CS -08

DATA STRUCTURE USING

C LANGUAGE

By Rachel

UNIT 2

SORTING AND SEARCHING

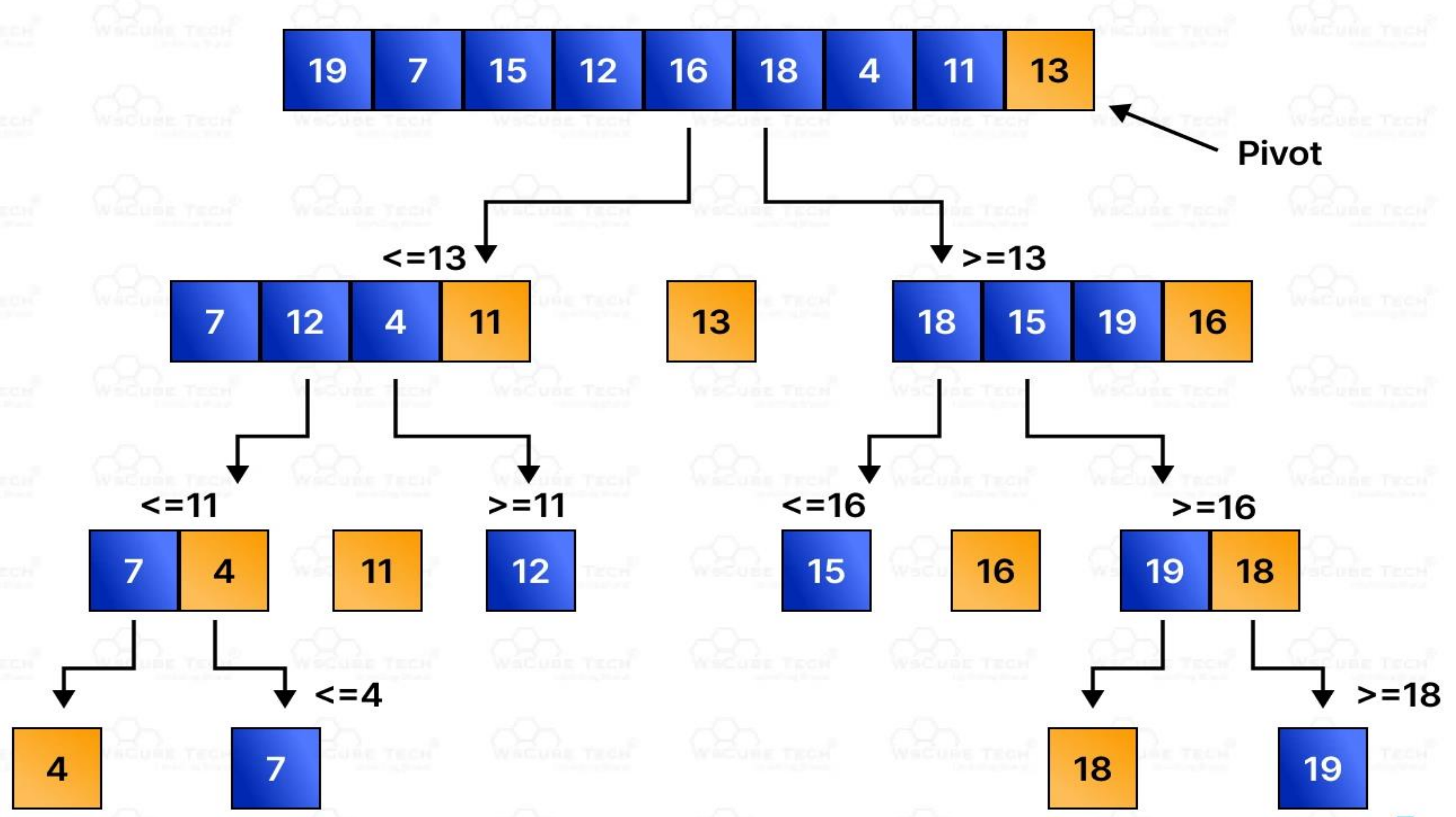
TOPIC 3

QUICK SORTING

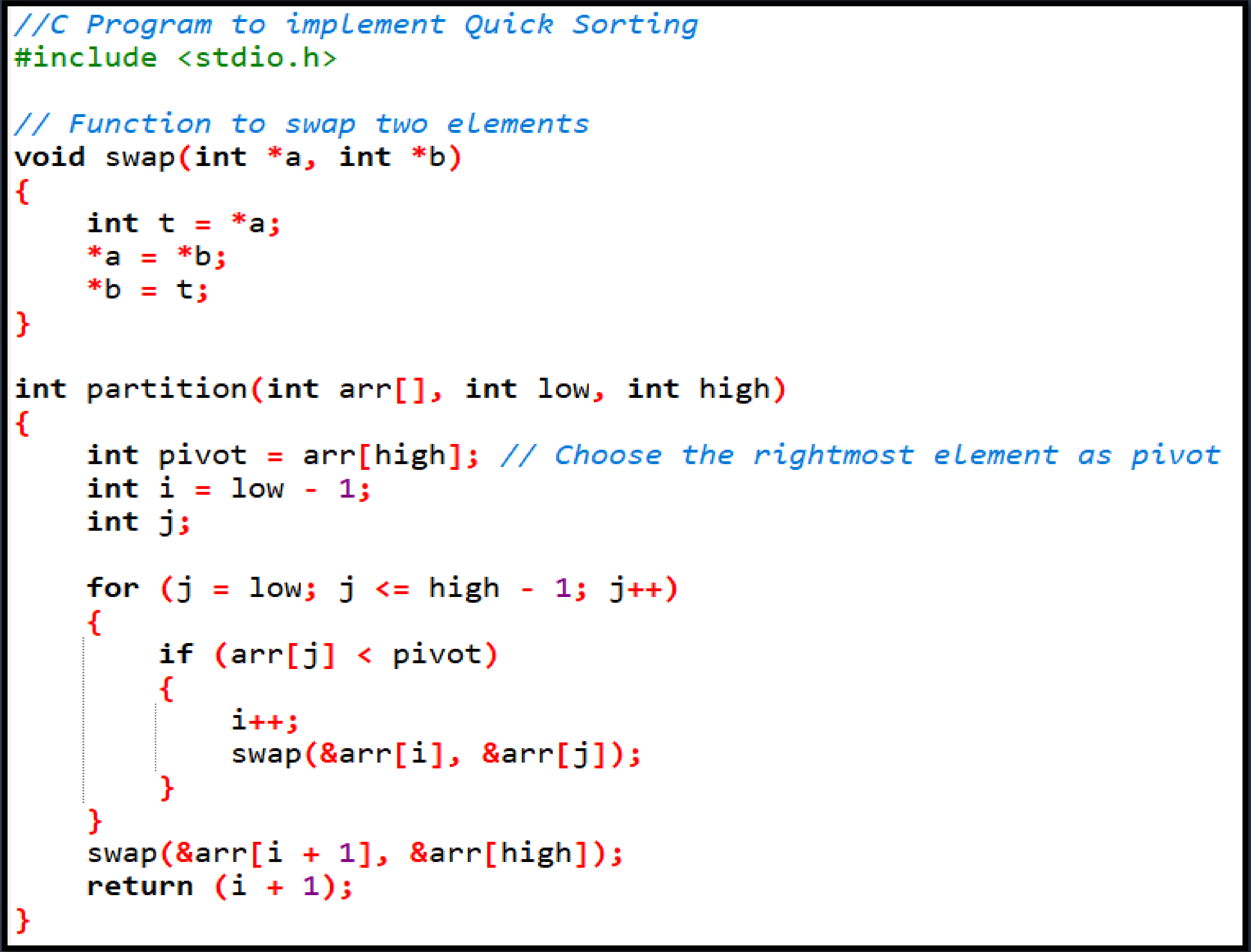
What is Quick Sorting?

- Quick sort is an efficient, in-place, divide-and-conquer sorting algorithm that works by selecting a 'pivot' element and partitioning other elements around it.
- Elements smaller than the pivot are moved to the left, and larger elements to the right.
- This process is applied recursively to the sub-arrays until the entire array is sorted.
- Time Complexity is O(n logn) and the Auxiliary Space complexity is O(log n).

