CS -08

DATA STRUCTURE USING

C LANGUAGE

By Rachel

UNIT 2

SORTING AND SEARCHING

TOPIC 5

MERGE SORTING

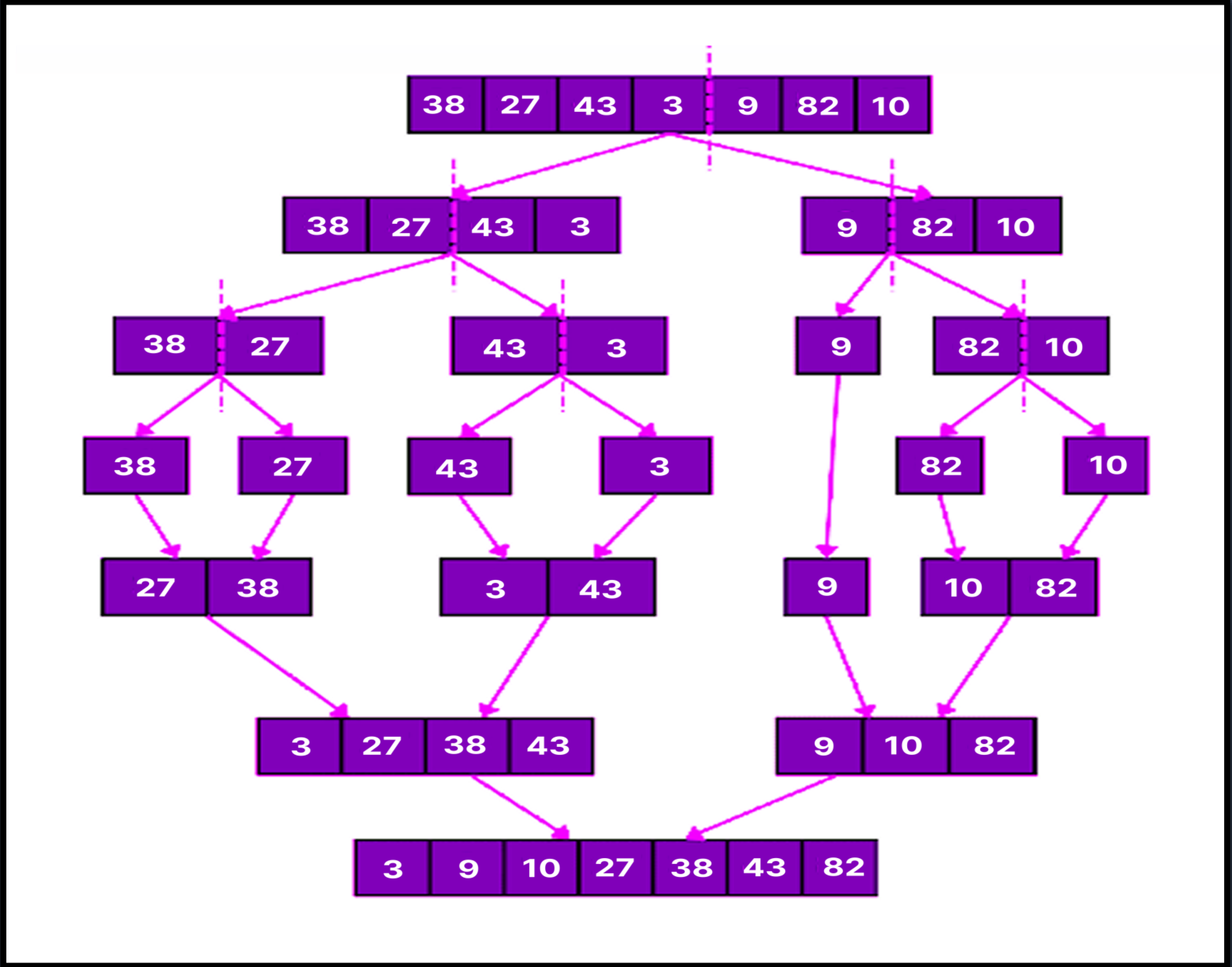
What is Merge Sorting?

- Merge Sort in C is a divide-and-conquer algorithm that efficiently sorts an array by repeatedly splitting it into smaller sections and then merging them in sorted order.
- Because of its efficiency, Merge Sort is usually used for handling large datasets and linked lists.
- Merge Sort is a classic example of the divide and conquer strategy.

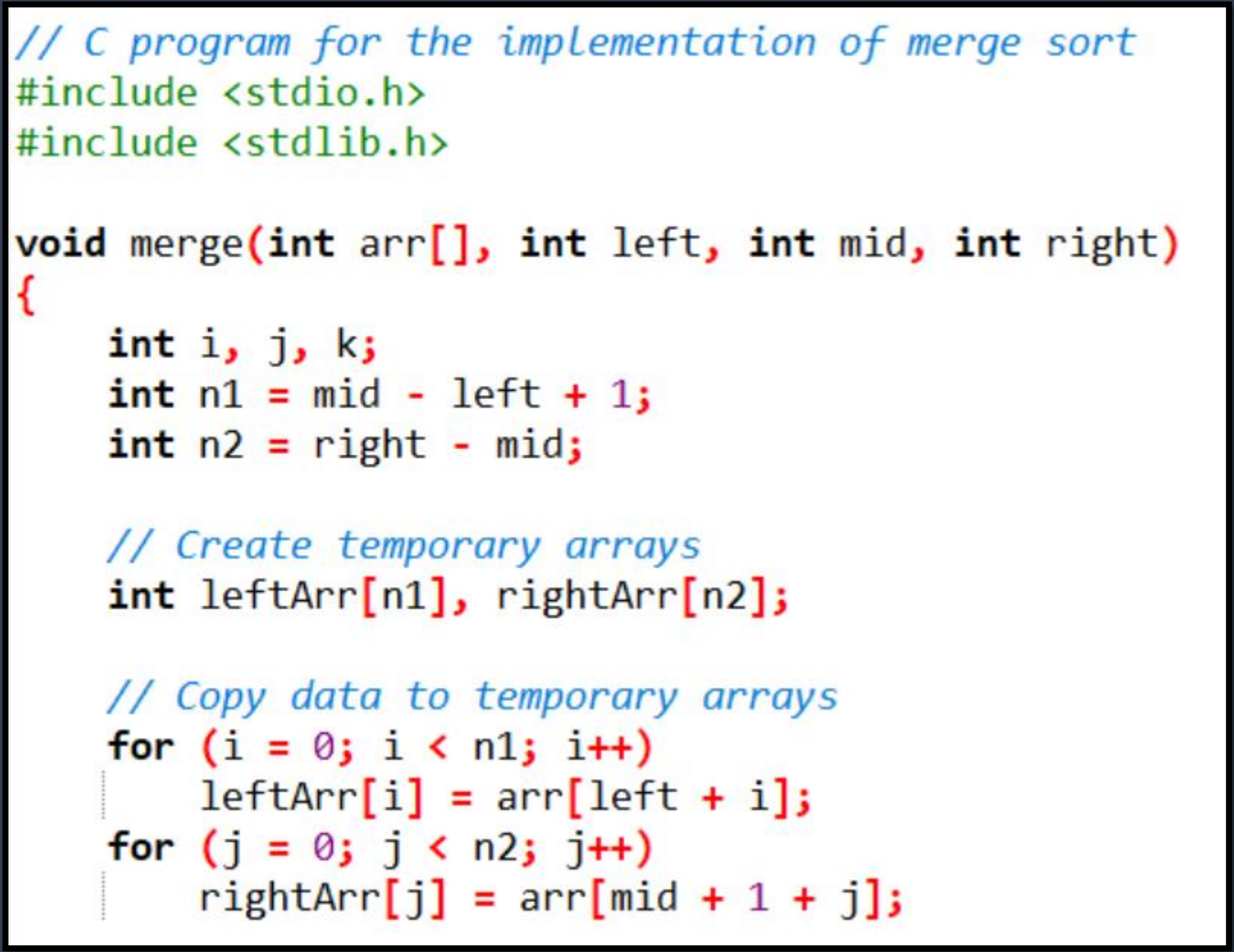
    Divide: The given array is split into two equal (or nearly equal) halves.

