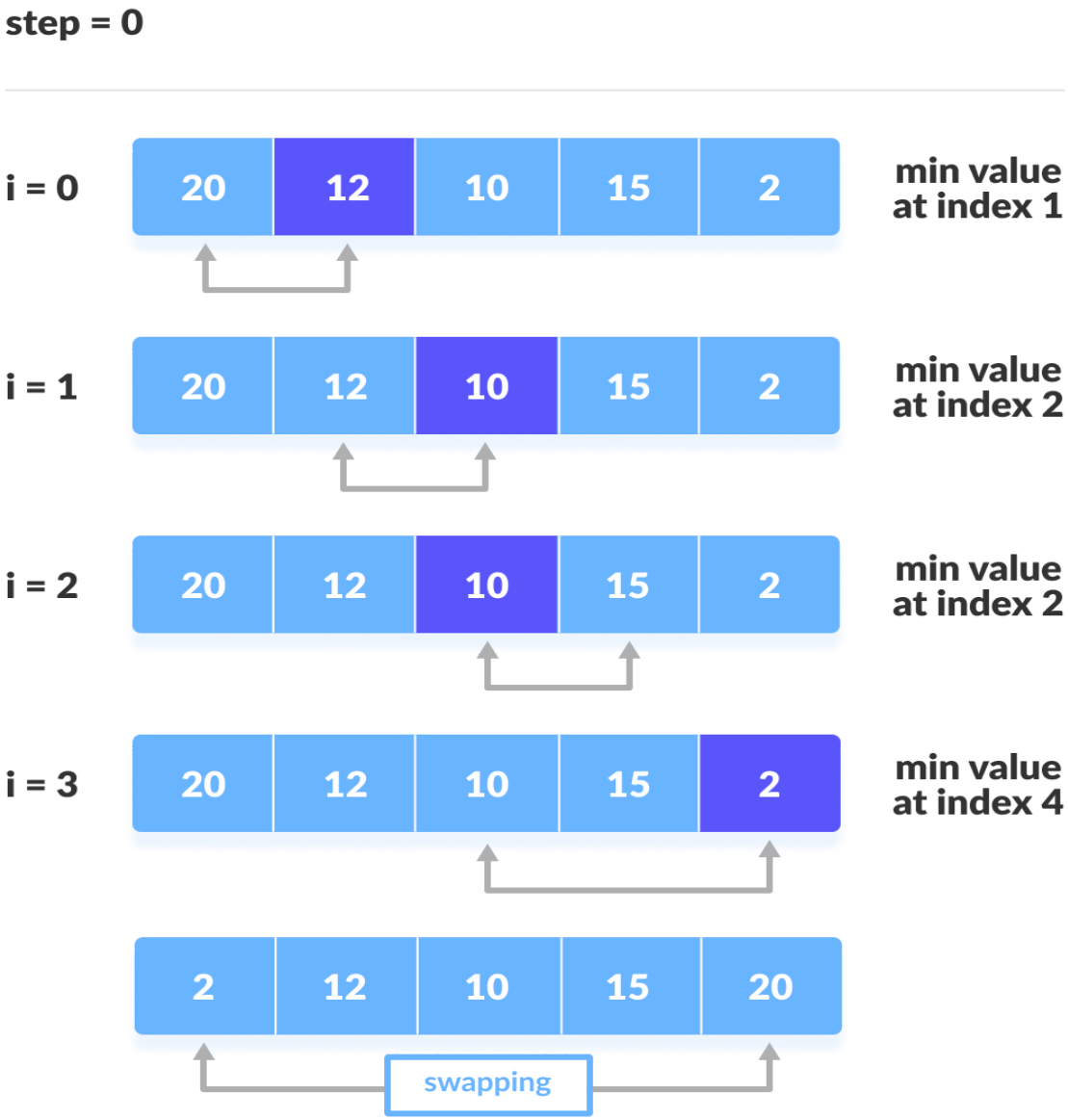



*3. After each iteration, minimum is placed in the front of the unsorted list*

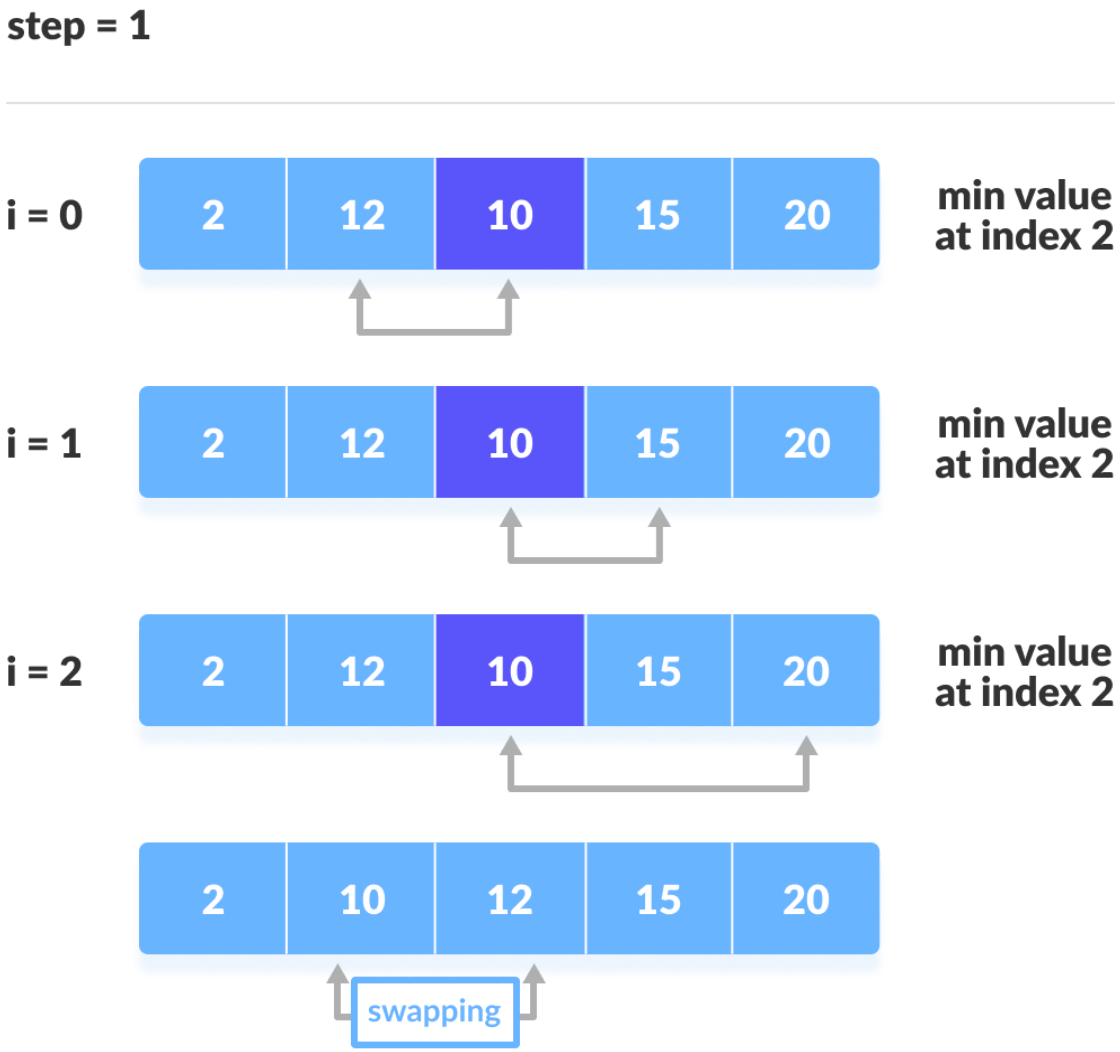



Swap the first with minimum

*4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions*.