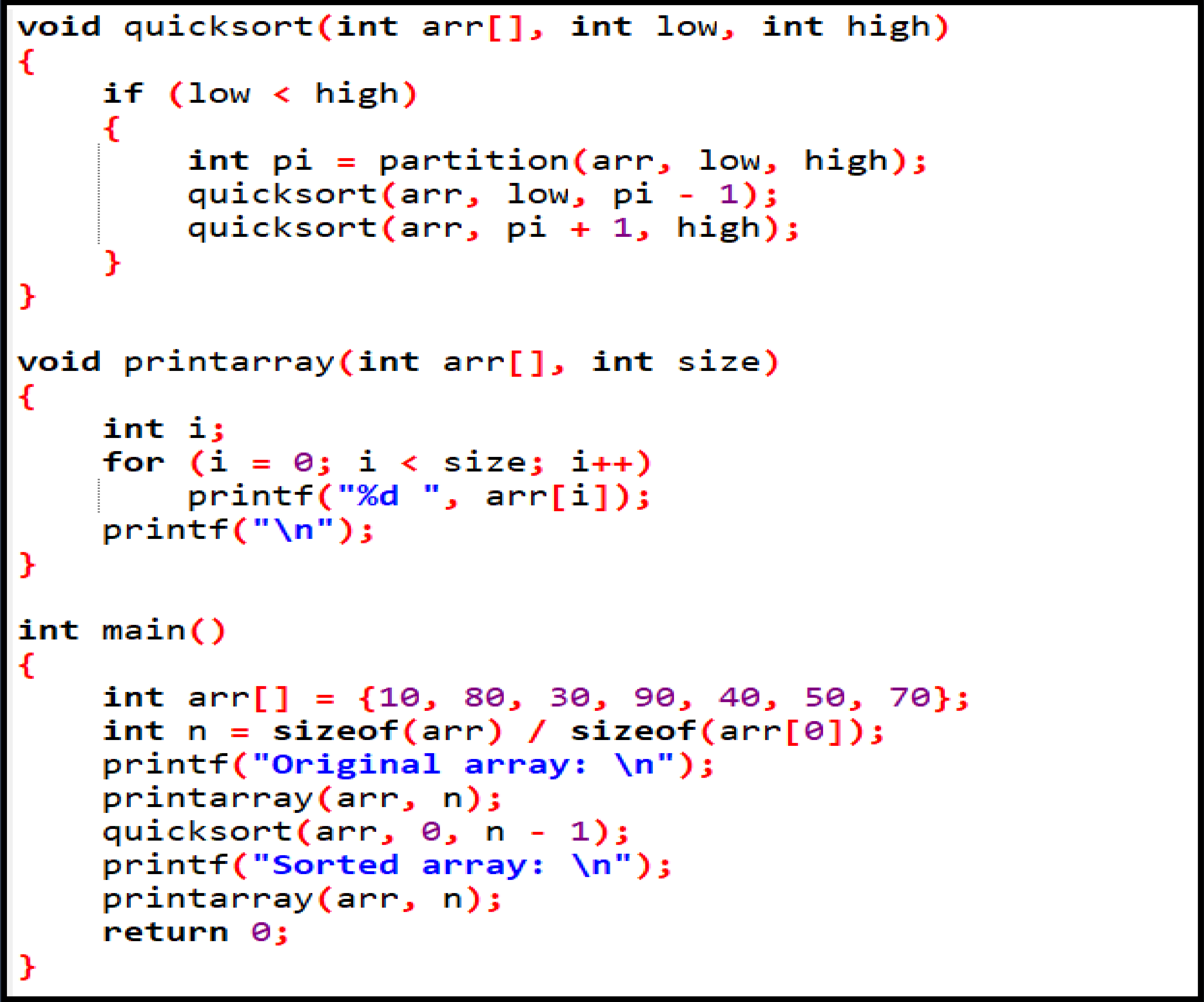
Quick Sort Algorithm



Program

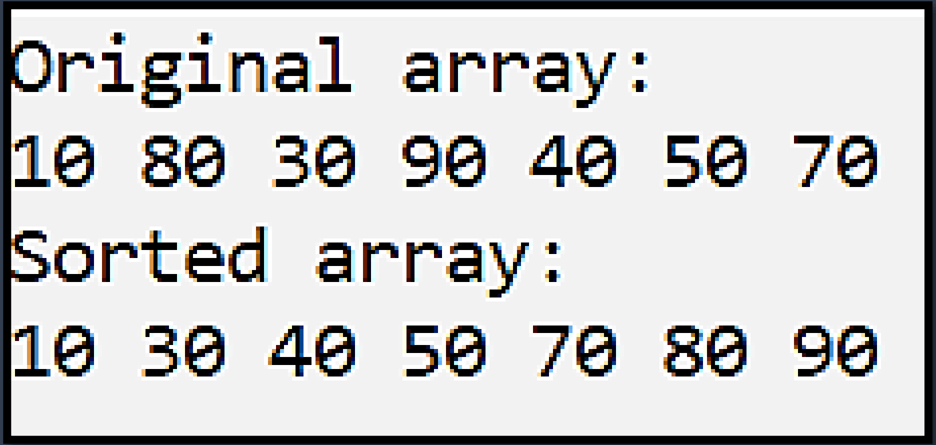


```c
//C Program to implement Quick Sorting
#include <stdio.h>

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

int partition(int arr[], int low, int high)
{
    int pivot = arr[high]; // Choose the rightmost element as pivot
    int i = low - 1;
    int j;

    for (j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```