    Conquer: Each half is sorted separately using recursion.

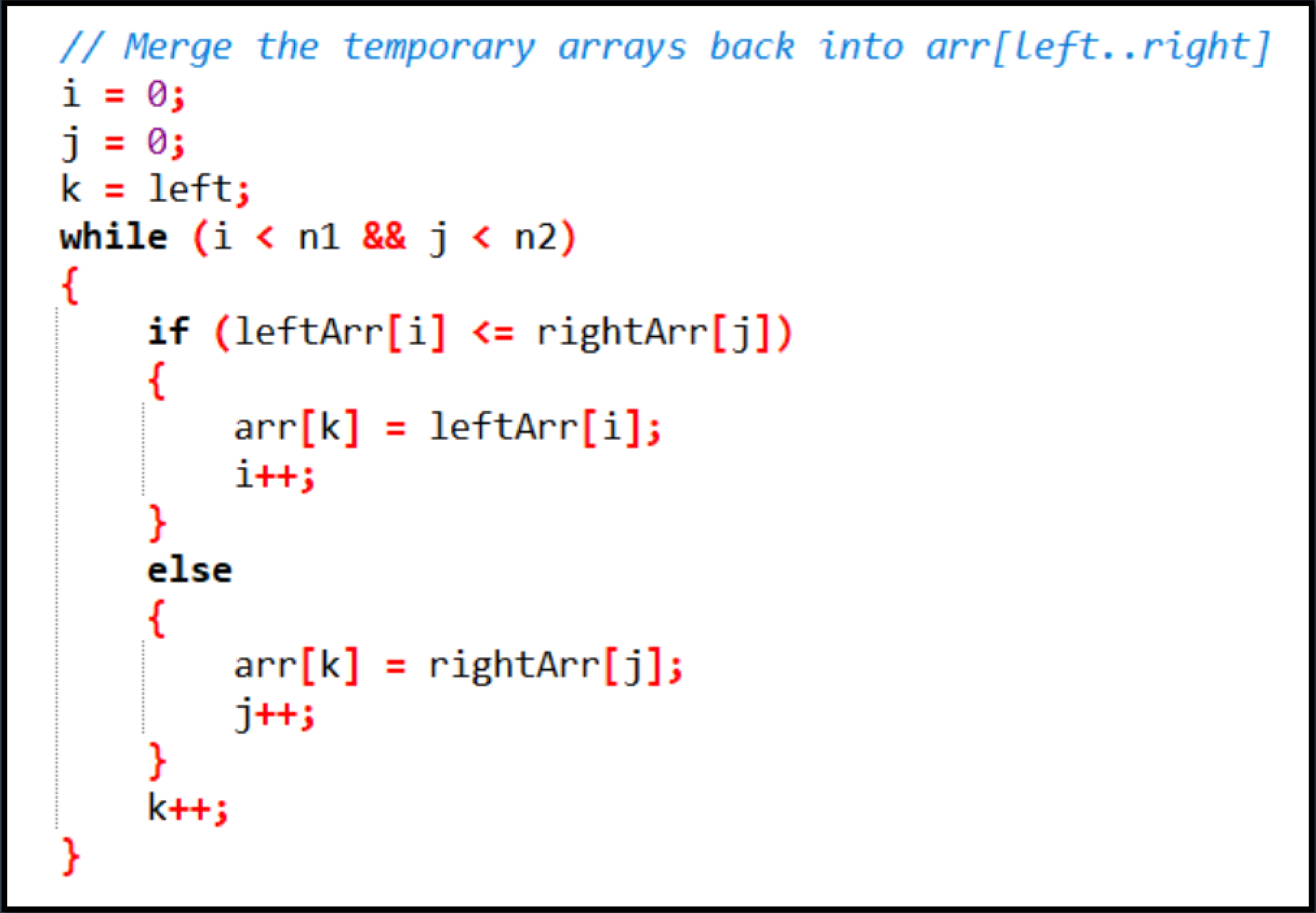
    Combine: The sorted halves are merged to form a single, fully sorted array.


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9
82 10
38
27
43
3
82
10
27 38
3 43
9
10 82
3 27 38 43
9 10 82
3 9 10 27 38 43 82

