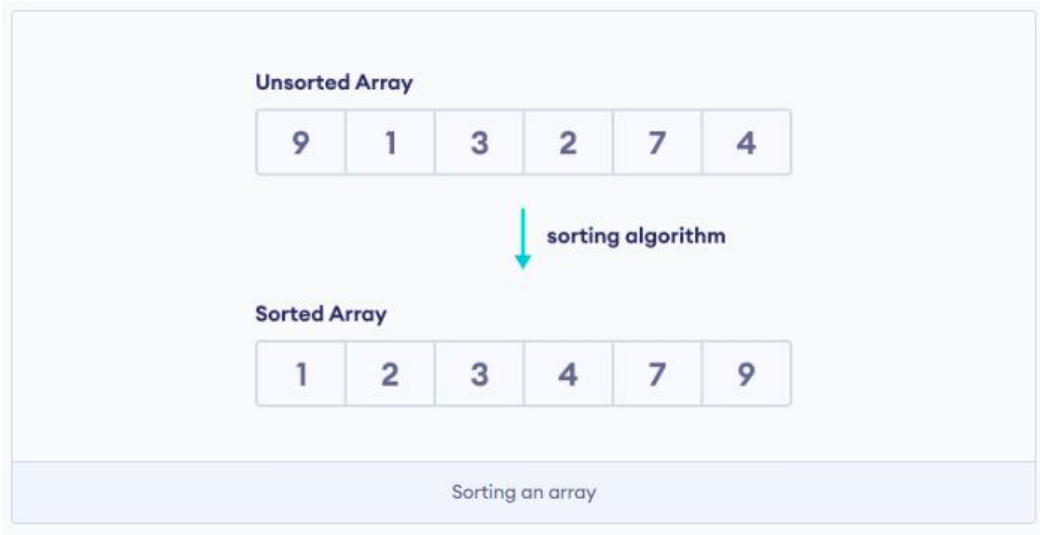
UNIT 4

Searching and Sorting:
Interpolation Search
Sorts: Selection Sort, Insertion Sort, Bubble Sort, Quick Sort, Merge Sort, Radix Sort

SORTING:

Sorting means arranging the given elements or data in an ordered sequence. The main purpose of sorting is to easily & quickly locate an element in a sorted list & design an efficient algorithm around it.



There are various sorting algorithms that can be used to complete this operation. And, we can use any algorithm based on the requirement.

Different Sorting Algorithms

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quicksort
- Radix Sort

Selection Sort Algorithm

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.

Selection sort is a simple sorting algorithm. This sorting algorithm, 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.

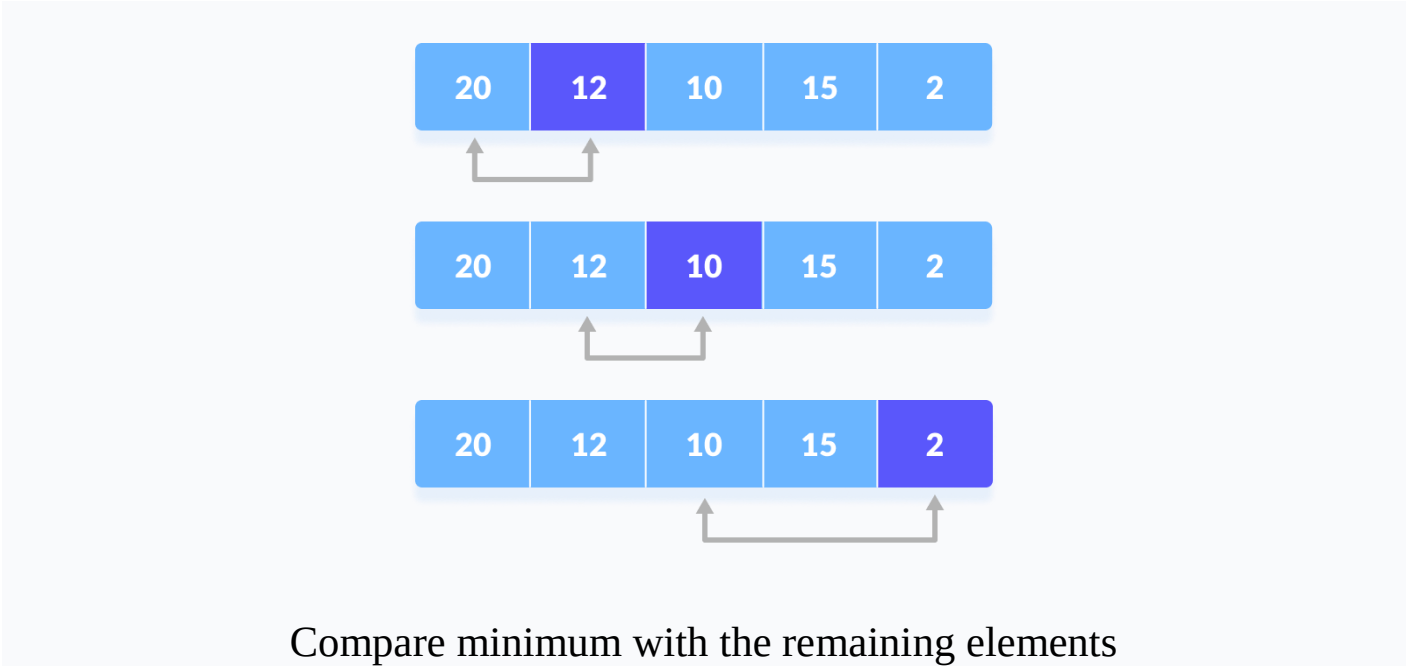
Working of Selection Sort

1. Set the 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.

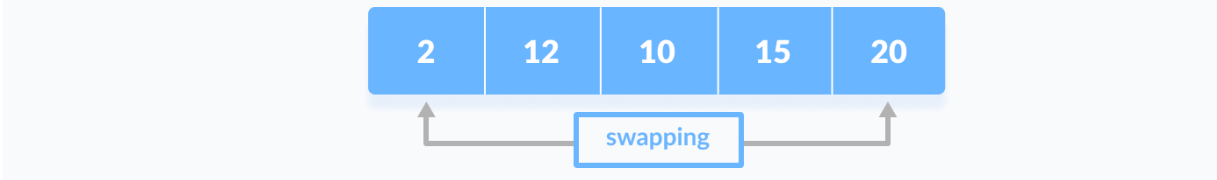
   Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing.

   The process goes on until the last element.