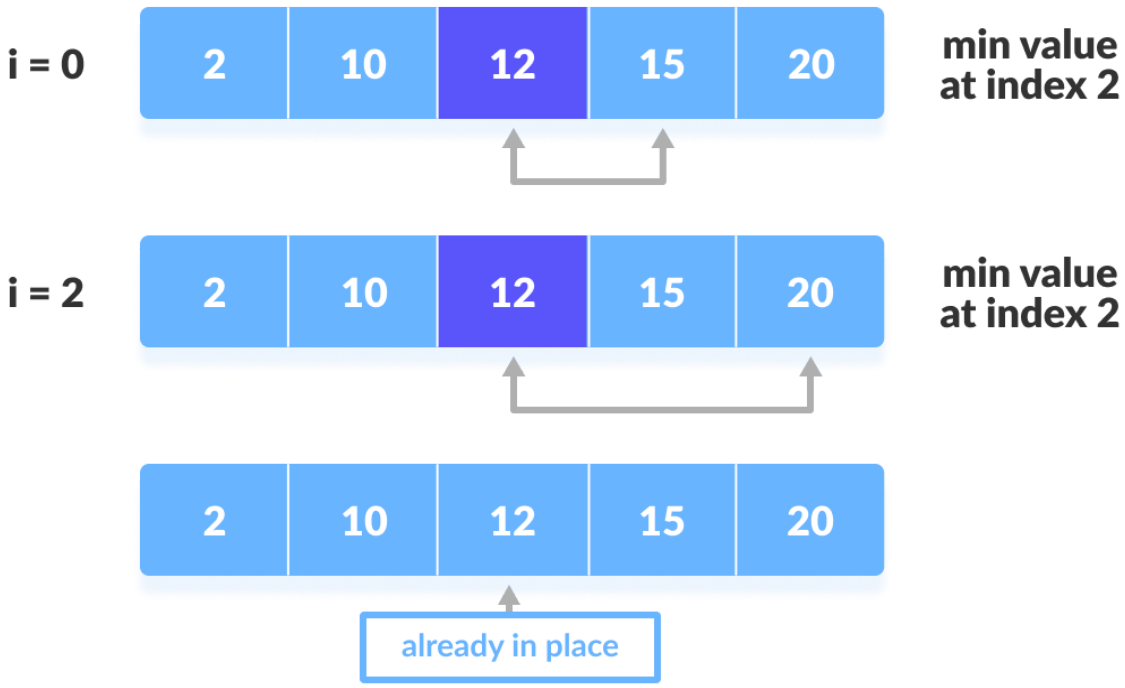



The first iteration



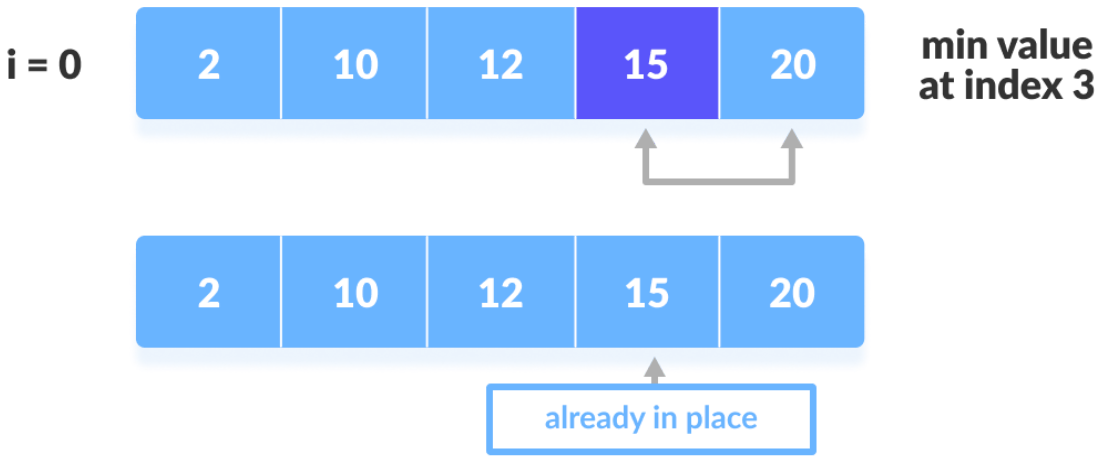

The second iteration

step = 2




The third iteration

step = 3



The fourth iteration

**Selection Sort Algorithm:**

```
for (i = 0; i < n - 1; i++) {
min_idx = i;
for (j = i + 1; j < n; j++){
if (arr[j] < arr[min_idx]){
min_idx = j;
}
Int temp = a[i];
a[i]=a[min];
a[min]=temp;
}
}
```

**Selection Sort Complexity**

*Time Complexity:*

Best - O(n2)

Worst - O(n2)

Average - O(n2)

Space Complexity - O(1)

Stability - No

## INSERTION SORT

*Definition:*

- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.
- Insertion sort works similarly as we sort cards in our hand in a card game.