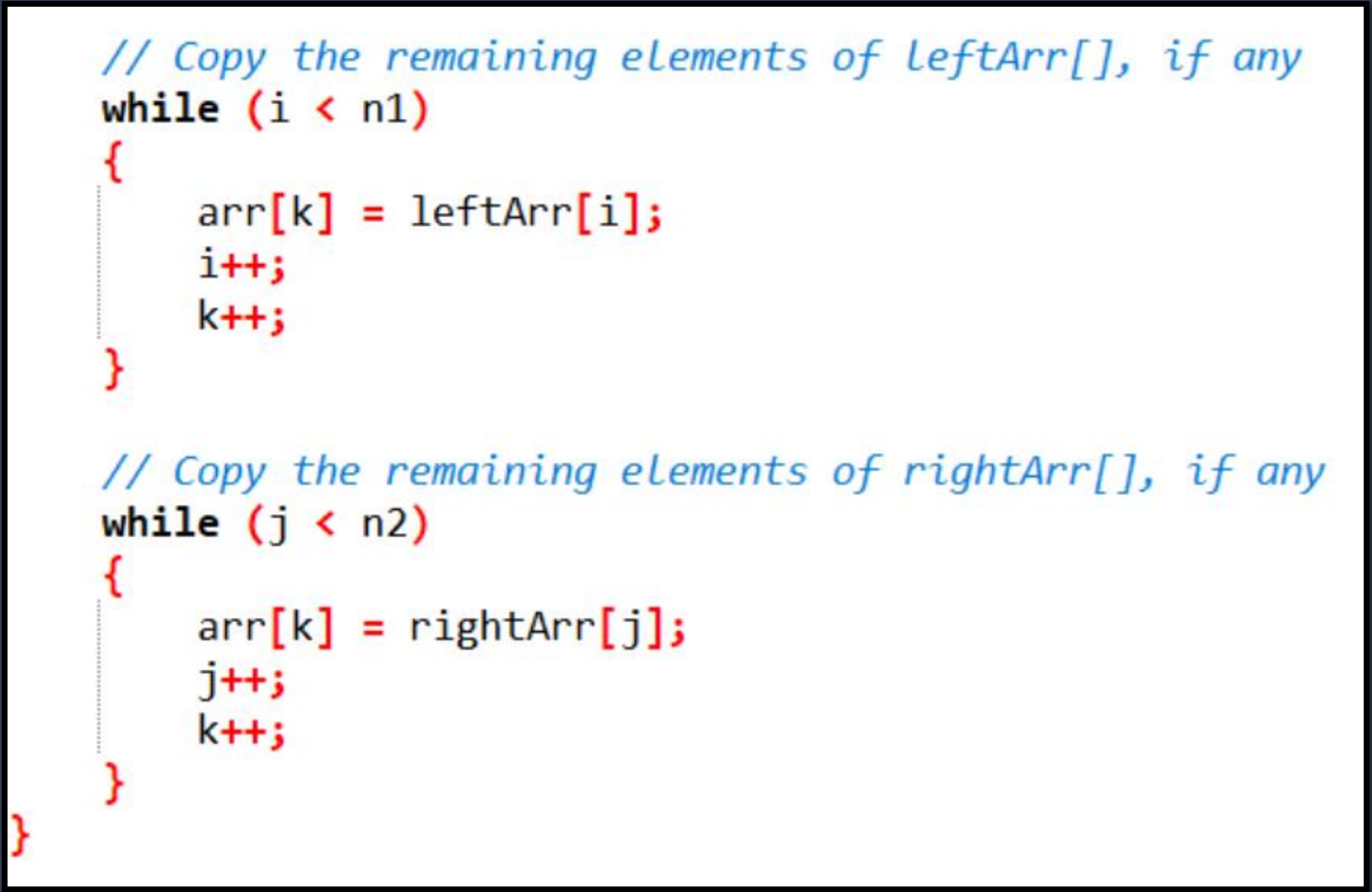
Program



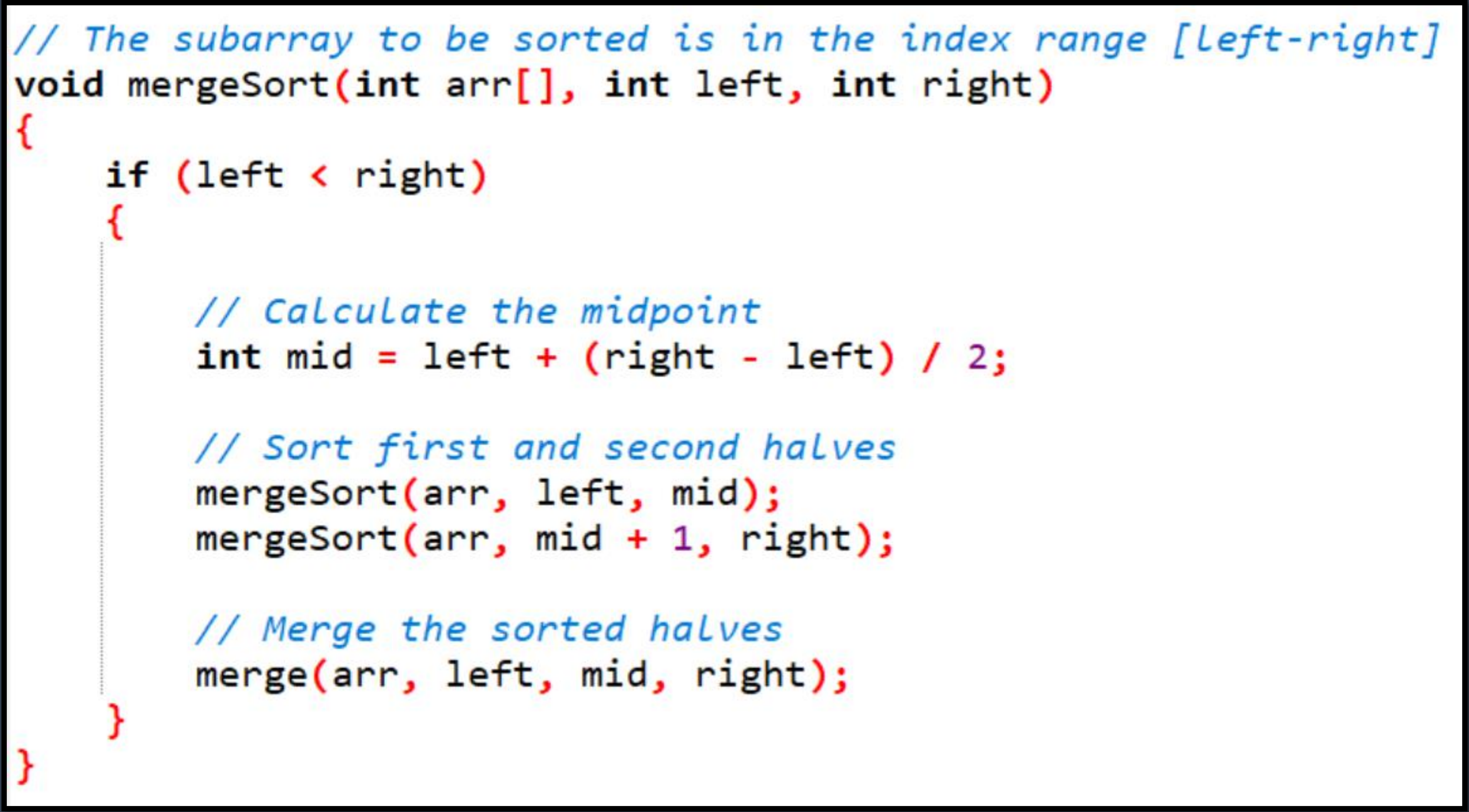
```
// C program for the implementation of merge sort
#include <stdio.h>
#include <stdlib.h>

void merge(int arr[], int left, int mid, int right)
{
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    int leftArr[n1], rightArr[n2];

    // Copy data to temporary arrays
    for (i = 0; i < n1; i++)
        leftArr[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        rightArr[j] = arr[mid + 1 + j];
```

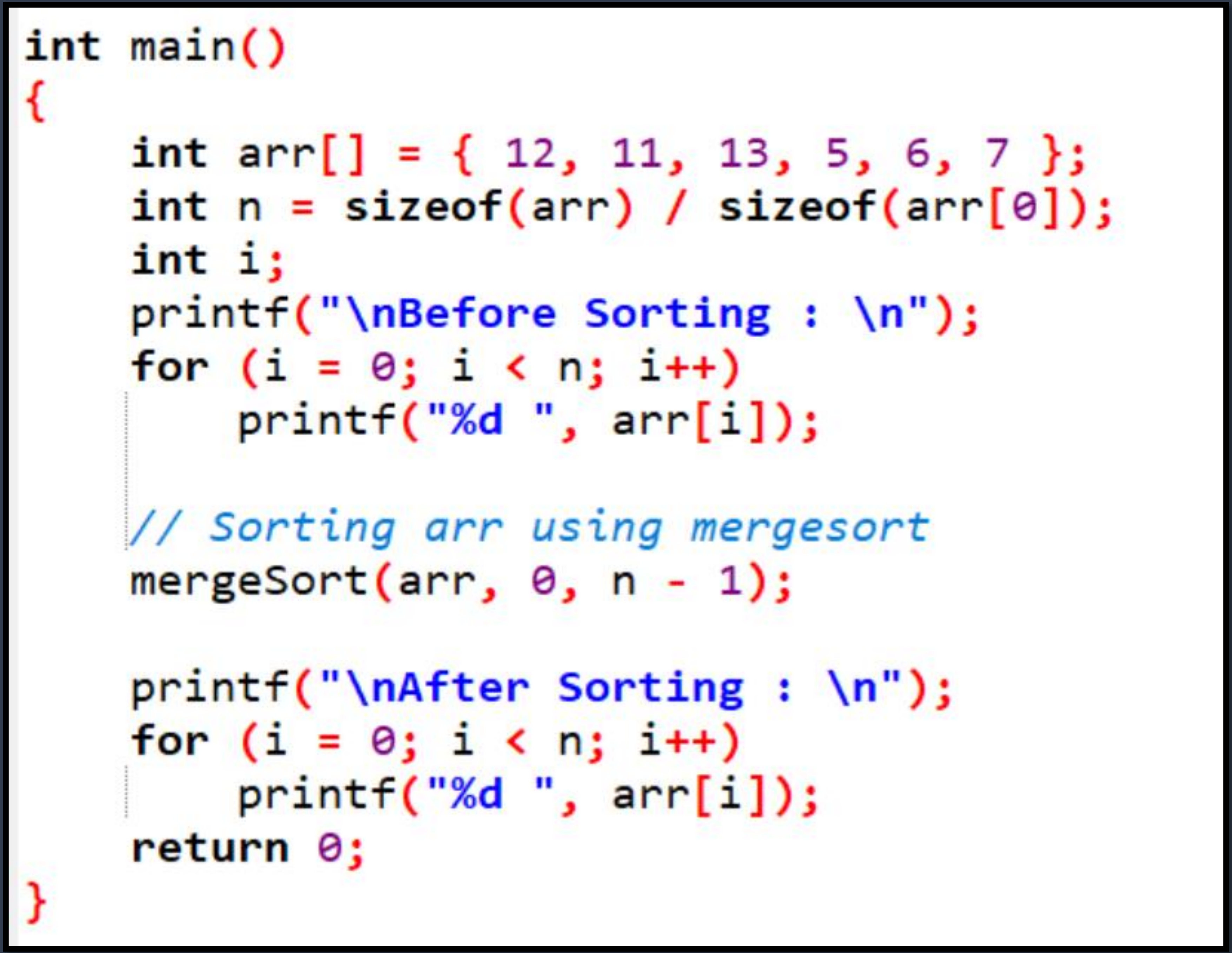


```
// Merge the temporary arrays back into arr[left..right]
i = 0;
j = 0;
k = left;
while (i < n1 && j < n2)
{
    if (leftArr[i] <= rightArr[j])
    {
        arr[k] = leftArr[i];
        i++;
    }
    else
    {
        arr[k] = rightArr[j];
        j++;
    }
    k++;
}
```