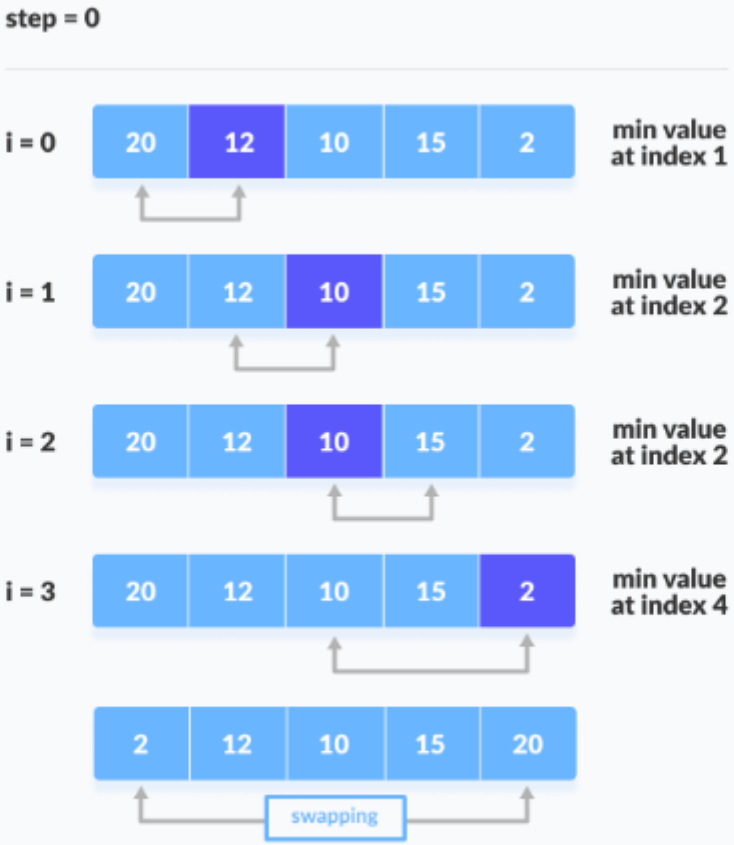


Compare minimum with the remaining elements

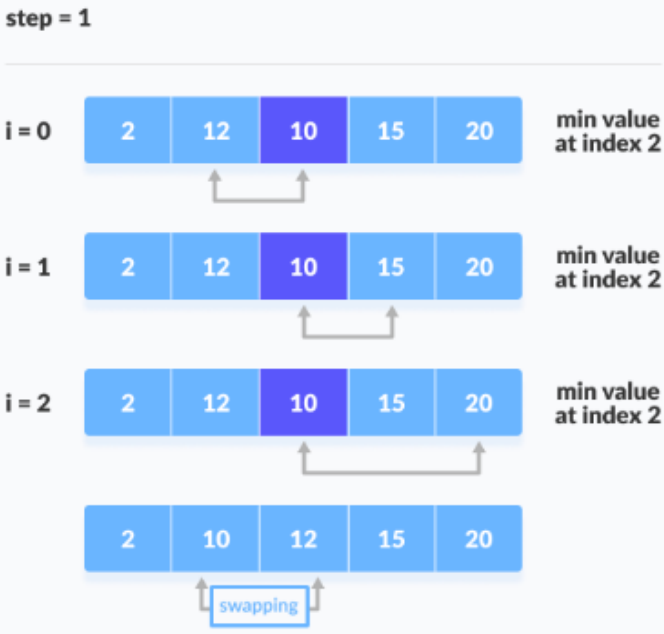
3. After each iteration, `minimum` is placed in the front of the unsorted list.



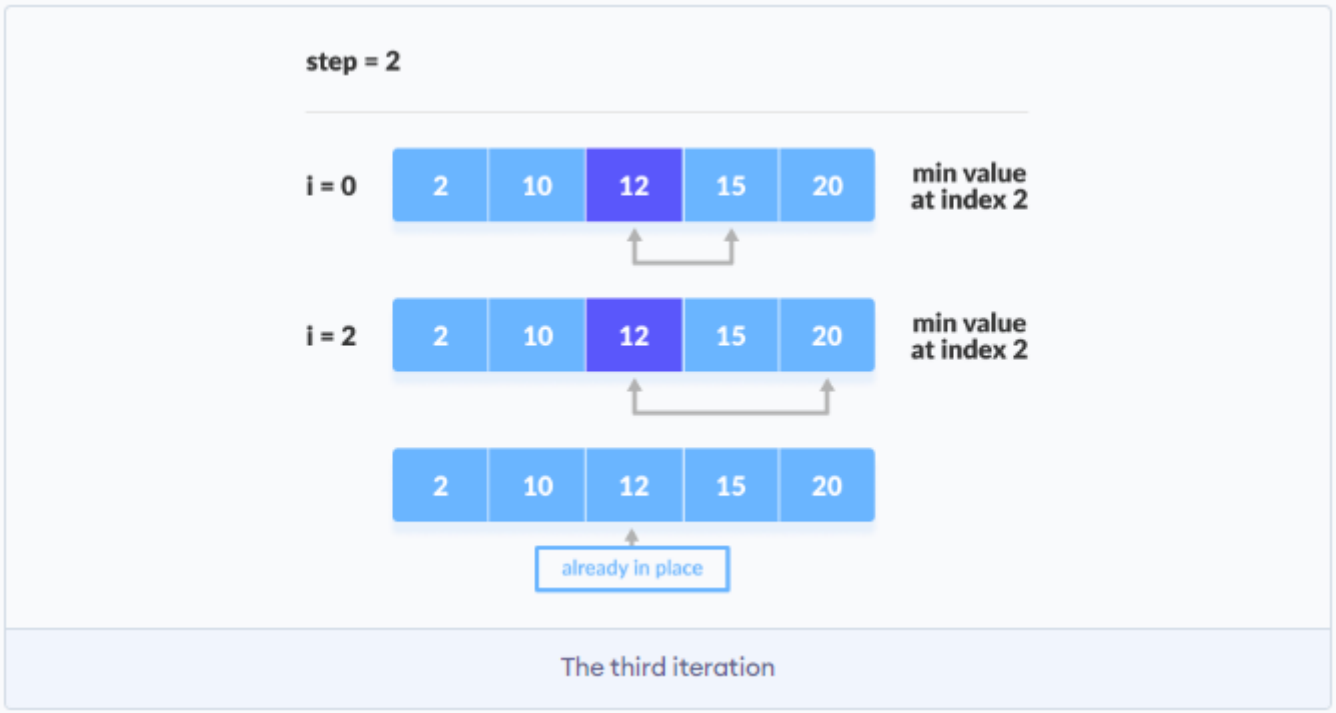
4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions



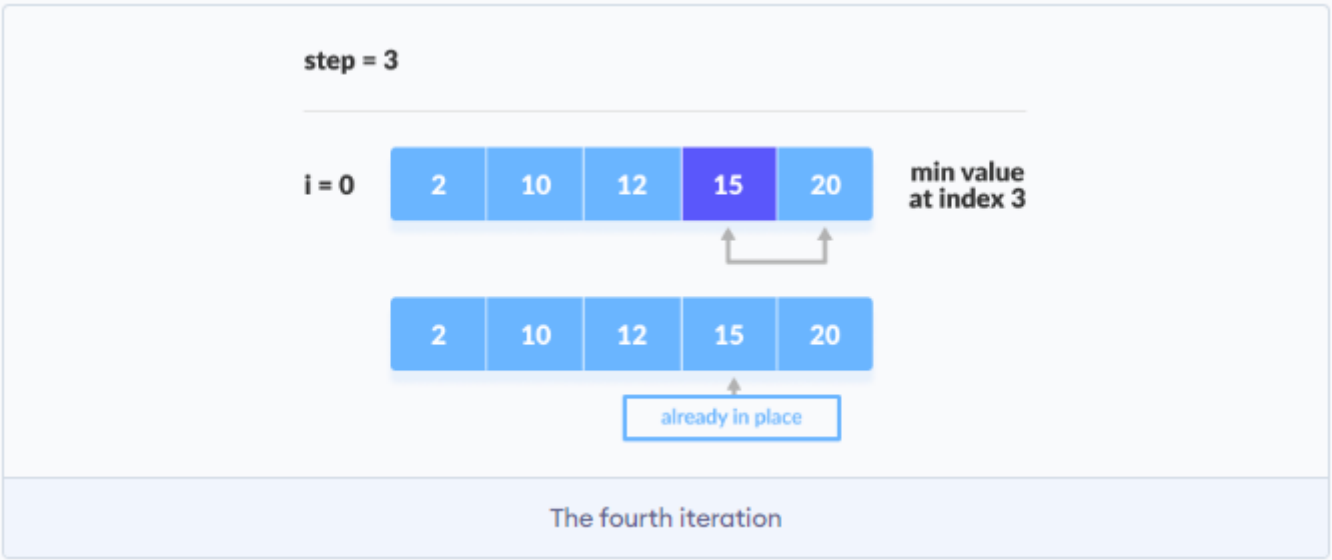
The first iteration



The second iteration



The third iteration



The fourth iteration

Program:

```
#include <stdio.h>
int main() {
   int arr[] = {64, 25, 12, 22, 11};
   int n = sizeof(arr)/sizeof(arr[0]);
   int i, j, min_idx, temp;
   printf("Unsorted array: \n");
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");
```

```
    // Selection Sort function
    for (i = 0; i < n-1; i++) {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }       }
        // Swap the found minimum element with the first element
        temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
    printf("Sorted array: \n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