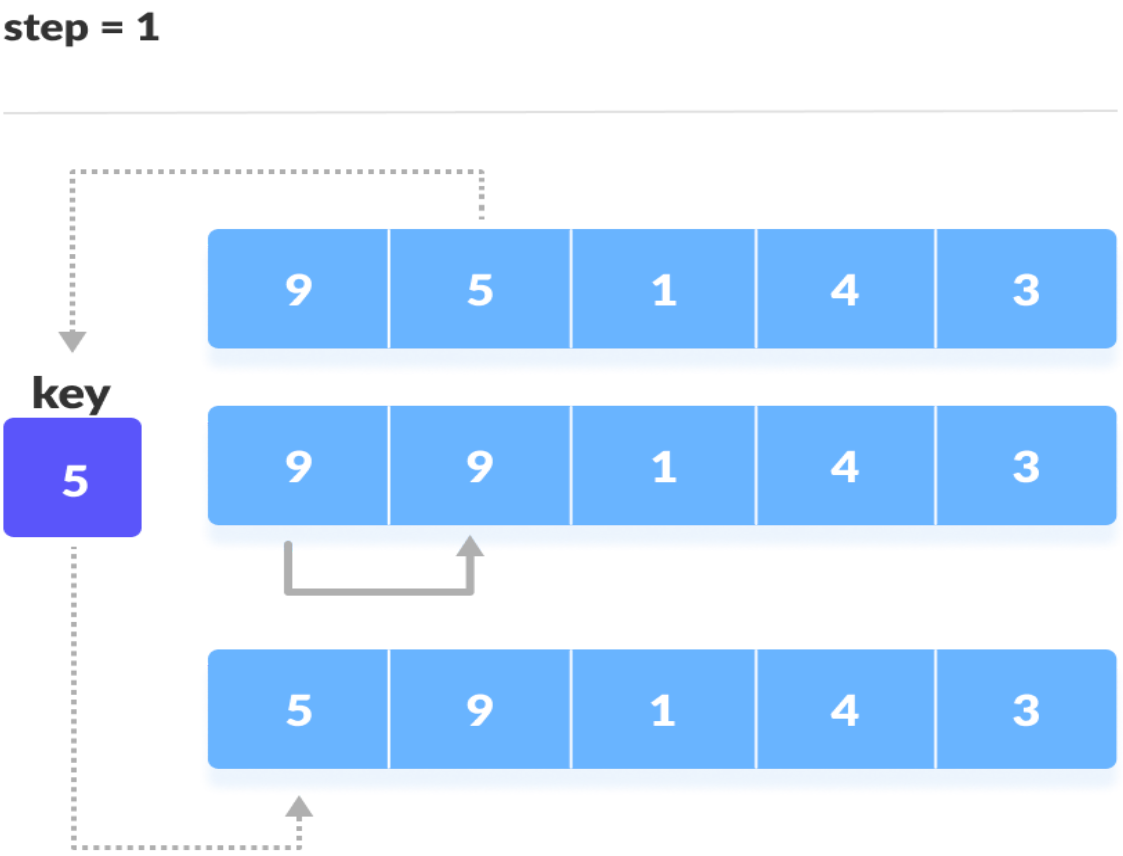
*Working of Insertion Sort:*

Suppose we need to sort the following array.


Initial array

*1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.*

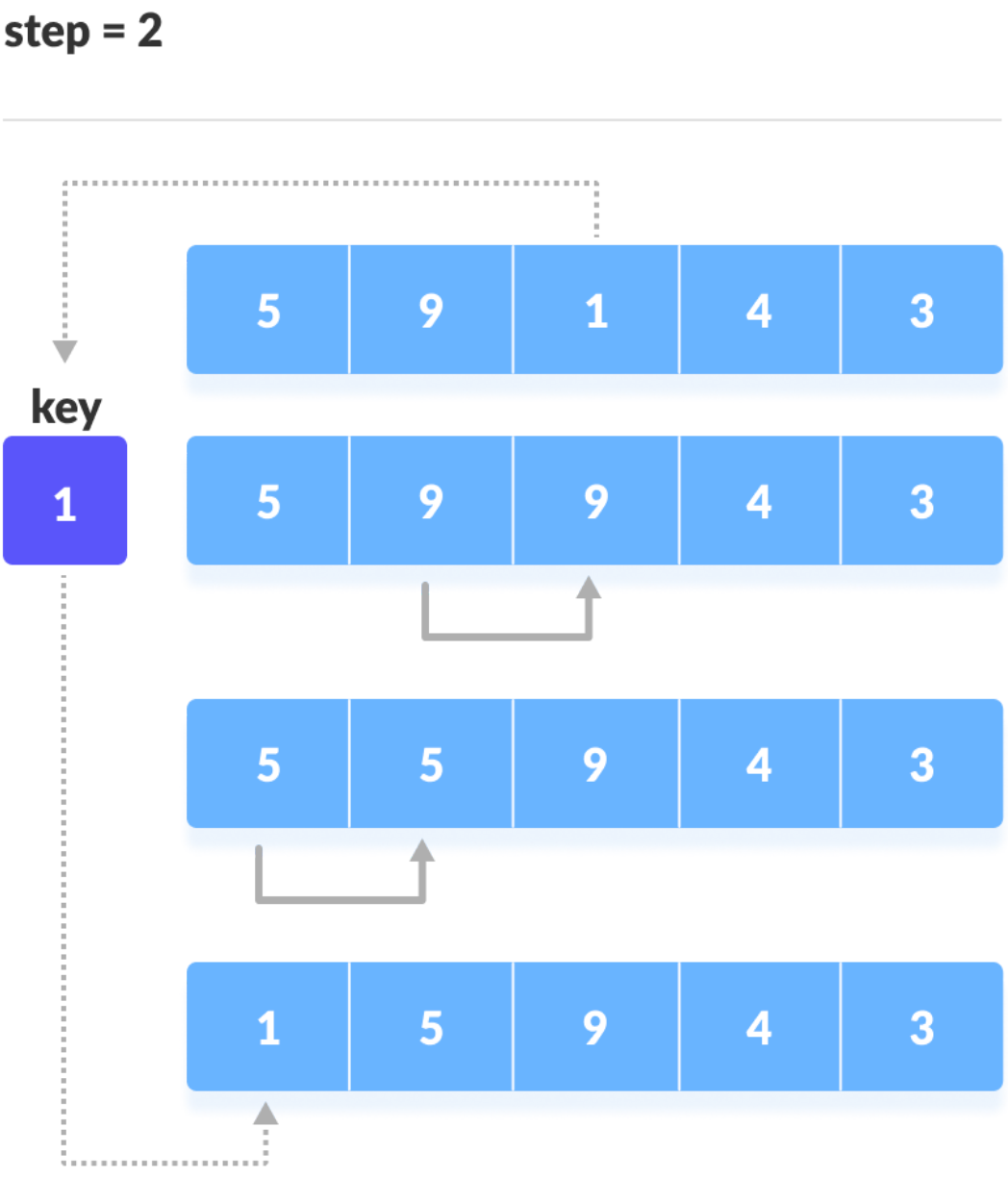
- Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.




If the first element is greater than key, then key is placed in front of the first element.