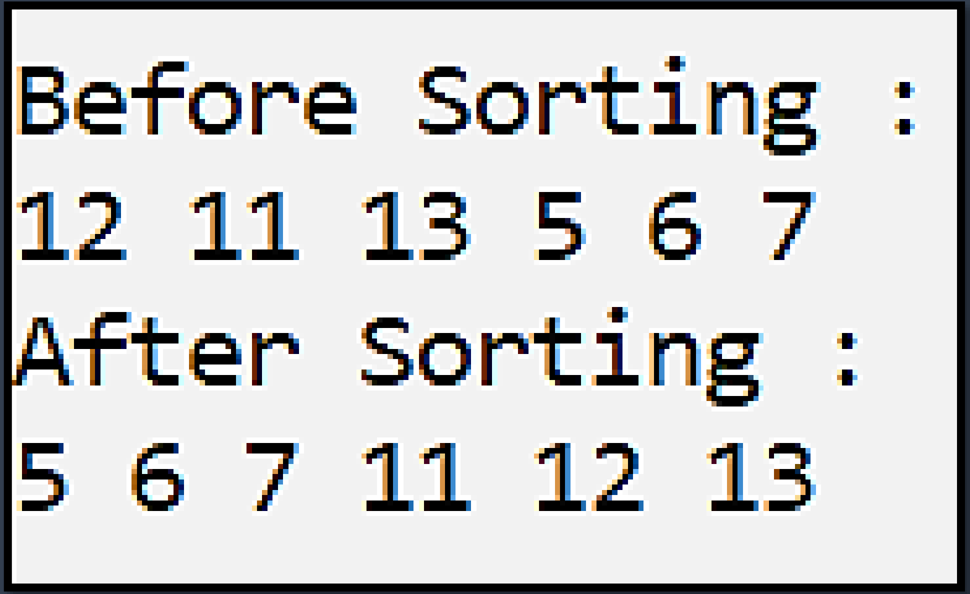
```
    // Copy the remaining elements of leftArr[], if any
    while (i < n1)
    {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    // Copy the remaining elements of rightArr[], if any
    while (j < n2)
    {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
```


```
// The subarray to be sorted is in the index range [left-right]
void mergeSort(int arr[], int left, int right)
{
    if (left < right)
    {

        // Calculate the midpoint
        int mid = left + (right - left) / 2;

        // Sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}
```



```
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int i;
    printf("\nBefore Sorting : \n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    // Sorting arr using mergesort
    mergeSort(arr, 0, n - 1);

    printf("\nAfter Sorting : \n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Output