```
void quicksort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}

void printarray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main()
{
    int arr[] = {10, 80, 30, 90, 40, 50, 70};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: \n");
    printarray(arr, n);
    quicksort(arr, 0, n - 1);
    printf("Sorted array: \n");
    printarray(arr, n);
    return 0;
}
```

Output



```
Original array:
10 80 30 90 40 50 70
Sorted array:
10 30 40 50 70 80 90
```

Advantages of Quick Sorting

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.
- Fastest general purpose algorithm for large data when stability is not required.

Disadvantages of Quick Sorting

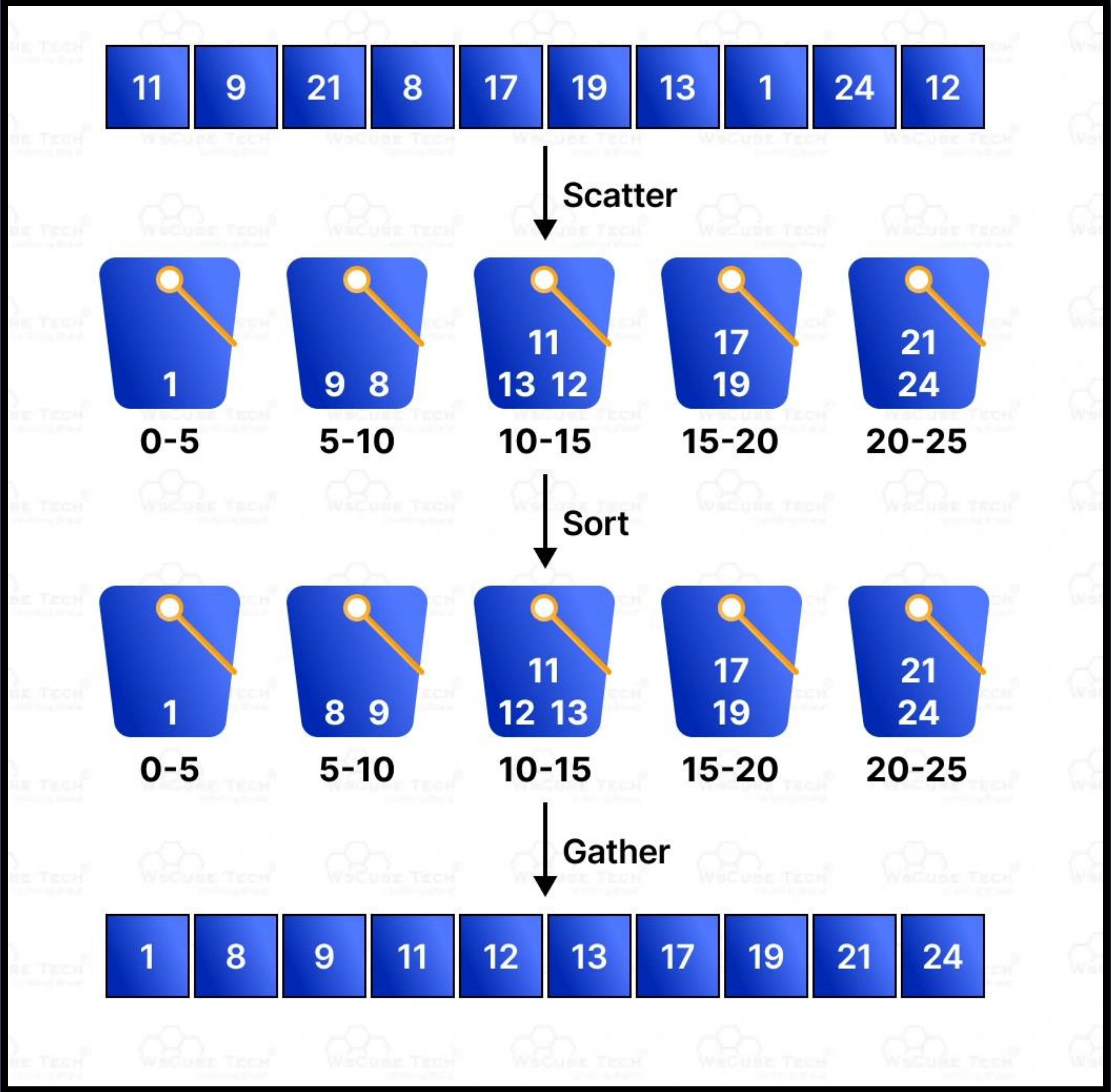
- It has a worst-case time complexity, which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

TOPIC 4

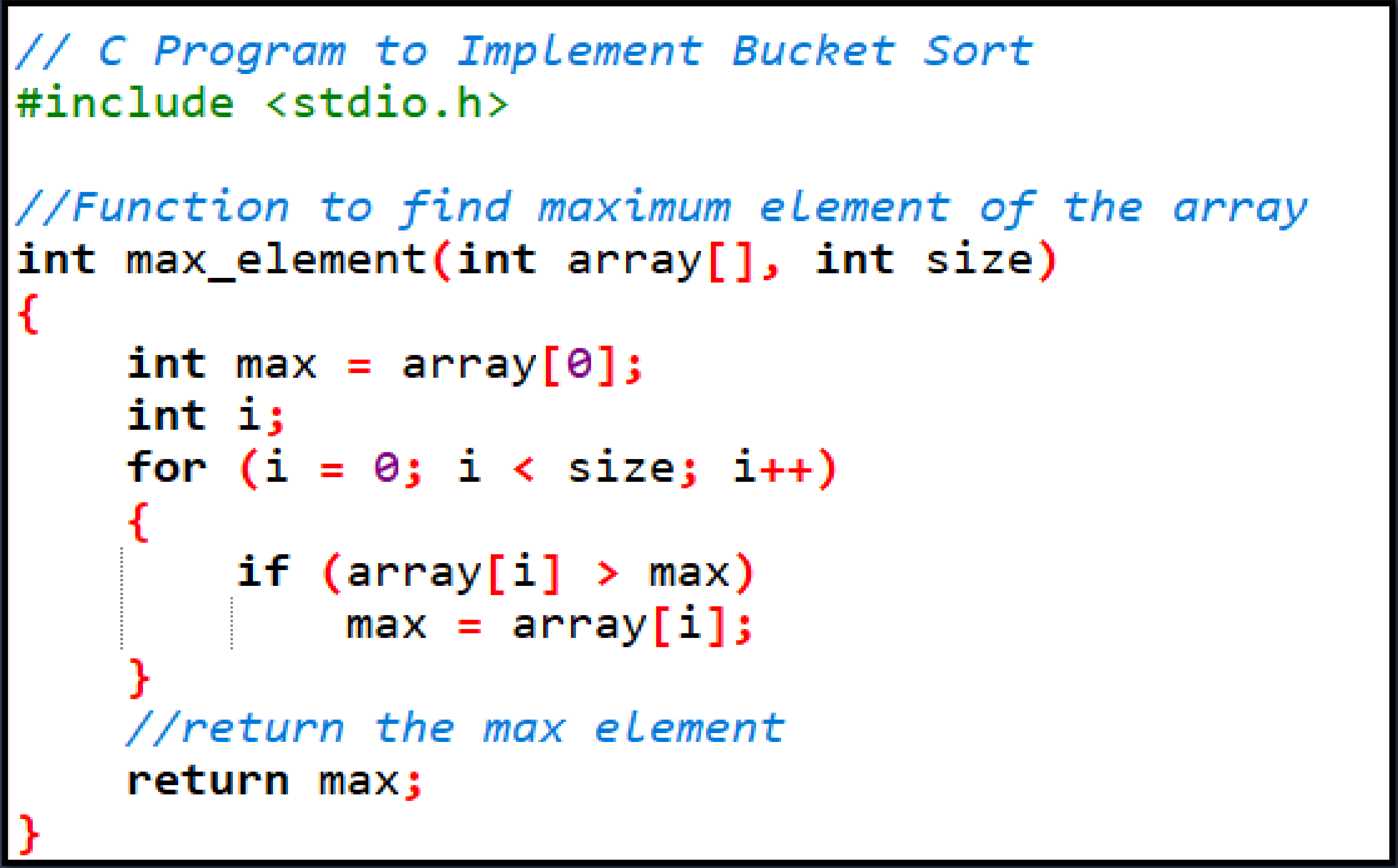