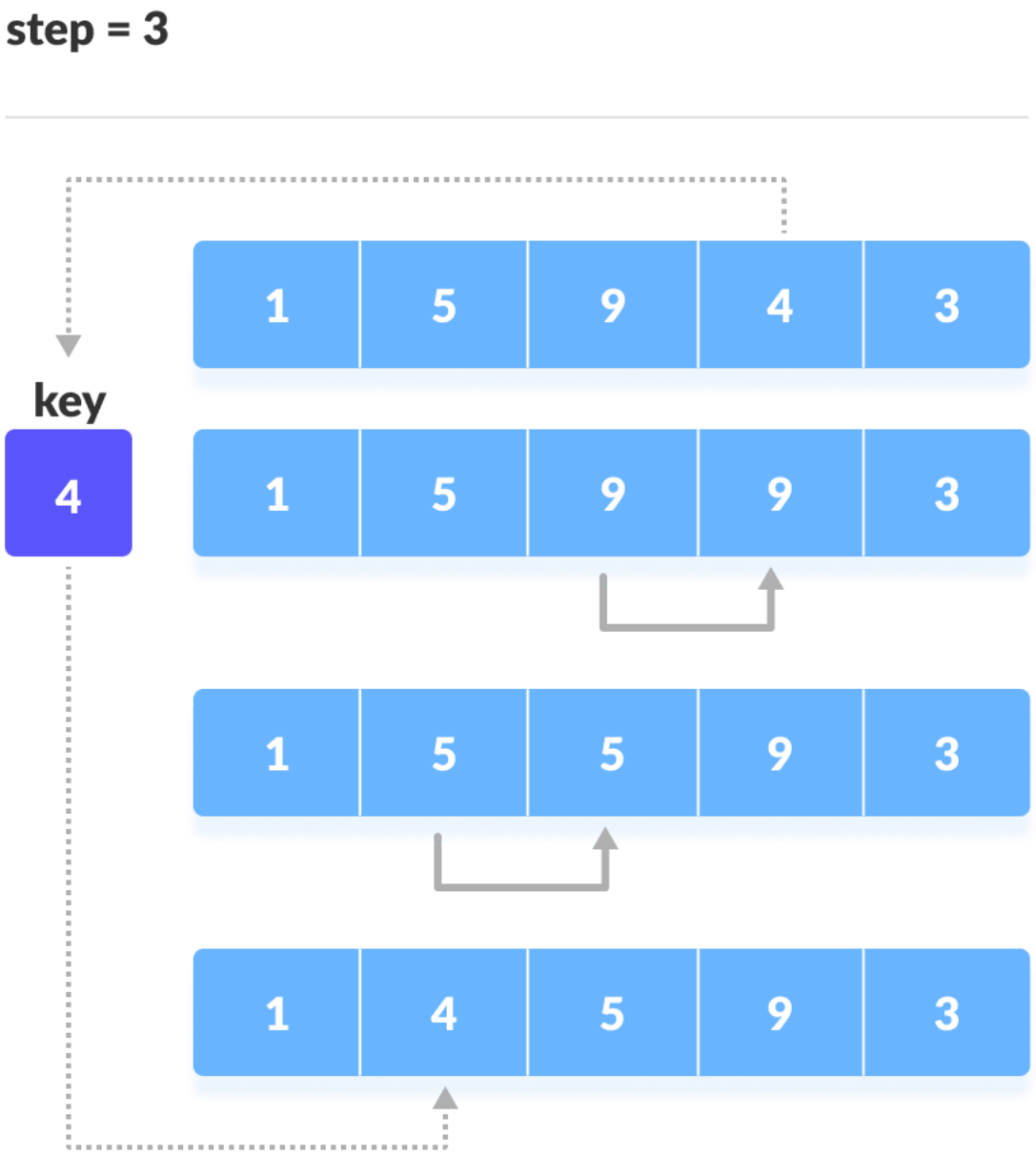
*2.Now, the first two elements are sorted.*

- Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.



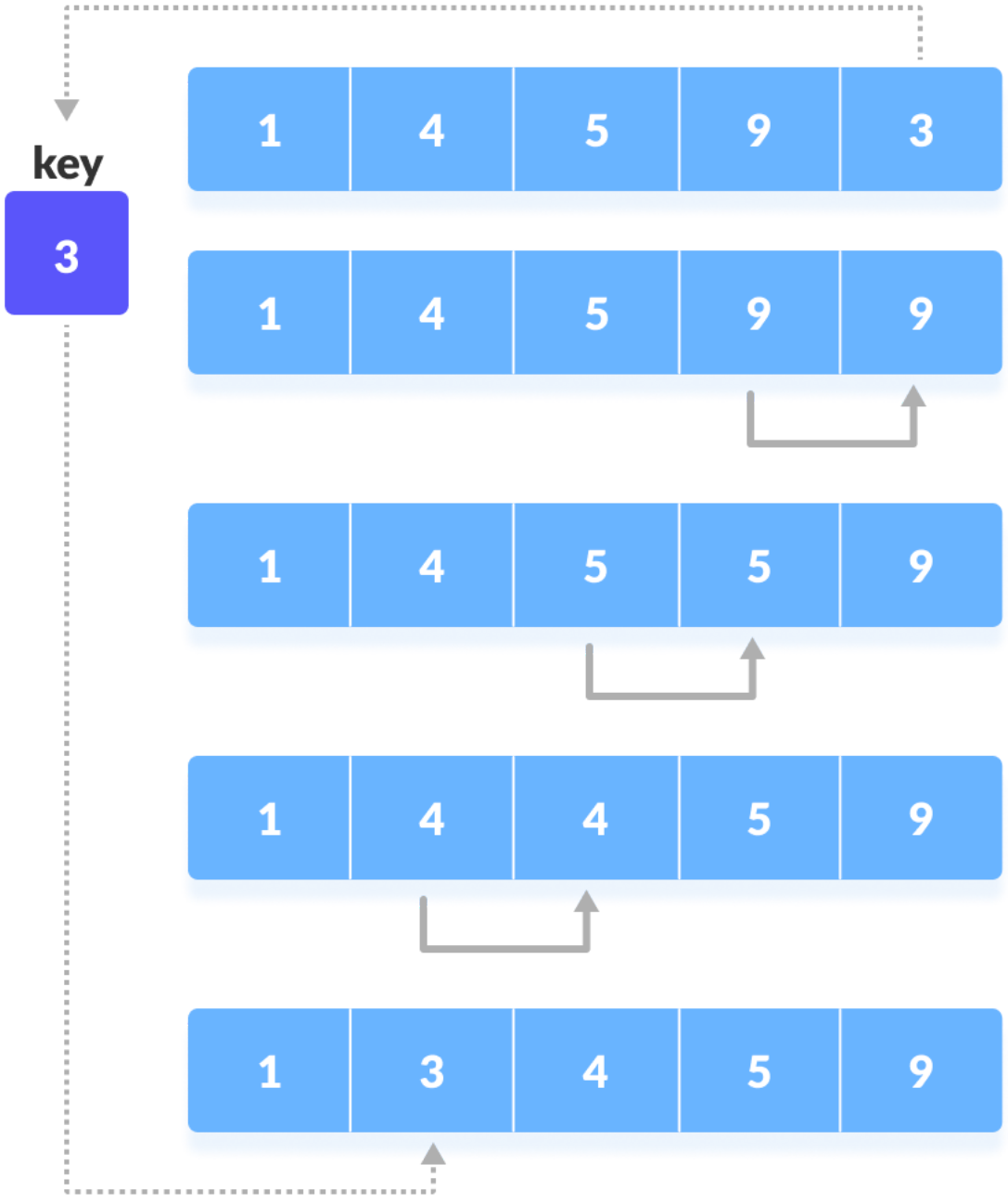

Place 1 at the beginning

*3.Similarly, place every unsorted element at its correct position*




Place 4 behind 1

step = 4




Place 3 behind 1 and then array is sorted

**Insertion sort Algorithm:**

```
// Function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
    printf("%d ", array[i]);
  }
  printf("\n");
}
void insertionSort(int array[], int size) {
  for (int step = 1; step < size; step++) {
    int key = array[step];
    int j = step - 1;
    // Compare key with each element on the left of it until an element smaller than
    // it is found.
    // For descending order, change key<array[j] to key>array[j].
    while (key < array[j] && j >= 0) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
```

```
  }
}
```