```
Before Sorting :
12 11 13 5 6 7
After Sorting :
5 6 7 11 12 13
```

Advantages of Merge Sorting

- Stability : Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- Guaranteed worst-case performance: Merge sort has a worst-case time complexity of O(n log n) , which means it performs well even on large datasets.
- Simple to implement: The divide-and-conquer approach is straightforward.
- Naturally Parallel : We independently merge subarrays that makes it suitable for parallel processing.

Disadvantages of Merge Sorting

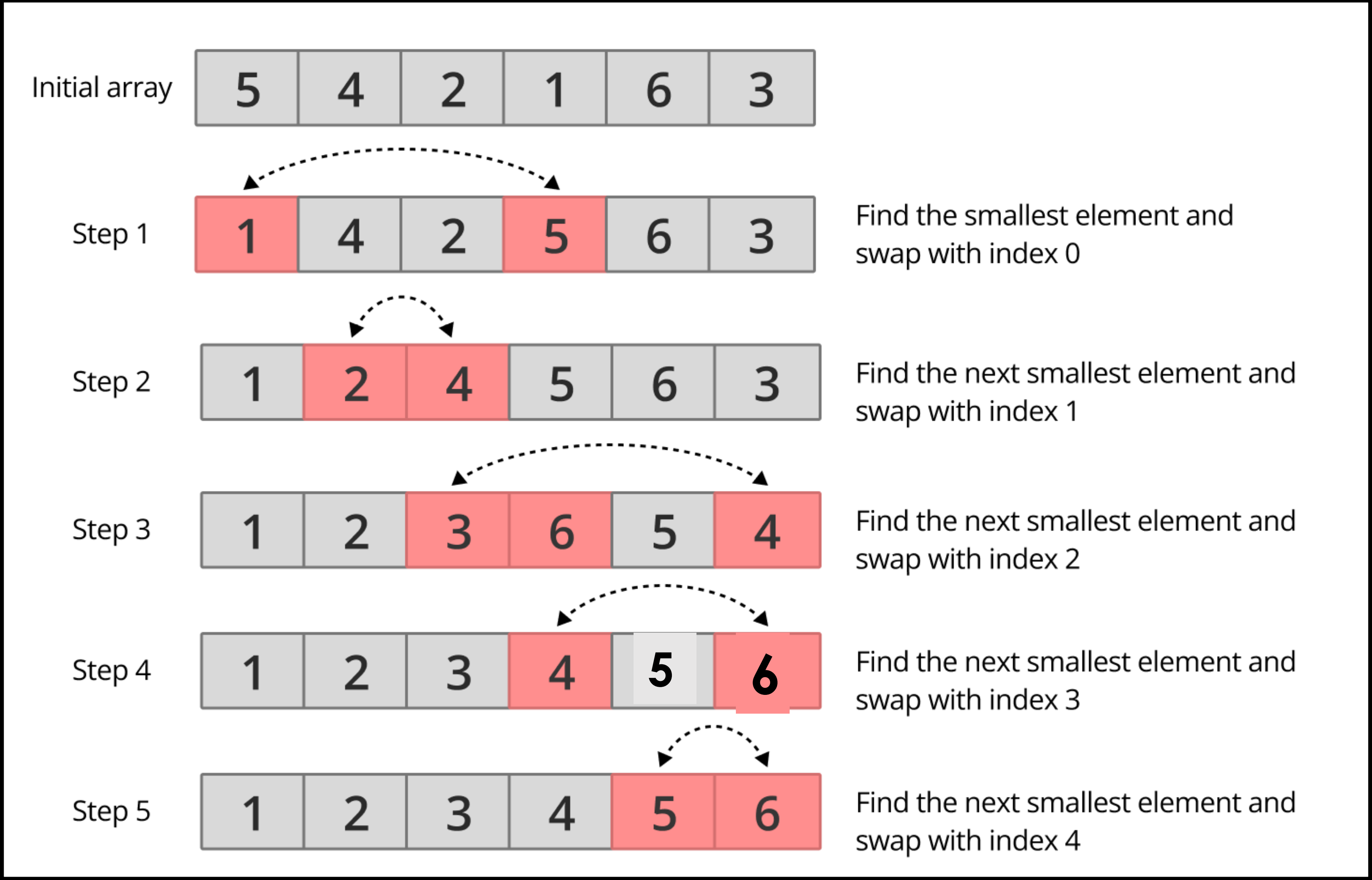
- Space complexity: Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- Not in-place: Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- Merge Sort is Slower than Quick Sort in general.

TOPIC 6

SELECTION SORTING

What is Selection Sorting?

- Selection Sort is a comparison-based sorting algorithm to sort the given input in ascending or descending order.
- In this sorting technique, we will find the smallest element from the array and place it in the first position of the array. After that, we will take the second smallest number from the array and place it in the second position of the array in this way we will sort the entire array in ascending order.



Find the smallest element and swap with index 0

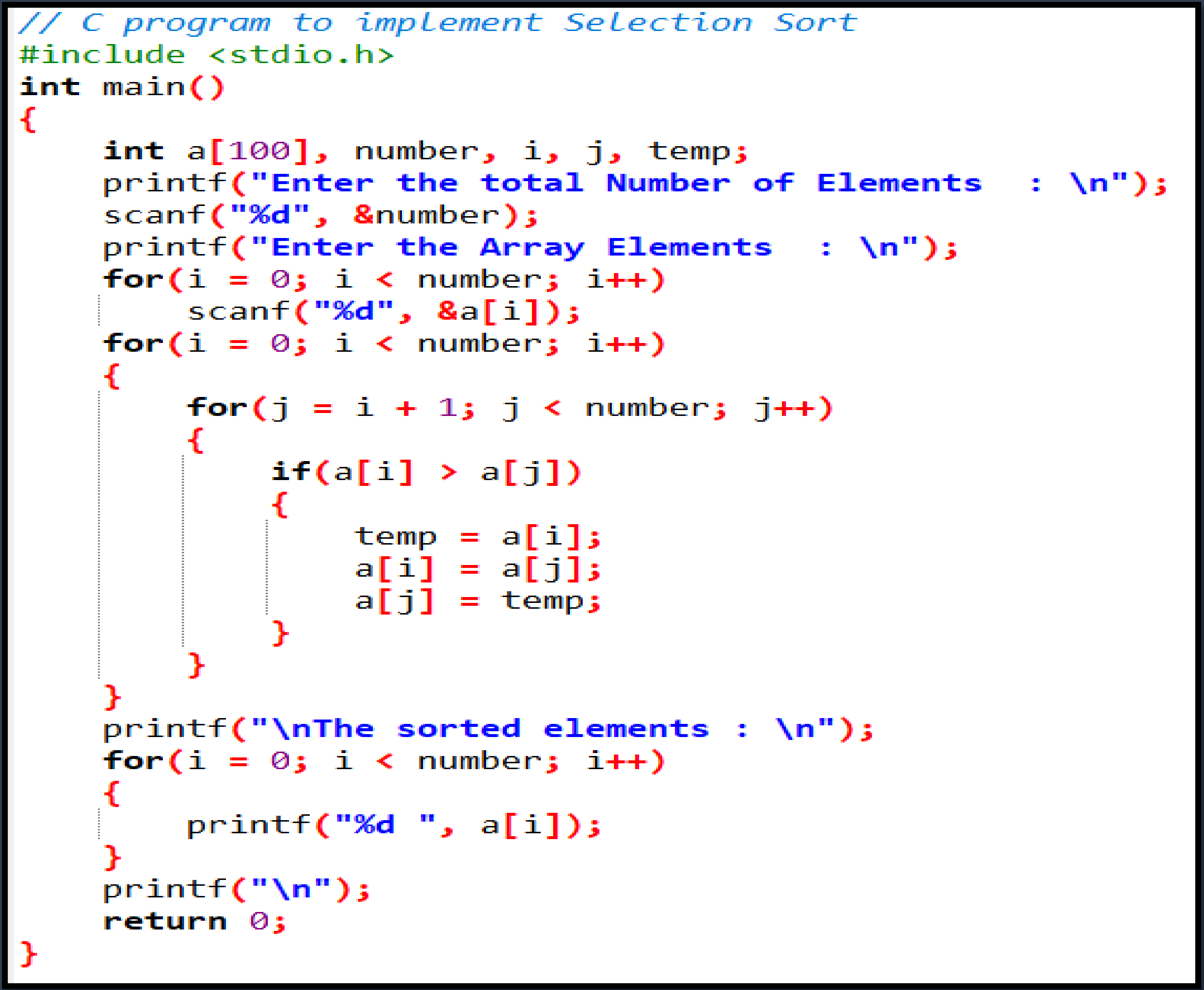
Find the next smallest element and swap with index 1

Find the next smallest element and swap with index 2

Find the next smallest element and swap with index 3

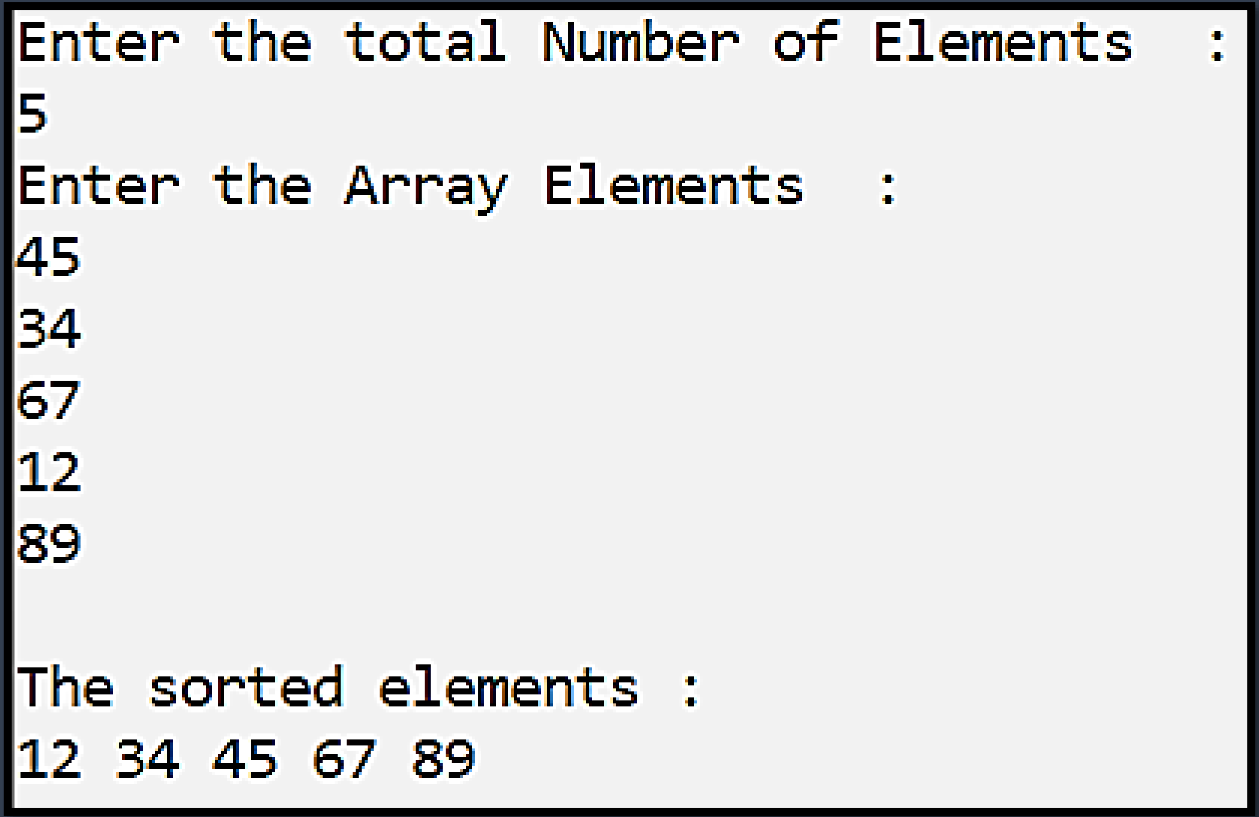
Find the next smallest element and swap with index 4

Program



