جامـــعة المـــستـقبـل
AL MUSTAQBAL UNIVERSITY

قســـم الأمــــــــن الـــــــــــسيبرانـــــــــي

DEPARTMENT OF CYBER SECURITY

SUBJECT:

SEARCHING AND SORTING ALGORITHMS

CLASS:

SECOND

LECTURER: M.SC.MUNTATHER AL-MUSSAWEE

LECTURE: (2)

SORTING METHODS

Bubble Sort

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., X[i] with X[i+1] and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

❖ EXAMPLE:

Consider the array x[n] which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements).

We compare X[i] and X[i+1] for i = 0, 1, 2, 3, and 4, and interchange **X[i]** and **X[i+1]** **if X[i] > X[i+1]**. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

<u>Pass 2</u>: (second element is compared).

i.e., we compare **X[i]** with **X[i+1]** for i=0, 1, 2, and 3 and interchange **X[i]** and **X[i+1]** **if X[i] > X[i+1]**. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks
33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to **X[4]**.

<u>Pass 3:</u> (third element is compared).

We repeat the same process, but this time we leave both **X[4]** and **X[5]**. By doing this, we move the third biggest number 44 to **X[3]**.

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared).

We repeat the process leaving **X[3], X[4],** and **X[5]**. By doing this, we move the fourth biggest number 33 to **X[2]**.

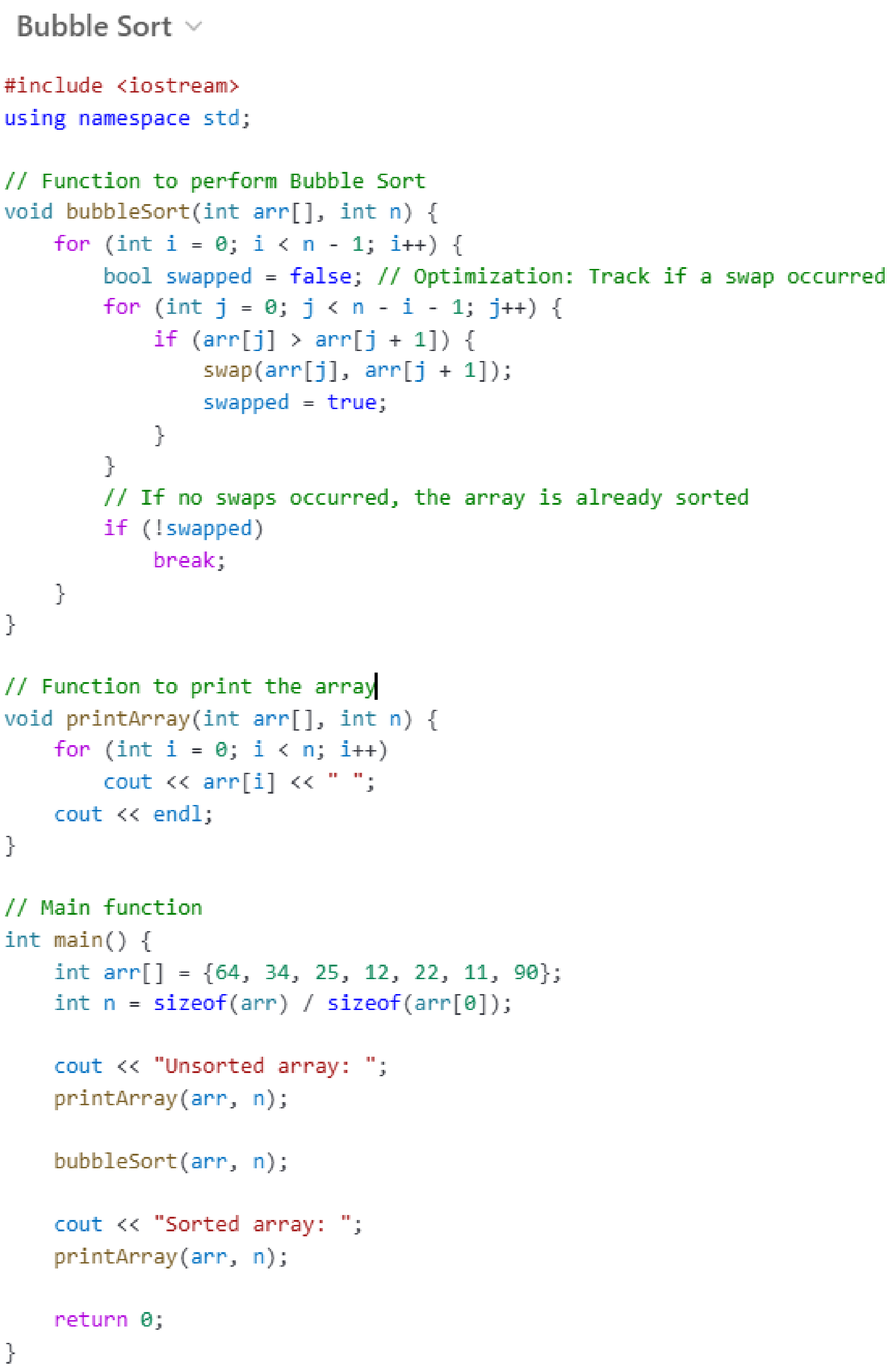
X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22	33	