**Insertion Sort Complexity**

*Time Complexity:*

Best - O(n)

Worst - O(n2)

Average - O(n2)

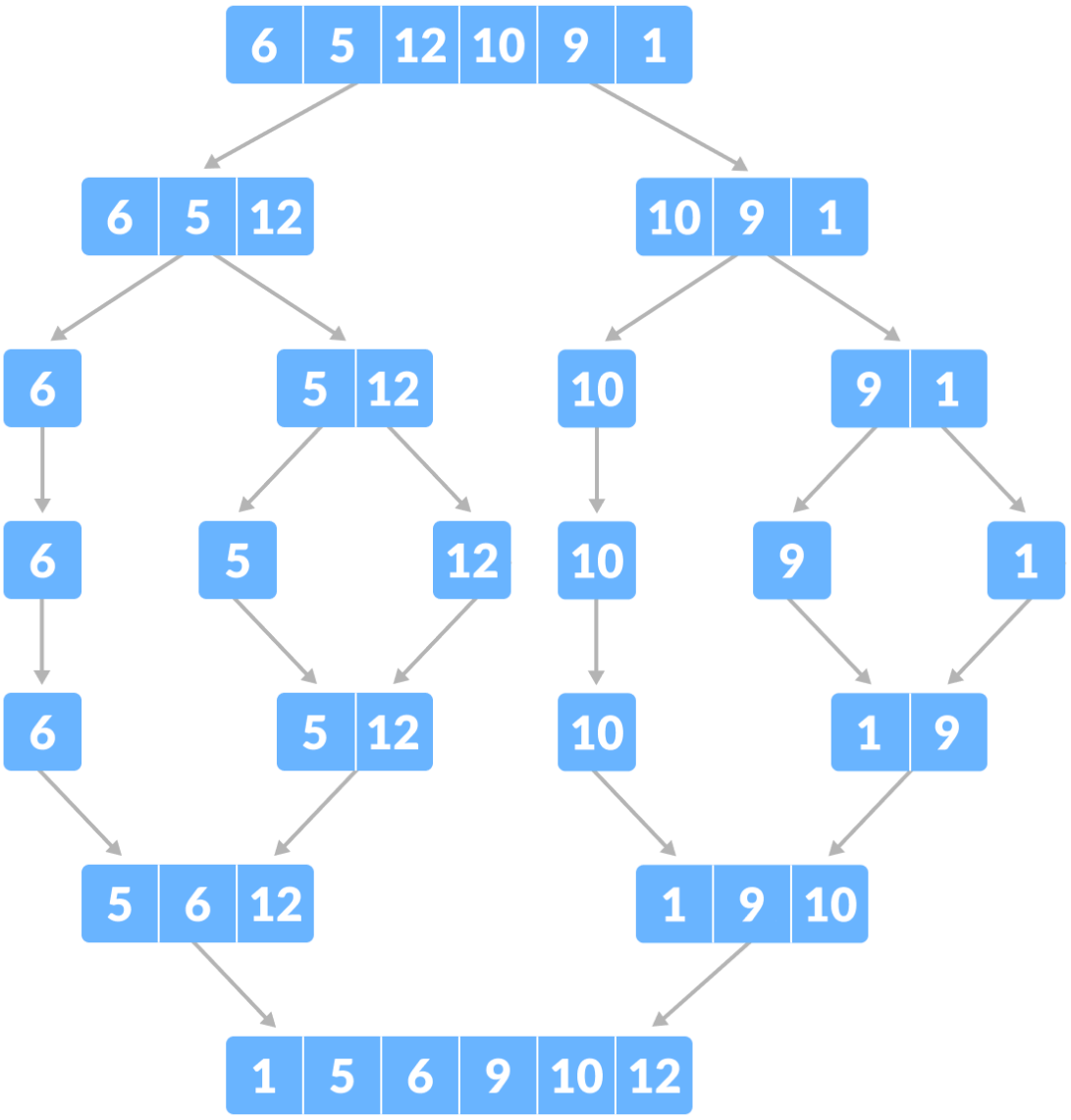
Space Complexity - O(1)

Stability - Yes

## MERGE SORT

*Definition:*

- Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.
- Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.




Merge Sort Example

**Merge sort Algorithm:**

// Until we reach either end of either L or M, pick larger among

// elements L and M and place them in the correct position at A[p..r]

while (i < n1 && j < n2) {

if (L[i] <= M[j]) {

```
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = M[j];
      j++;
    }
    k++;
  }
  // When we run out of elements in either L or M,
  // pick up the remaining elements and put in A[p..r]
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }
  while (j < n2) {
    arr[k] = M[j];
    j++;
    k++;
  }
}
// Divide the array into two subarrays, sort them and merge them
```

```
void mergeSort(int arr[], int l, int r) {
  if (l < r) {
    // m is the point where the array is divided into two
subarrays
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    // Merge the sorted subarrays
    merge(arr, l, m, r);
  }
}
// Print the array
void printArray(int arr[], int size) {
  for (int i = 0; i < size; i++)
    printf("%d ", arr[i]);
  printf("\n");
}
```

**Merge Sort Complexity**

*Time Complexity:*

Best - O(n*log n)

Worst - O(n*log n)

Average - O(n*log n)

Space Complexity - O(n)

Stability - Yes

## QUICK SORT

*Definition:*

- Quicksort is a sorting algorithm based on the divide and conquer approach where
- An array is divided into subarrays by selecting a pivot element (element selected from the array).
- While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
- The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
- At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

*Working of Quicksort Algorithm:*

***1. Select the Pivot Element:***

- There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.