BUCKET SORTING

What is Bucket Sorting?

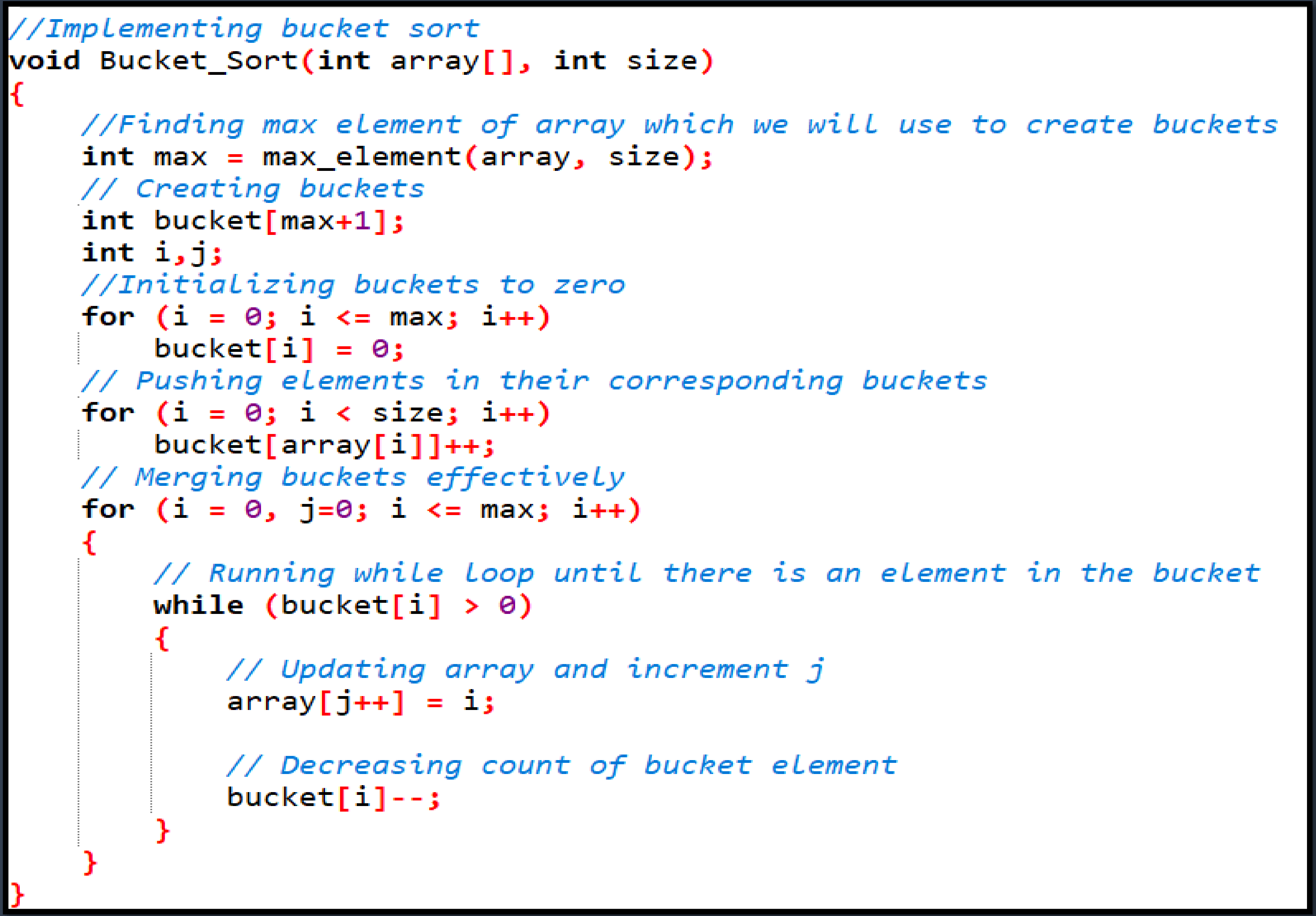
- Bucket sort is a sorting technique that involves dividing elements into various groups, or buckets. These buckets are formed by uniformly distributing the elements.
- Once the elements are divided into buckets, they can be sorted using any other sorting algorithm. Finally, the sorted elements are gathered together in an ordered fashion.