```c
// C program to implement Selection Sort
#include <stdio.h>
int main()
{
    int a[100], number, i, j, temp;
    printf("Enter the total Number of Elements  : \n");
    scanf("%d", &number);
    printf("Enter the Array Elements  : \n");
    for(i = 0; i < number; i++)
        scanf("%d", &a[i]);
    for(i = 0; i < number; i++)
    {
        for(j = i + 1; j < number; j++)
        {
            if(a[i] > a[j])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    printf("\nThe sorted elements : \n");
    for(i = 0; i < number; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");
    return 0;
}
```

Output



```
Enter the total Number of Elements   :
5
Enter the Array Elements   :
45
34
67
12
89

The sorted elements :
12 34 45 67 89
```

Advantages of Selection Sorting

- Easy to understand and implement, making it ideal for teaching basic sorting concepts.
- Requires only a constant O(1) extra memory space.
- It requires less number of swaps (or memory writes) compared to many other standard algorithms.

Disadvantages of Selection Sorting

- Selection sort has a time complexity of $O(n^2)$ makes it slower compared to algorithms like Quick Sort or Merge Sort.
- Does not maintain the relative order of equal elements which means it is not stable.

TOPIC 7

SHELL SORTING

What is Shell Sorting?

- Shell Sort works by comparing elements that are far apart first, then gradually reducing the gap. This allows faster movement of elements across the array.
- Here’s the step-by-step explanation:
  1. Choose a gap sequence (commonly n/2, n/4, ... , 1).
  2. Sort elements at each gap using Insertion Sort.
  3. Reduce the gap and repeat until the gap becomes 1.

01 | Initial Array & Gap Sequence

arr[] =