Select a pivot element

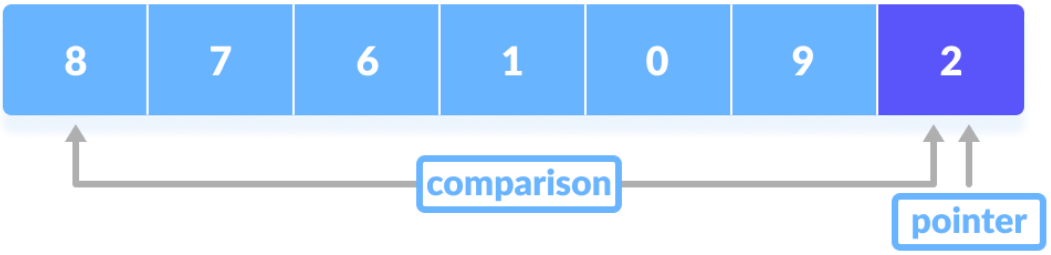
***2. Rearrange the Array:***

- Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.


Put all the smaller elements on the left and greater on the right of pivot element.

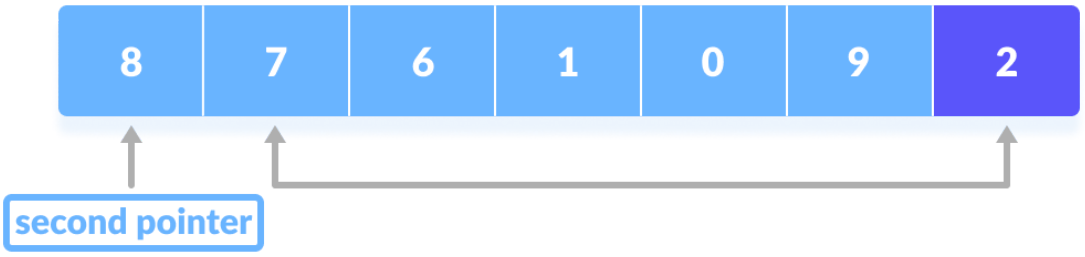
*Here's how we rearrange the array:*

*1.A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.*

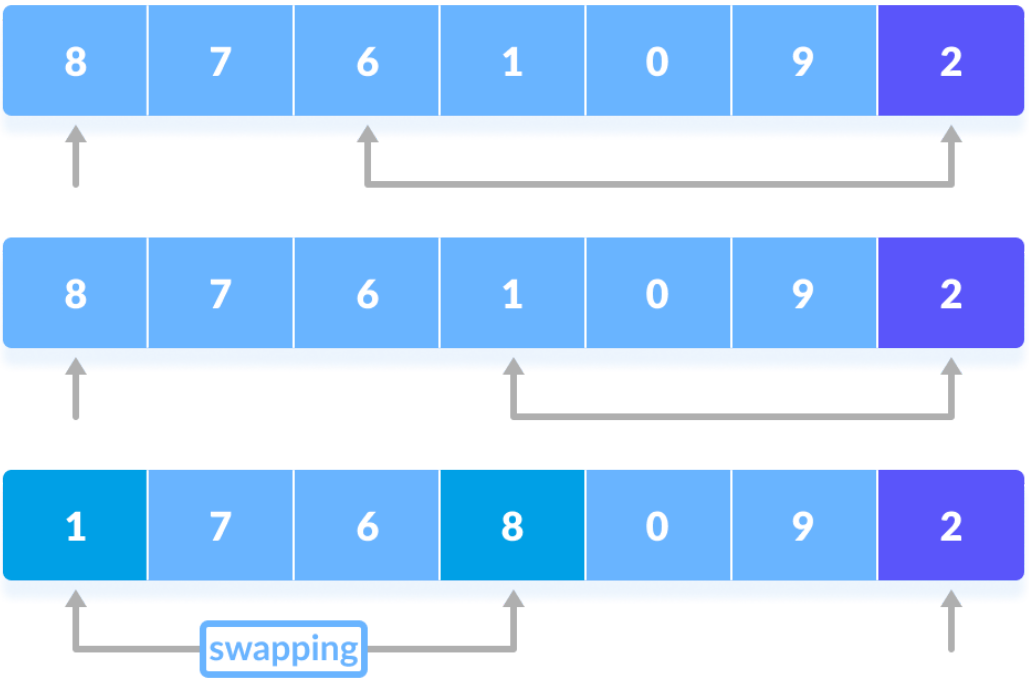



Comparison of pivot element with element beginning from the first index.

*2.If the element is greater than the pivot element, a second pointer is set for that element.*