- Works well when the input array elements are uniformly distributed across a range.
- A stable algorithm because we use Insertion Sort (which is stable) to sort the individual buckets.


11 9 21 8 17 19 13 1 24 12
Scatter
1
9 8
11
13 12
17
19
21
24
0-5
5-10
10-15
15-20
20-25
Sort
1
8 9
11
12 13
17
19
21
24
0-5
5-10
10-15
15-20
20-25
Gather
1 8 9 11 12 13 17 19 21 24

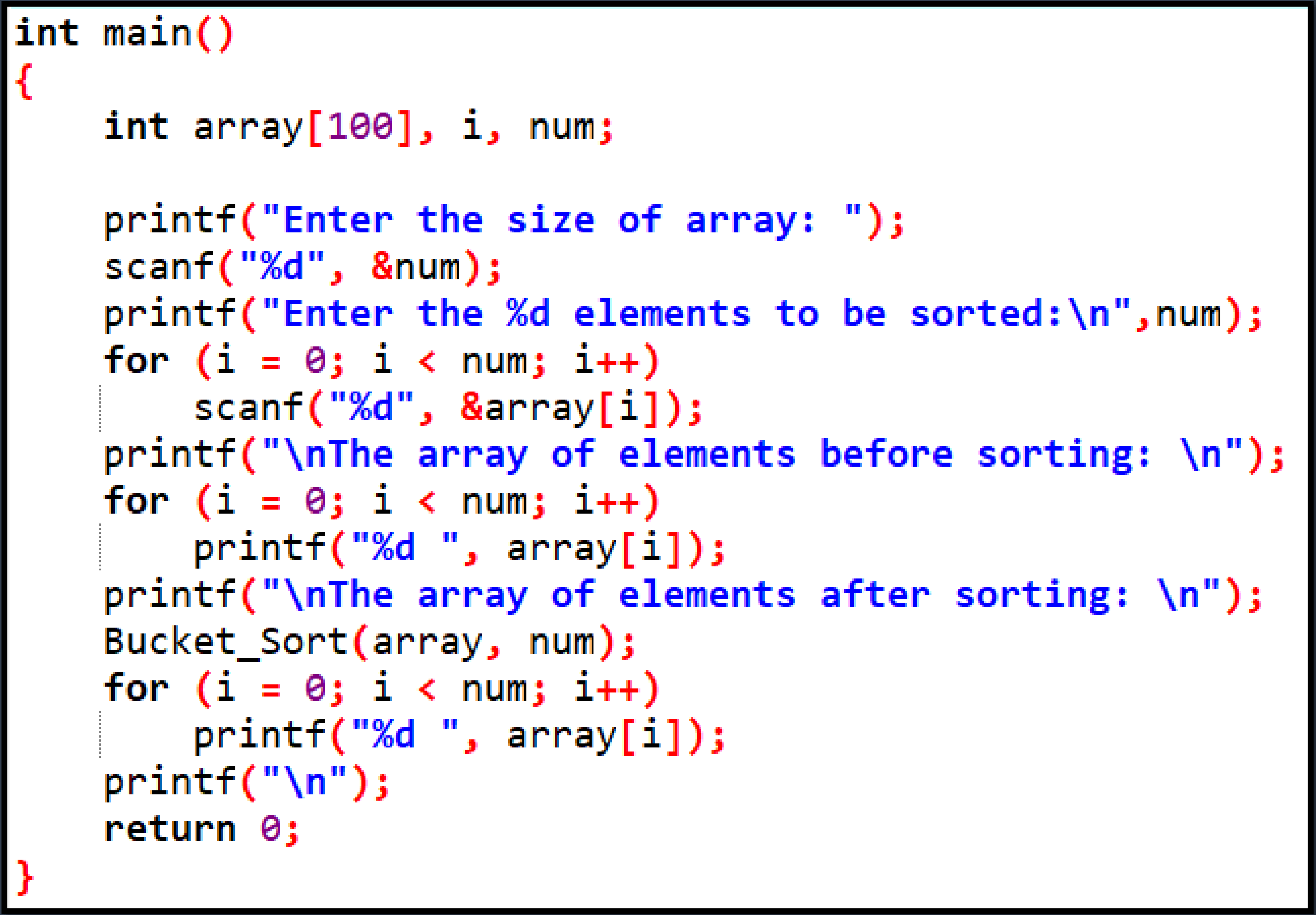
Program