22	34	25	12	64	11	90	88	45
0	1	2	3	4	5	6	7	8

Gap Sequence:

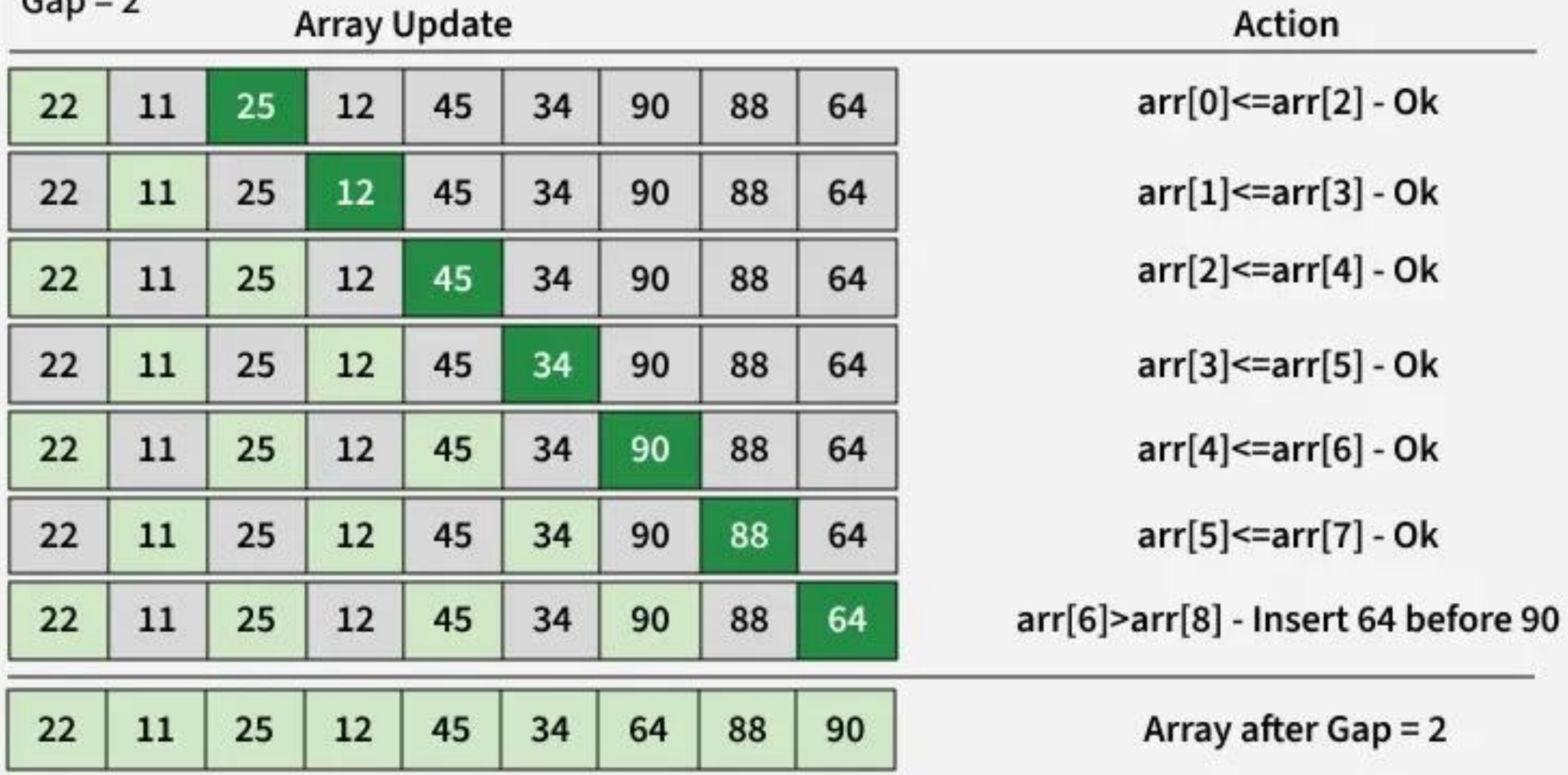
Array size (n) = 9

1. First gap = n/2 = 4
2. Second gap = n/4 = 2
3. Third gap = n/8 = 1

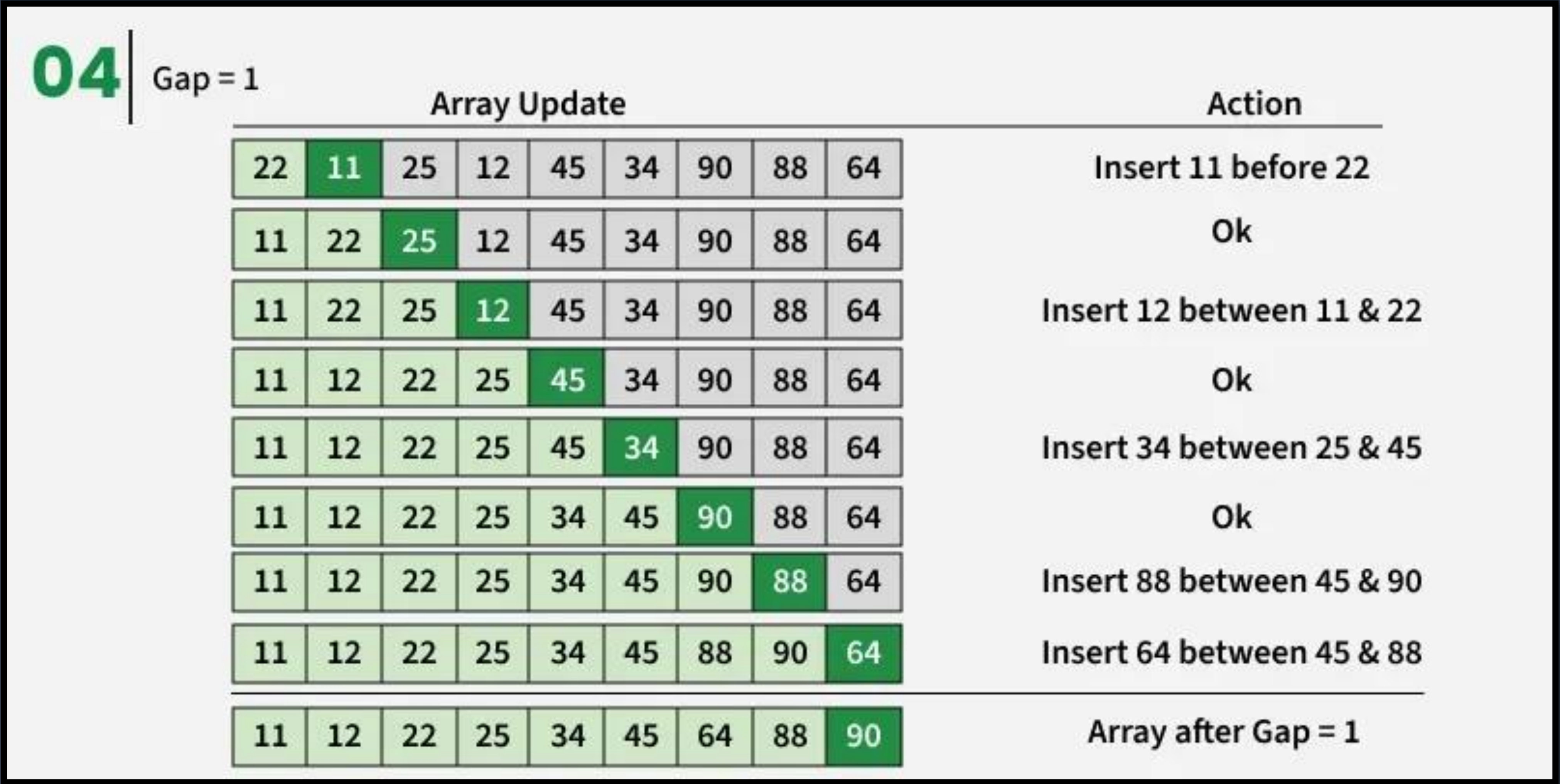
02 | Gap = 4

Array Update									Action
22	34	25	12	64	11	90	88	45	arr[0]<=arr[4] - Ok
22	34	25	12	64	11	90	88	45	arr[1]>arr[5] - Insert 11 before 34
22	11	25	12	64	34	90	88	45	arr[2]<=arr[6] - Ok
22	11	25	12	64	34	90	88	45	arr[3]<=arr[7] - Ok
22	11	25	12	64	34	90	88	45	arr[4]>arr[8] - Insert 45 before 64
22	11	25	12	45	34	90	88	64	Array after Gap = 4

03 | Gap = 2



Array Update									Action
22	11	25	12	45	34	90	88	64	arr[0]<=arr[2] - Ok
22	11	25	12	45	34	90	88	64	arr[1]<=arr[3] - Ok
22	11	25	12	45	34	90	88	64	arr[2]<=arr[4] - Ok
22	11	25	12	45	34	90	88	64	arr[3]<=arr[5] - Ok
22	11	25	12	45	34	90	88	64	arr[4]<=arr[6] - Ok
22	11	25	12	45	34	90	88	64	arr[5]<=arr[7] - Ok
22	11	25	12	45	34	90	88	64	arr[6]>arr[8] - Insert 64 before 90
22	11	25	12	45	34	64	88	90	Array after Gap = 2



04 | Gap = 1

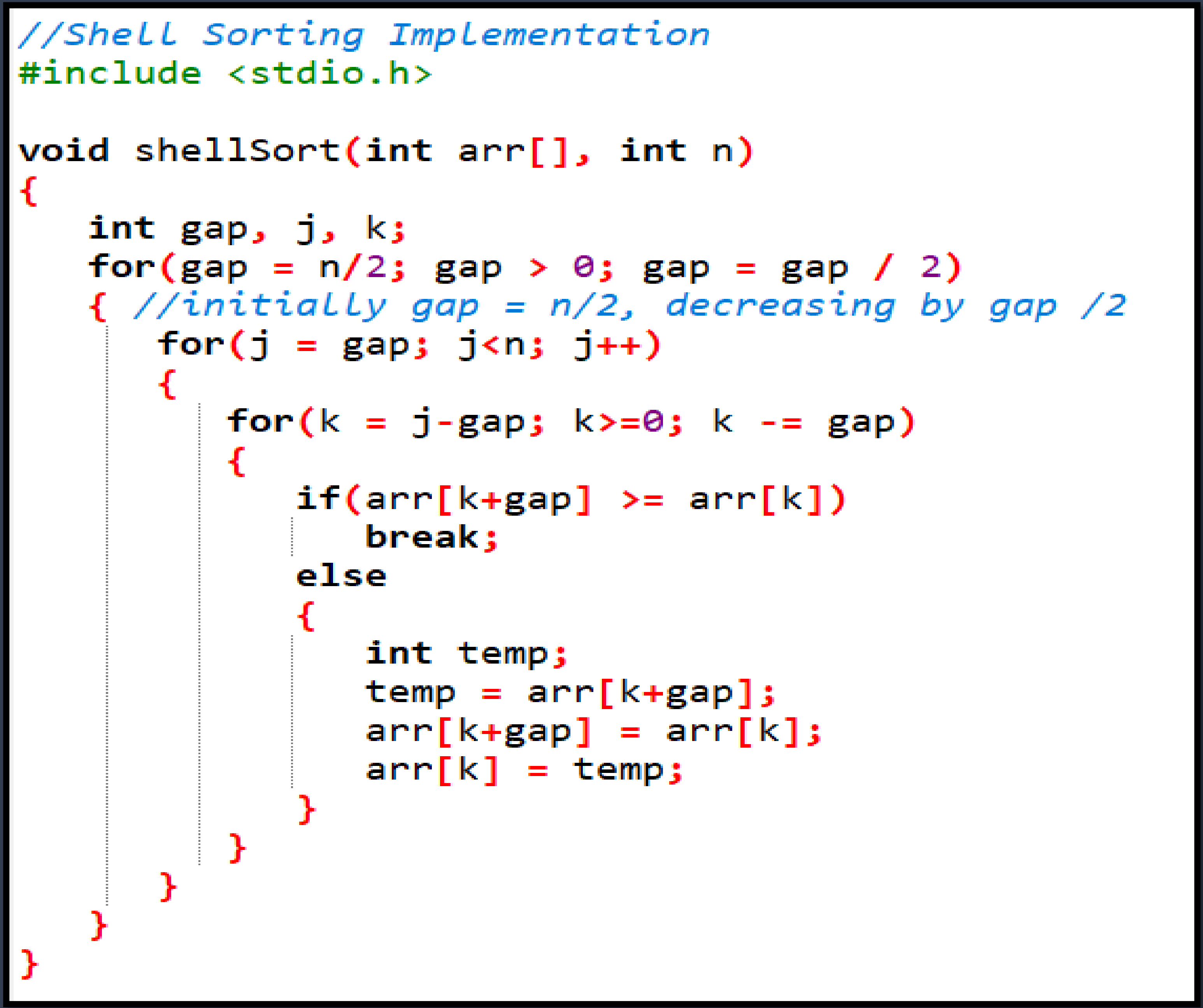
Array Update									Action
22	11	25	12	45	34	90	88	64	Insert 11 before 22
11	22	25	12	45	34	90	88	64	Ok
11	22	25	12	45	34	90	88	64	Insert 12 between 11 & 22
11	12	22	25	45	34	90	88	64	Ok
11	12	22	25	45	34	90	88	64	Insert 34 between 25 & 45
11	12	22	25	34	45	90	88	64	Ok
11	12	22	25	34	45	90	88	64	Insert 88 between 45 & 90
11	12	22	25	34	45	88	90	64	Insert 64 between 45 & 88
11	12	22	25	34	45	64	88	90	Array after Gap = 1

05 Final Sorted Array

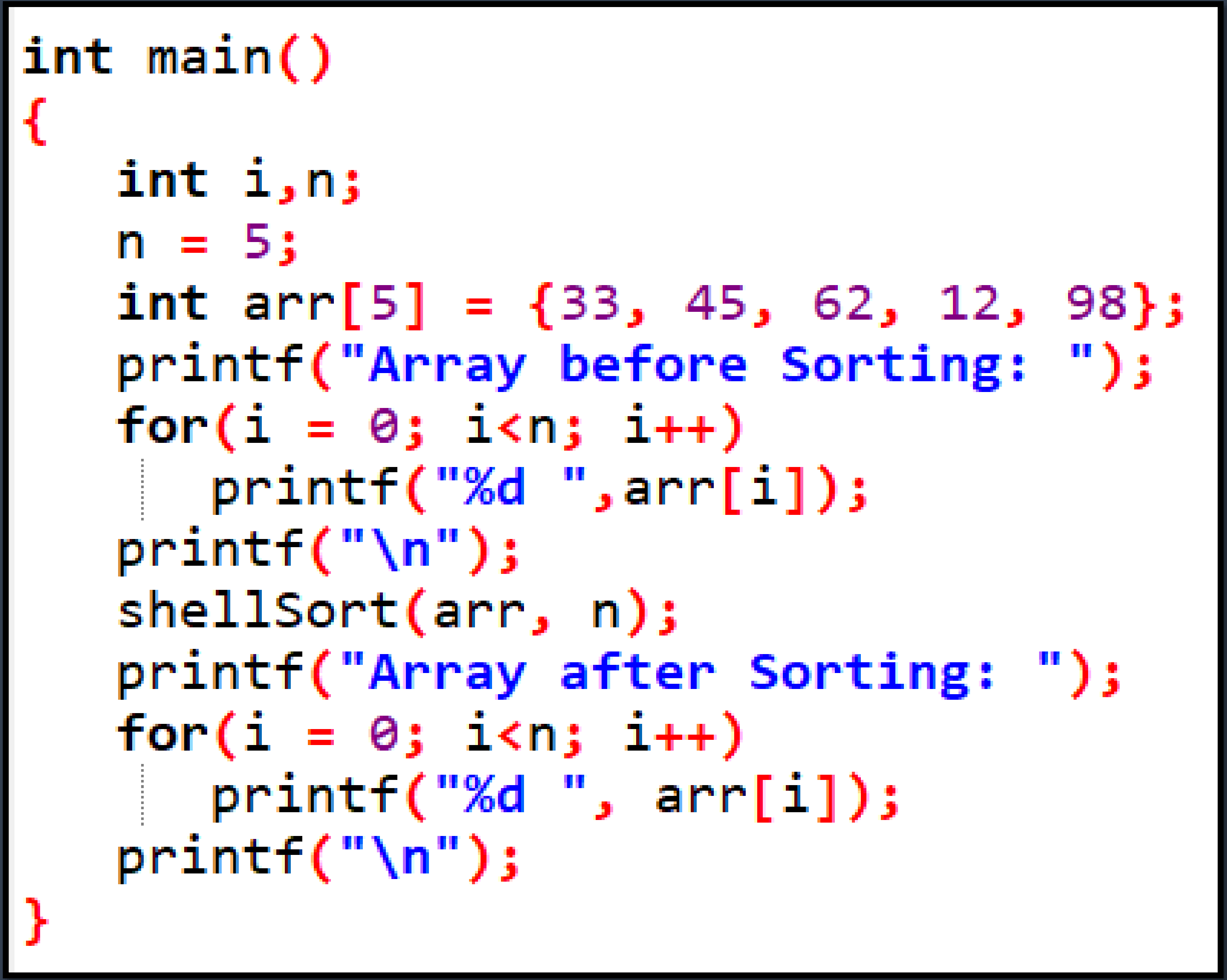
Since, gap size = 1. Hence, array is now completely sorted using Shell Sort:

arr[] =	11	12	22	25	34	45	64	88	90

Program

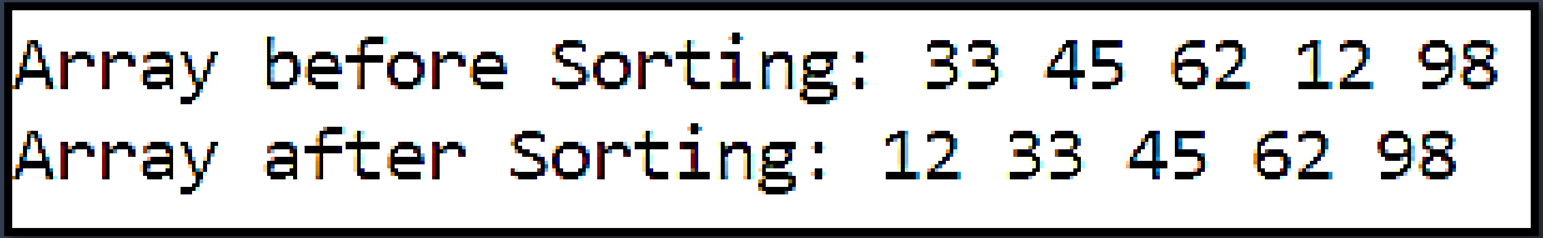