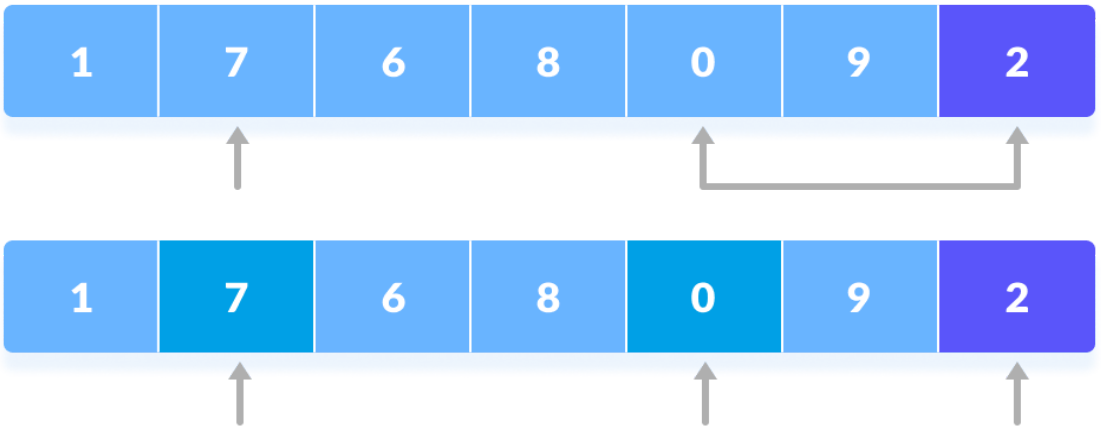



*3.Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.*



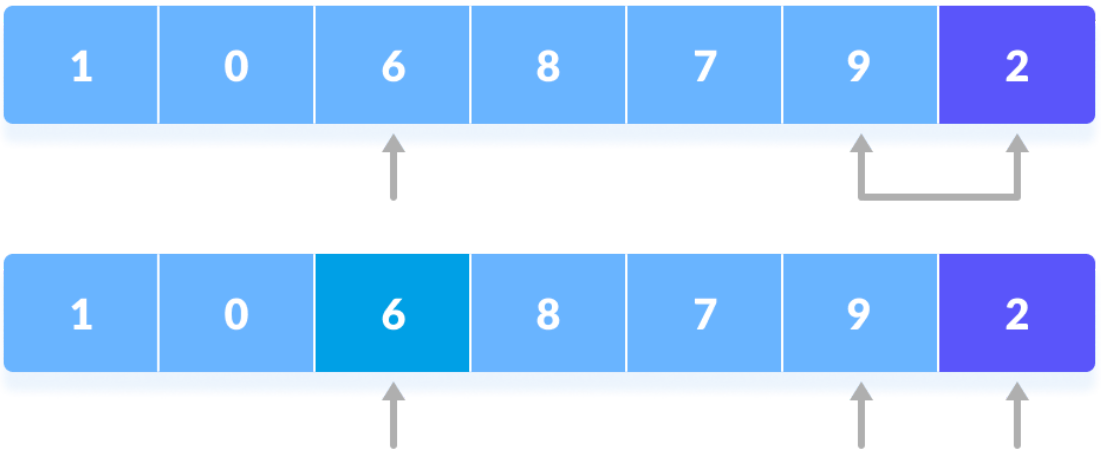

Pivot is compared with other elements

*4.Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.*

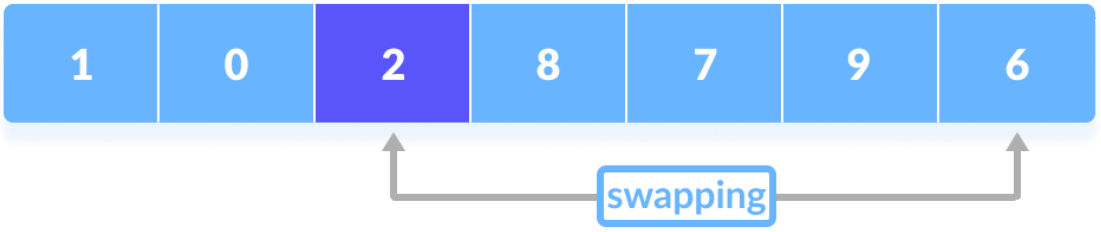



The process is repeated to set the next greater element as the second pointer.

*5.The process goes on until the second last element is reached.*

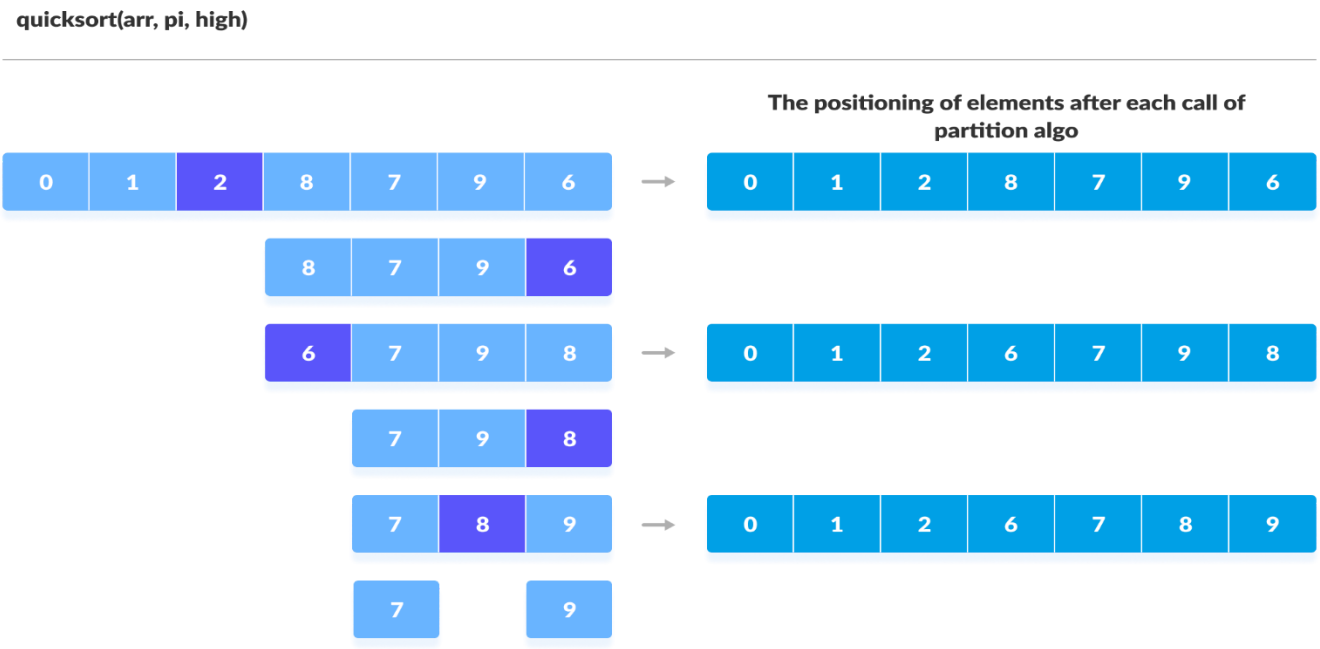



*6.Finally, the pivot element is swapped with the second pointer.*




## ***3. Divide Subarrays:***

- Pivot elements are again chosen for the left and the right sub-parts separately. And, step 2 is repeated.




Select pivot element of in each half and put at correct place using recursion.

**Quick Sort Algorithm:**

```
  // function to swap elements
void swap(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}

// function to find the partition position
int partition(int array[], int low, int high) {
  // select the rightmost element as pivot
  int pivot = array[high];
  // pointer for greater element
  int i = (low - 1);
  // traverse each element of the array
  // compare them with the pivot
  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {
      // if element smaller than pivot is found
      // swap it with the greater element pointed by i
      i++;
      // swap element at i with element at j
      swap(&array[i], &array[j]);
```

```
    }
  }
  // swap the pivot element with the greater element at i
  swap(&array[i + 1], &array[high]);
  // return the partition point
  return (i + 1);
}
void quickSort(int array[], int low, int high) {
  if (low < high) {
    // find the pivot element such that
    // elements smaller than pivot are on left of pivot
    // elements greater than pivot are on right of pivot
    int pi = partition(array, low, high);
    // recursive call on the left of pivot
    quickSort(array, low, pi - 1);
    // recursive call on the right of pivot
    quickSort(array, pi + 1, high);
  }
}
// function to print array elements
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
```

```
  }
  printf("\n");
}
}
```

**Quick Sort Complexity**

*Time Complexity:*

Best - O(n*log n)

Worst - O(n2)

Average - O(n*log n)