Output:

Insertion Sort Algorithm

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.

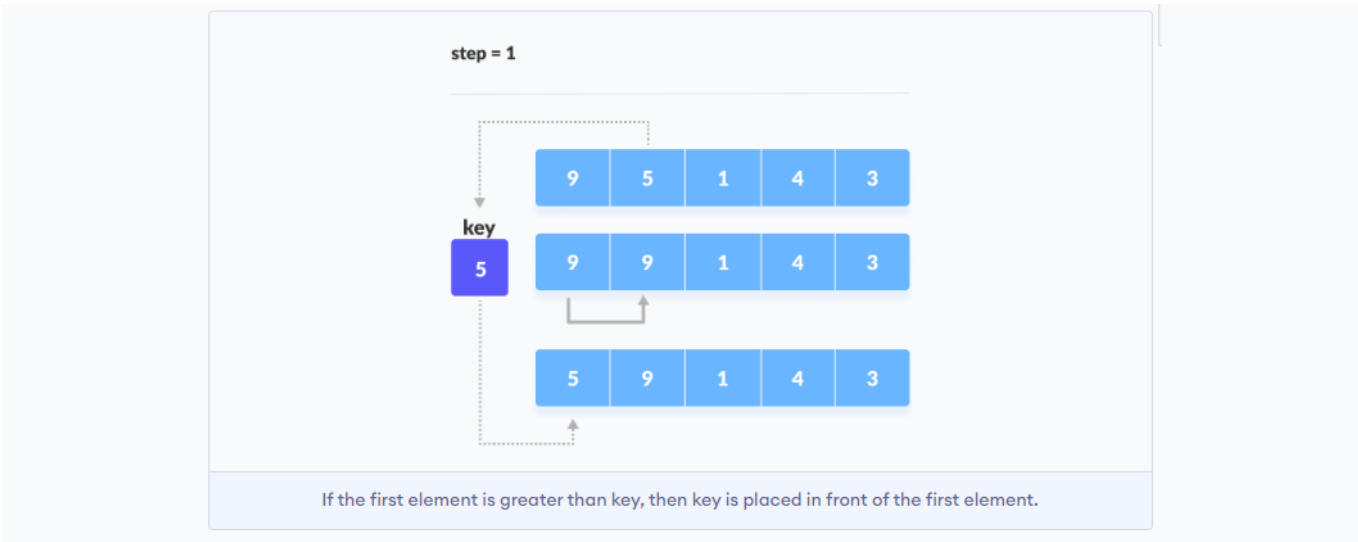
We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

Working of Insertion Sort

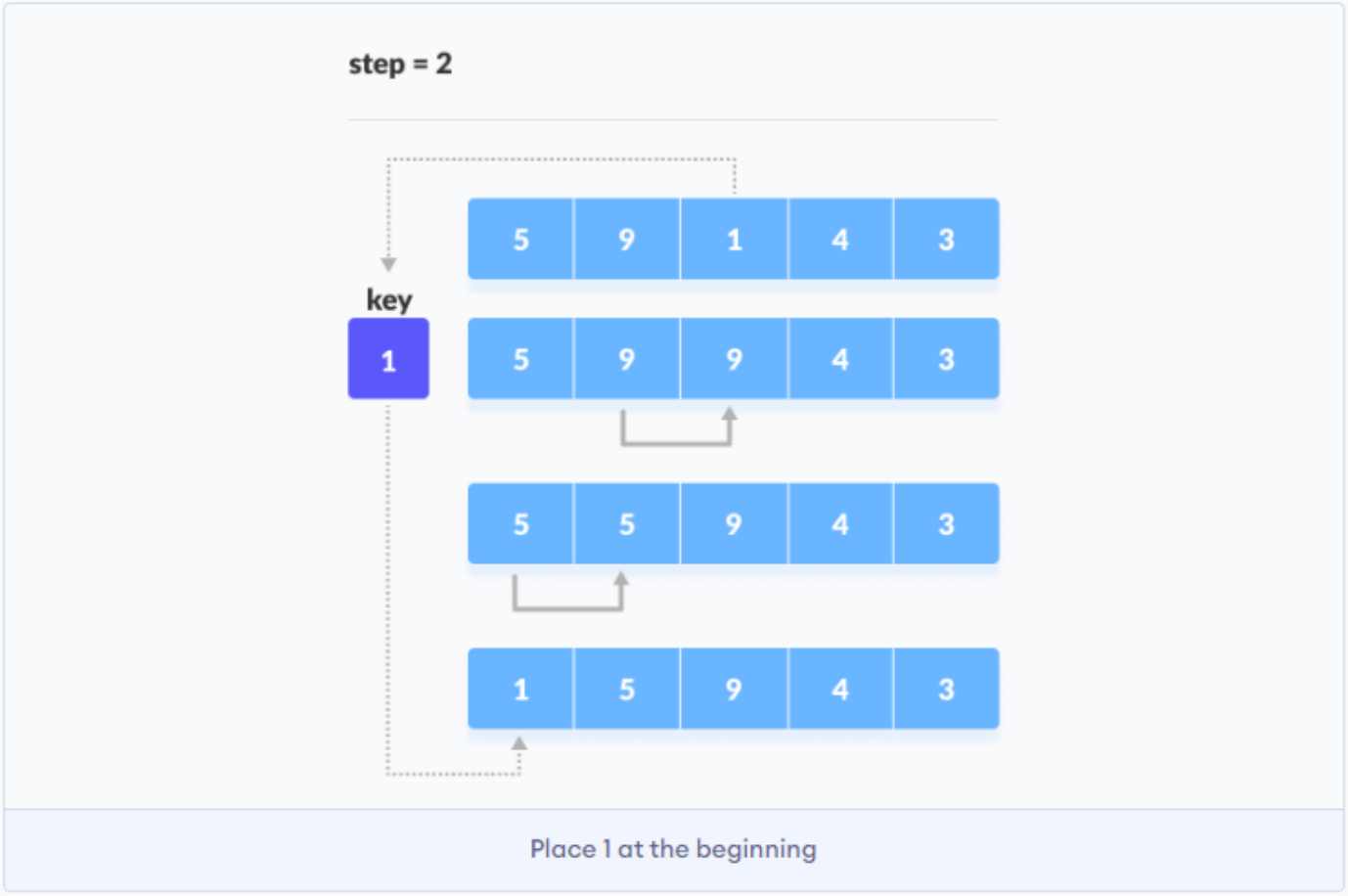
Suppose we need to sort the following array.

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.



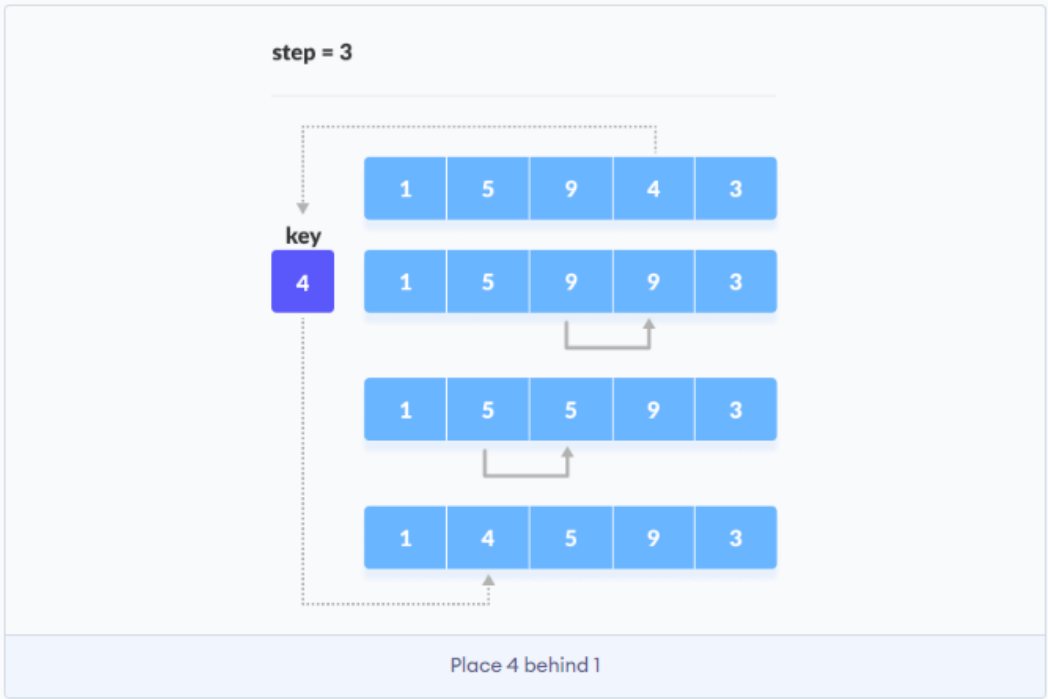
2. Now, the first two elements are sorted.

   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.

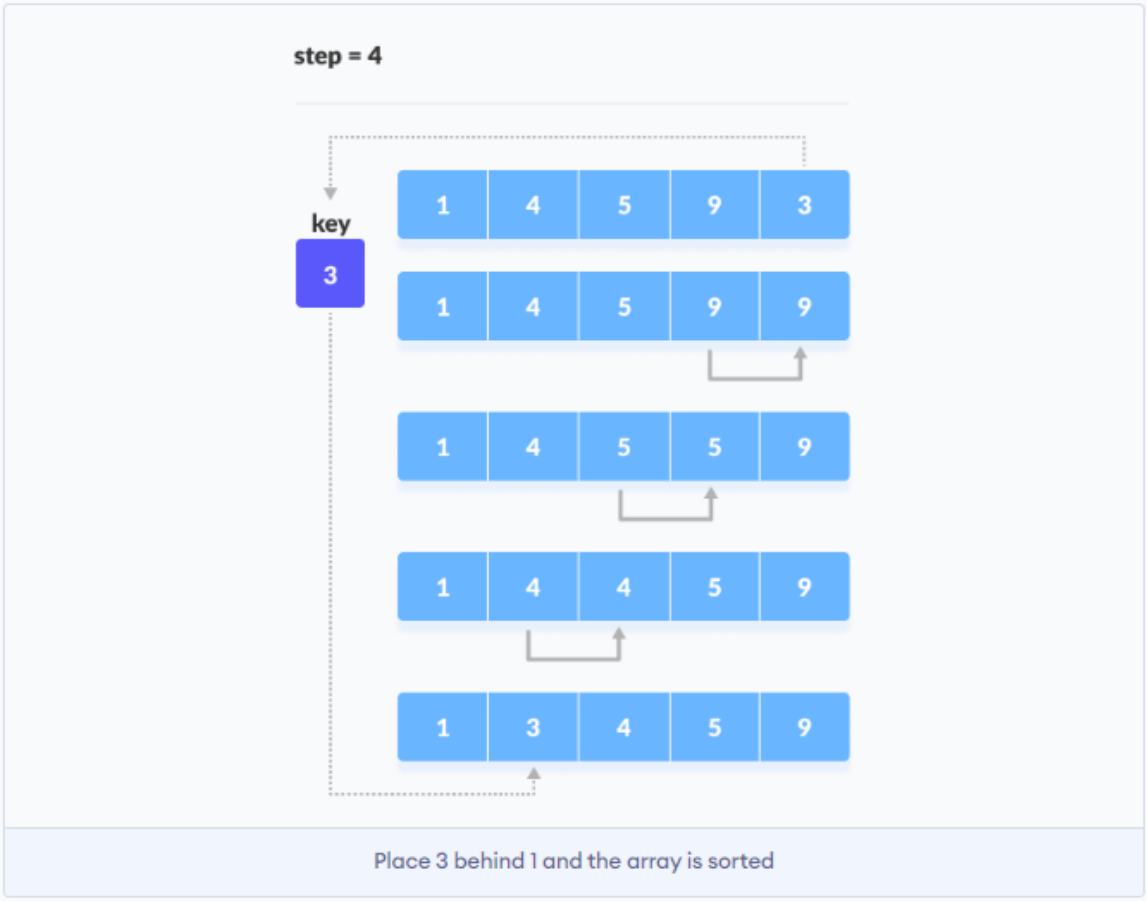


Place 1 at the beginning

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



Place 4 behind 1



Program:

```
#include <stdio.h>
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    int i, j, key;
    printf("Unsorted array: \n");
    for (i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
    // Insertion Sort function
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

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

```
            j = j - 1;
        }
        arr[j + 1] = key;
    }
    printf("Sorted array: \n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

```
Unsorted array:
9 5 1 4 3

Sorted array:
1 3 4 5 9
```

Bubble Sort

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

Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

Working of Bubble Sort

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

1. First Iteration (Compare and Swap)

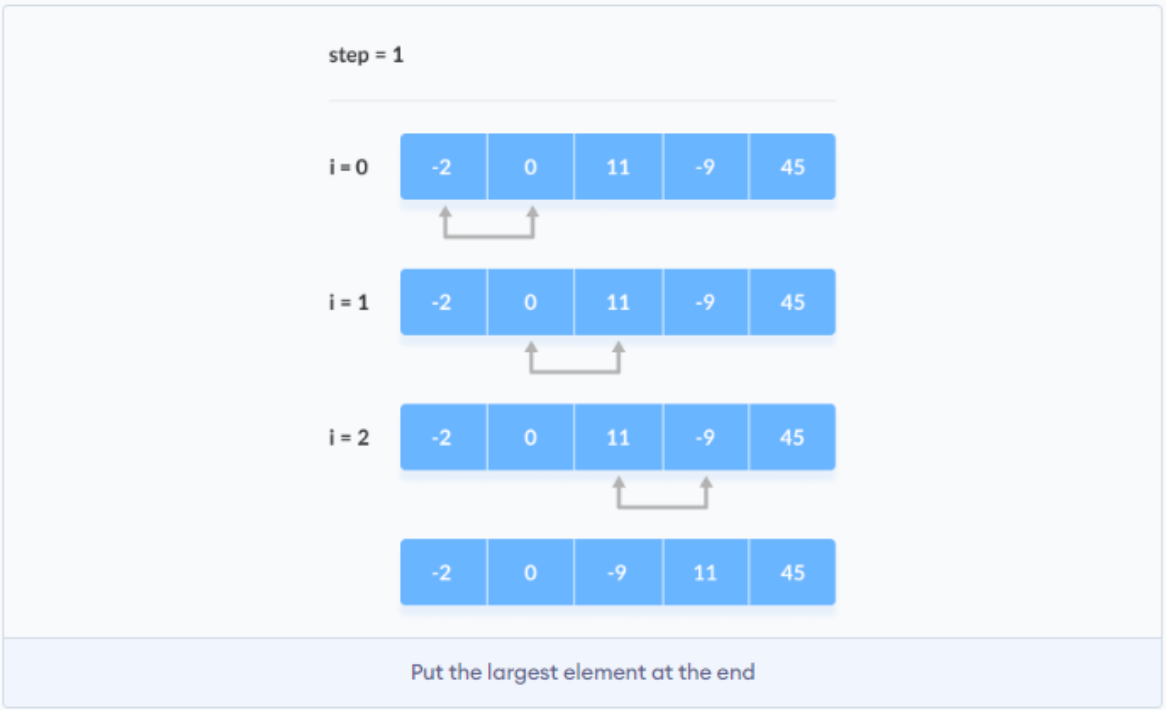
1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.

   The above process goes on until the last element.

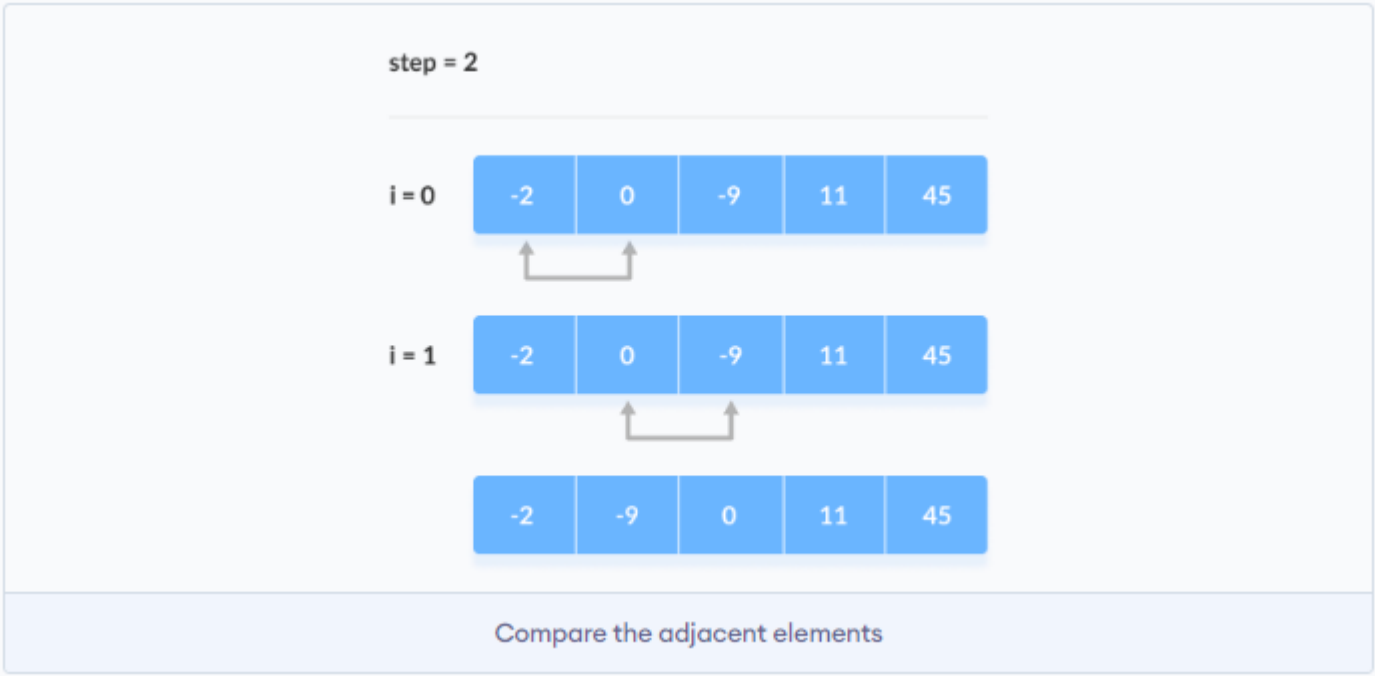
2. Remaining Iteration

The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.



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



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

Program:

```
#include <stdio.h>

int main()
{
   int numbers[] = {-2, 45, 0, 11, -9};
   int length = sizeof(numbers) / sizeof(numbers[0]);
   int outer, inner, temporary;

   printf("Unsorted numbers: \n");
   for (outer = 0; outer < length; outer++)
```

```
    {
        printf("%d ", numbers[outer]);
    }
    printf("\n");

    // Bubble Sort algorithm without swapped flag
    for (outer = 0; outer < length-1; outer++) {
        for (inner = 0; inner < length-outer-1; inner++) {
            if (numbers[inner] > numbers[inner+1]) {
                // Swap the elements
                temporary = numbers[inner];
                numbers[inner] = numbers[inner+1];
                numbers[inner+1] = temporary;
            }
        }
    }

    printf("Sorted numbers: \n");
    for (outer = 0; outer < length; outer++) {
        printf("%d ", numbers[outer]);
    }
    printf("\n");

    return 0;
}
```

```
Unsorted numbers:
-2 45 0 11 -9
Sorted numbers:
-9 -2 0 11 45
```

Quicksort Algorithm

Quicksort is a sorting algorithm based on the **divide and conquer approach** where

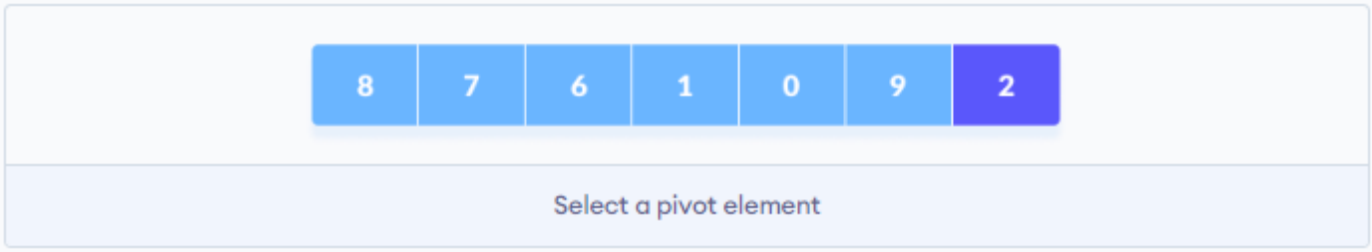
1. An array is divided into subarrays by selecting a **pivot element** (element selected from the array).

   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.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Working of Quicksort Algorithm

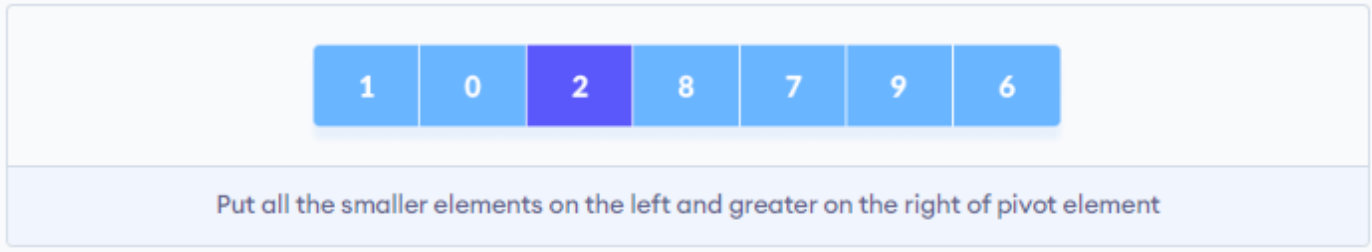
1. Select the Pivot Element

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.



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.

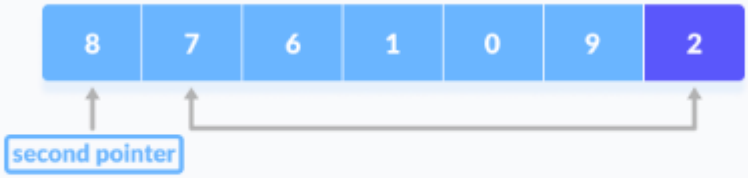


Here's how we rearrange the array:

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.

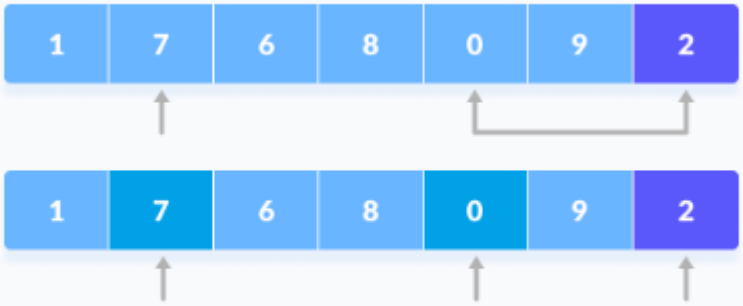


If the element is greater than the pivot element, a second pointer is set for that element.

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.

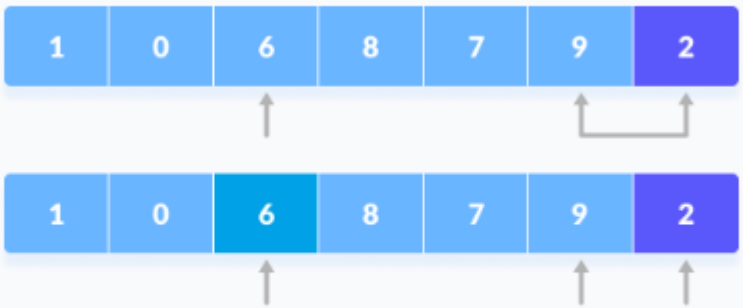
Pivot is compared with other elements.

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



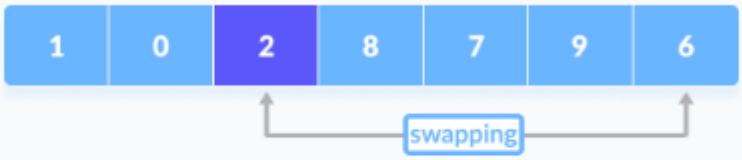
The process is repeated to set the next greater element as the second pointer.

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



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

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



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.

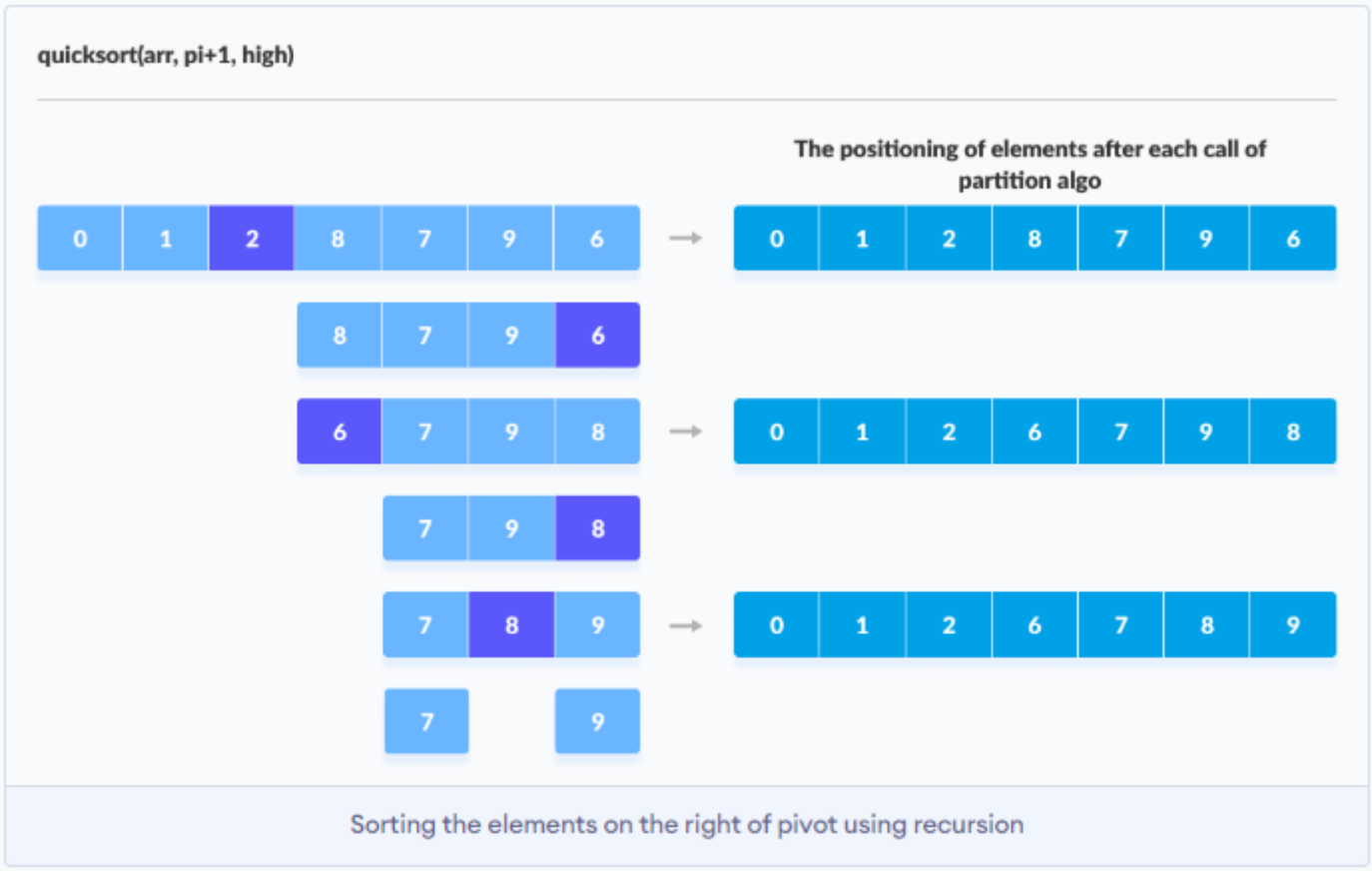
Select pivot element of in each half and put at correct place using recursion

The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted.

Visual Illustration of Quicksort Algorithm

You can understand the working of quicksort algorithm with the help of the illustrations below.

Sorting the elements on the left of pivot using recursion



Program:

```
#include <stdio.h>
int main() {
   int numbers[] = {8,7,6,1,0,9,2};
   int length = sizeof(numbers) / sizeof(numbers[0]);
   printf("Unsorted numbers: \n");
   for (int i = 0; i < length; i++) {
      printf("%d ", numbers[i]);
   }
   printf("\n");
   // Quick Sort algorithm without using separate functions or stack
   int low = 0;
   int high = length - 1;
   // Sorting using a recursive approach within main
   while (low < high) {
      int pivot = numbers[high];
      int i = low - 1;
      for (int j = low; j < high; j++) {
```