```c
//Shell Sorting Implementation
#include <stdio.h>

void shellSort(int arr[], int n)
{
    int gap, j, k;
    for(gap = n/2; gap > 0; gap = gap / 2)
    { //initially gap = n/2, decreasing by gap /2
        for(j = gap; j<n; j++)
        {
            for(k = j-gap; k>=0; k -= gap)
            {
                if(arr[k+gap] >= arr[k])
                    break;
                else
                {
                    int temp;
                    temp = arr[k+gap];
                    arr[k+gap] = arr[k];
                    arr[k] = temp;
                }
            }
        }
    }
}
```



```c
int main()
{
    int i,n;
    n = 5;
    int arr[5] = {33, 45, 62, 12, 98};
    printf("Array before Sorting: ");
    for(i = 0; i<n; i++)
        printf("%d ",arr[i]);
    printf("\n");
    shellSort(arr, n);
    printf("Array after Sorting: ");
    for(i = 0; i<n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

Output



```
Array before Sorting: 33 45 62 12 98
Array after Sorting: 12 33 45 62 98
```

Advantages of Shell Sorting

- In-place Sorting: Requires only O(1) extra space.
- Faster than Insertion Sort: Especially for larger arrays.
- Easy to Implement: Straightforward improvement over Insertion Sort.

Disadvantages of Shell Sorting

- Not Stable: Does not maintain the order of equal elements.
- Performance depends on gap sequence: Bad choices can lead to $O(n^2)$.
- Slower than advanced algorithms: Merge Sort and Quick Sort usually outperform it on large datasets.