```c
// C Program to Implement Bucket Sort
#include <stdio.h>

//Function to find maximum element of the array
int max_element(int array[], int size)
{
    int max = array[0];
    int i;
    for (i = 0; i < size; i++)
    {
        if (array[i] > max)
            max = array[i];
    }
    //return the max element
    return max;
}
```

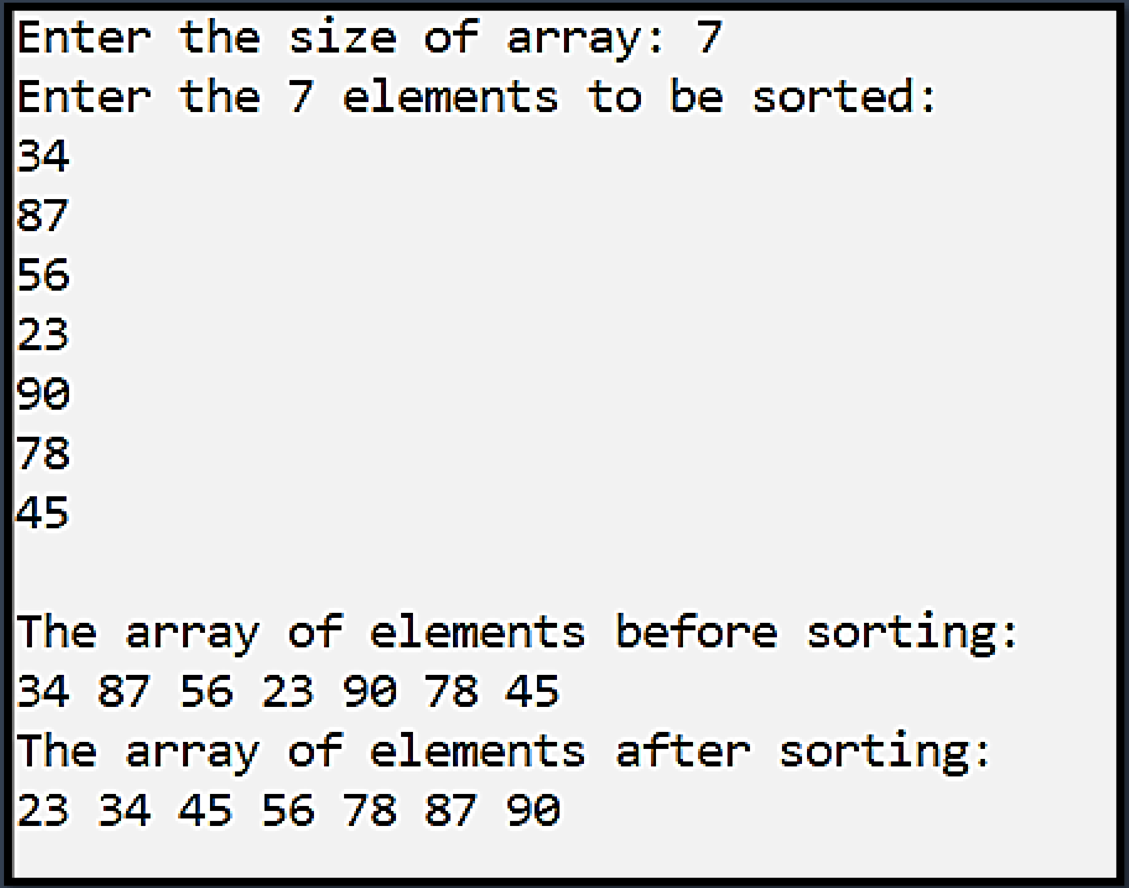


```
//Implementing bucket sort
void Bucket_Sort(int array[], int size)
{
    //Finding max element of array which we will use to create buckets
    int max = max_element(array, size);
    // Creating buckets
    int bucket[max+1];
    int i,j;
    //Initializing buckets to zero
    for (i = 0; i <= max; i++)
        bucket[i] = 0;
    // Pushing elements in their corresponding buckets
    for (i = 0; i < size; i++)
        bucket[array[i]]++;
    // Merging buckets effectively
    for (i = 0, j=0; i <= max; i++)
    {
        // Running while loop until there is an element in the bucket
        while (bucket[i] > 0)
        {
            // Updating array and increment j
            array[j++] = i;

            // Decreasing count of bucket element
            bucket[i]--;
        }
    }
}
```



```c
int main()
{
    int array[100], i, num;

    printf("Enter the size of array: ");
    scanf("%d", &num);
    printf("Enter the %d elements to be sorted:\n",num);
    for (i = 0; i < num; i++)
        scanf("%d", &array[i]);
    printf("\nThe array of elements before sorting: \n");
    for (i = 0; i < num; i++)
        printf("%d ", array[i]);
    printf("\nThe array of elements after sorting: \n");
    Bucket_Sort(array, num);
    for (i = 0; i < num; i++)
        printf("%d ", array[i]);
    printf("\n");
    return 0;
}
```

Output



```
Enter the size of array: 7
Enter the 7 elements to be sorted:
34
87
56
23
90
78
45

The array of elements before sorting:
34 87 56 23 90 78 45
The array of elements after sorting:
23 34 45 56 78 87 90
```

Advantages of Bucket Sorting

- Efficient for sorting elements uniformly distributed over a range.
- Can be faster than comparison-based sorting algorithms for certain types of input data.
- Can be parallelized easily, making it suitable for parallel computing environments.

Disadvantages of Bucket Sorting

- Requires prior knowledge of the data range to determine the bucket size, which may not always be available.
- May not perform well if the input data is not uniformly distributed.
- Extra space is required for creating buckets, which can be a concern for large datasets.