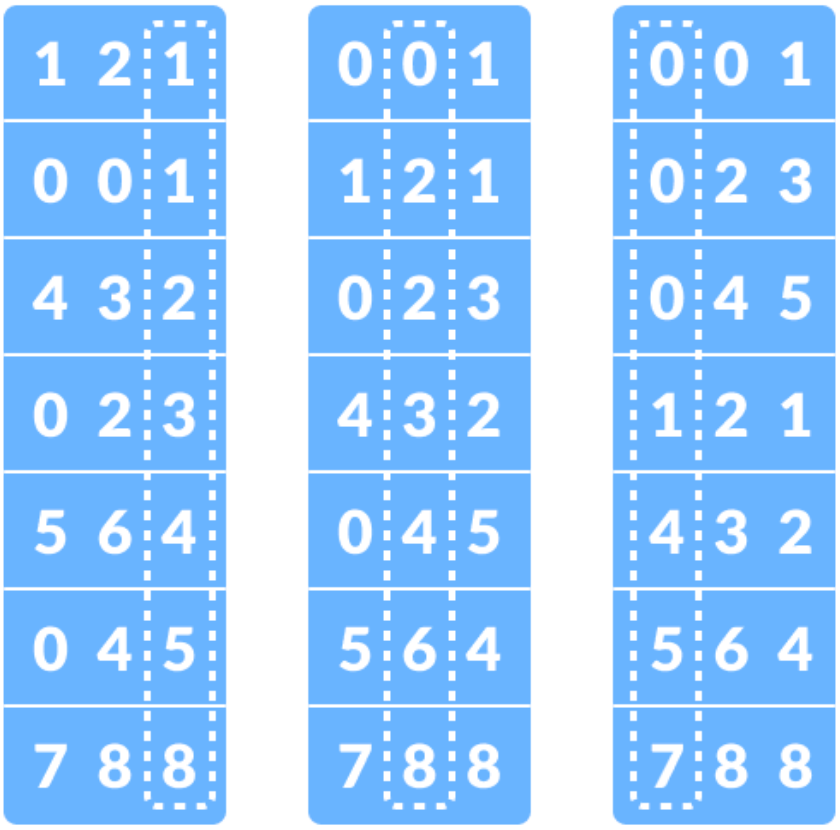
Space Complexity - O(log n)

Stability - Yes

## RADIX SORT

*Definition:*

- Radix sort is a sorting algorithm that sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order.
- Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place.




sorting the integers according to units, tens and hundreds place digits

Working of Radix Sort

*Working of Radix Sort:*

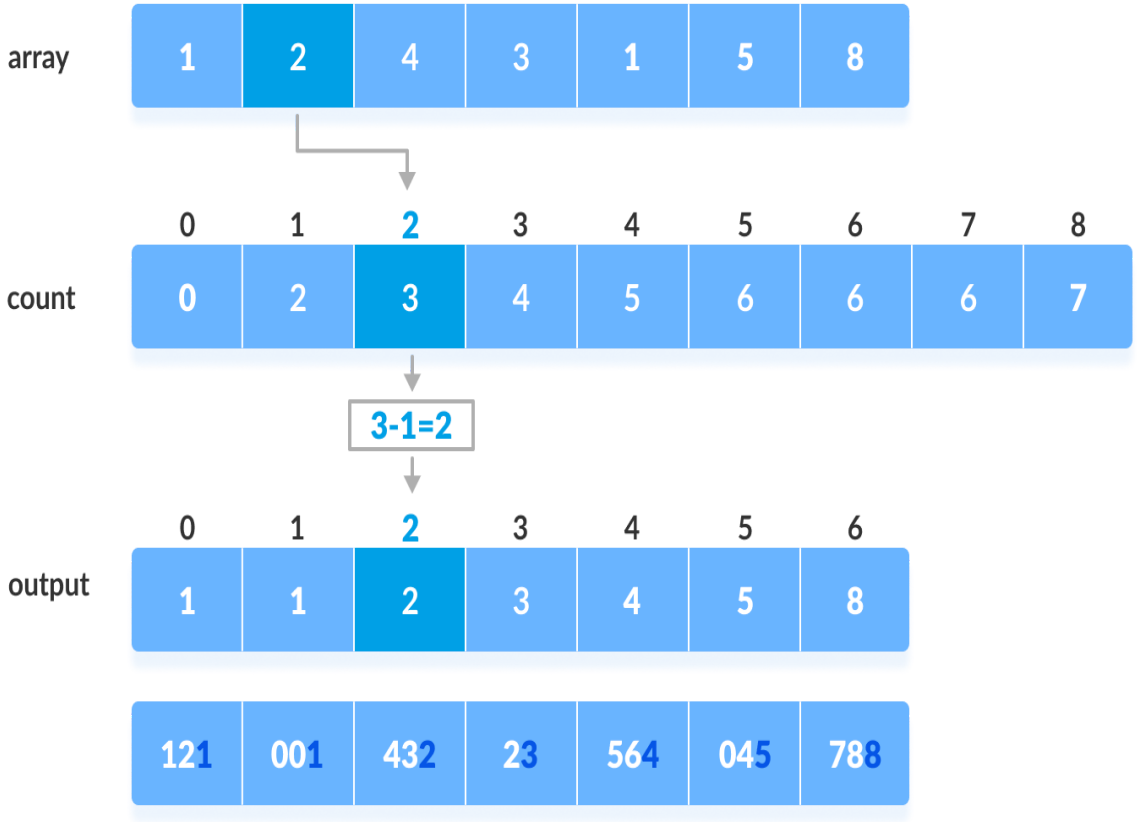
*1.Find the largest element in the array, i.e. max. Let X be the number of digits in max. X is calculated because we have to go through all the significant places of all elements.*

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

*2.Now, go through each significant place one by one.*

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

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




Using counting sort to sort elements based on unit place

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

Sort elements based on tens place

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


Sort elements based on hundreds place

**Radix Sort Algorithm:**

```
// Function to get the largest element from an array
int getMax(int array[], int n) {
  int max = array[0];
  for (int i = 1; i < n; i++)
    if (array[i] > max)
      max = array[i];
  return max;
}
// Using counting sort to sort the elements in the basis of significant places
void countingSort(int array[], int size, int place) {
  int output[size + 1];
  int max = (array[0] / place) % 10;
  for (int i = 1; i < size; i++) {
```

```
    if (((array[i] / place) % 10) > max)
      max = array[i];
  }
  int count[max + 1];
  for (int i = 0; i < max; ++i)
    count[i] = 0;
  // Calculate count of elements
  for (int i = 0; i < size; i++)
    count[(array[i] / place) % 10]++;
  // Calculate cumulative count
  for (int i = 1; i < 10; i++)
    count[i] += count[i - 1];
  // Place the elements in sorted order
  for (int i = size - 1; i >= 0; i--) {
    output[count[(array[i] / place) % 10] - 1] = array[i];
    count[(array[i] / place) % 10]--;
  }
  for (int i = 0; i < size; i++)
    array[i] = output[i];
}
// Main function to implement radix sort
void radixsort(int array[], int size) {
  // Get maximum element
```

```
  int max = getMax(array, size);


  // Apply counting sort to sort elements based on place
value.
  for (int place = 1; max / place > 0; place *= 10)
    countingSort(array, size, place);
}
// Print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}
```

**Radix Sort Complexity**

*Time Complexity:*

Best - O(n+k)

Worst - O(n+k)

Average - O(n+k)

Space Complexity - O(max)

Stability - Yes