```
            if (numbers[j] <= pivot) {
                i++;
                int temp = numbers[i];
                numbers[i] = numbers[j];
                numbers[j] = temp;
            }
        }
        int temp = numbers[i + 1];
        numbers[i + 1] = numbers[high];
        numbers[high] = temp;
        int pi = i + 1;
        // Recursively sort elements before and after partition
        if (pi - low < high - pi) {
            if (low < pi - 1) {
                high = pi - 1;
            } else {
                low = pi + 1;
            }
        } else {
            if (pi + 1 < high) {
                low = pi + 1;
            } else {
                high = pi - 1;
            }
        }
    }
    printf("Sorted numbers: \n");
    for (int i = 0; i < length; i++) {
        printf("%d ", numbers[i]);
    }
    printf("\n");
    return 0;
}
```

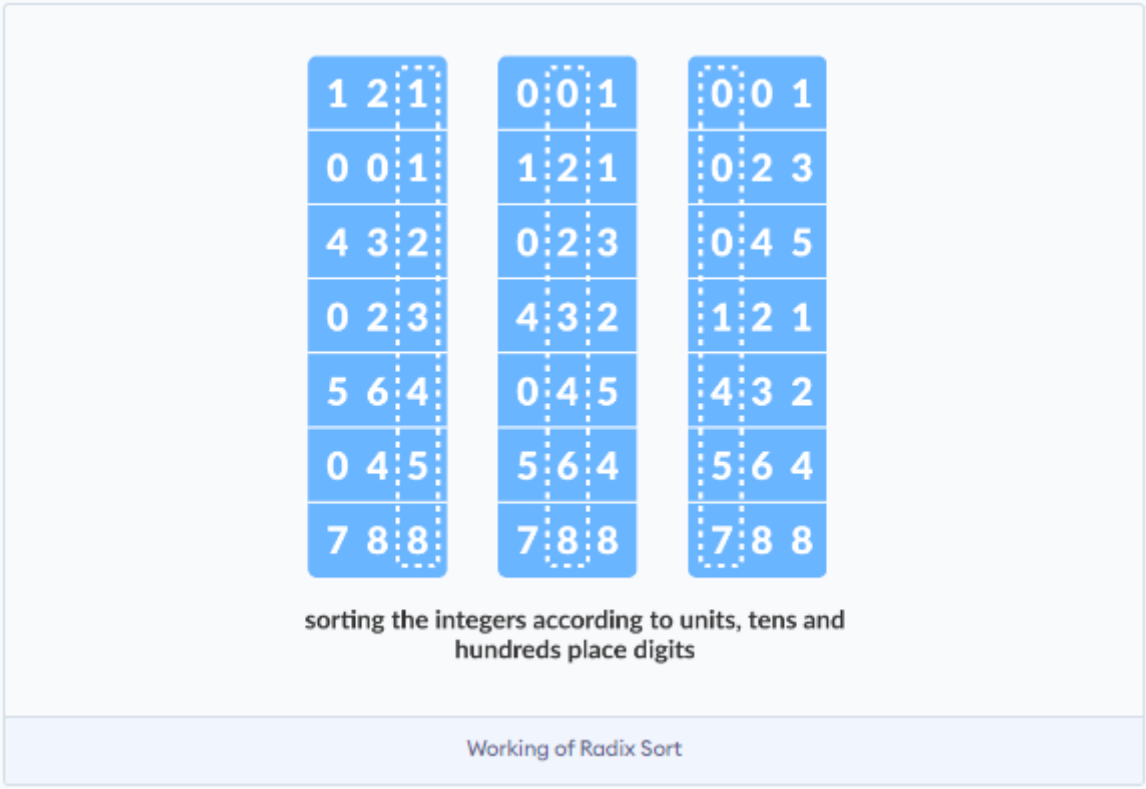
```
Unsorted numbers:
8 7 6 1 0 9 2
Sorted numbers:
0 1 2 8 7 9 6
```

Radix Sort Algorithm

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.

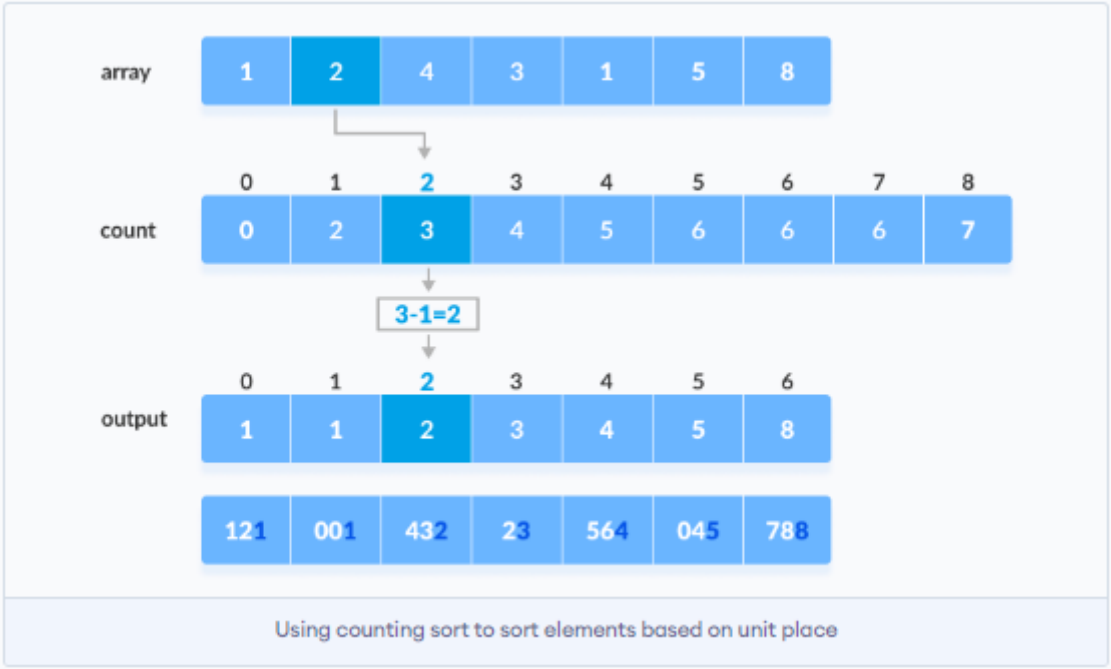
Let the initial array be [121, 432, 564, 23, 1, 45, 788]. It is sorted according to radix sort as shown in the figure below.



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

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int arr[] = {1,2,4,3,1,5,8};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Unsorted array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    // Find the maximum number to know the number of digits
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    // Do counting sort for every digit. exp is 10^i where i is the current digit number
    for (int exp = 1; max / exp > 0; exp *= 10) {
        int output[n];
        int count[10] = {0};
        // Store count of occurrences in count[]
        for (int i = 0; i < n; i++) {
            count[(arr[i] / exp) % 10]++;
        }
        // Change count[i] so that it contains the actual position of this digit in output[]
```

```
        for (int i = 1; i < 10; i++) {
            count[i] += count[i - 1];
        }
        // Build the output array
        for (int i = n - 1; i >= 0; i--) {
            output[count[(arr[i] / exp) % 10] - 1] = arr[i];
            count[(arr[i] / exp) % 10]--;
        }
        // Copy the output array to arr[], so that arr[] now contains sorted numbers according to the current digit
        for (int i = 0; i < n; i++) {
            arr[i] = output[i];
        }
    }
    printf("Sorted array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```