Pass 5: (fifth element is compared).

We repeat the process leaving **X[2], X[3], X[4],** and **X[5]**. By doing this, we move the fifth biggest number 22 to **X[1]**. At this time, we will have the smallest number 11 in **X[0]**. Thus, we see that we can sort the array of size 6 in 5 passes.

For an array of size n, we required (n-1) passes.

❖ Program for Bubble Sort



Bubble Sort

```cpp
#include <iostream>
using namespace std;

// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        bool swapped = false; // Optimization: Track if a swap occurred
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        // If no swaps occurred, the array is already sorted
        if (!swapped)
            break;
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Main function
int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Unsorted array: ";
    printArray(arr, n);

    bubbleSort(arr, n);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```

❖ Time Complexity:

The bubble sort method of sorting an array of size n requires **(n-1)** passes and **(n-1)** comparisons on each pass. Thus, the total number of comparisons is **(n-1) * (n-1) = n^2 - 2n + 1**, which is **O(n^2)**. Therefore, bubble sort is very inefficient when there are more elements to sorting.

Selection Sort

Selection sort will not require no more than n-1 interchanges. Suppose **x** is an array of size **n** stored in memory. The selection sort algorithm first selects the smallest element in the array **x** and place it at array position 0; then it selects the next smallest element in the array **x** and place it at array position 1. It simply continues this procedure until it places the biggest element in the last position of the array.

The array is passed through (n-1) times and the smallest element is placed in its respective position in the array as detailed below:

Pass 1: Find the location ***j*** of the smallest element in the array **x [0], x[1],** **x[n-1]**, and then interchange **x[j]** with **x[0]**. Then **x[0]** is sorted.

Pass 2: Leave the first element and find the location j of the smallest element in the sub-array **x[1], x[2],** **x[n-1]**, and then interchange **x[1]** with **x[j]**. Then

x[0], x[1] are sorted.

<u>Pass 3:</u> Leave the first two elements and find the location j of the smallest element in the sub-array **x[2], x[3], ... x[n-1]**, and then interchange **x[2]** with **x[j]**. Then **x[0], x[1], x[2]** are sorted.

<u>Pass (n-1):</u> Find the location j of the smaller of the elements **x[n-2]** and **x[n-1]**, and then interchange **x[j]** and **x[n-2]**. Then **x[0], x[1],** **x[n-2]** are sorted. Of course, during this pass **x[n-1]** will be the biggest element and so the entire array is sorted.

❖ TIME COMPLEXITY:

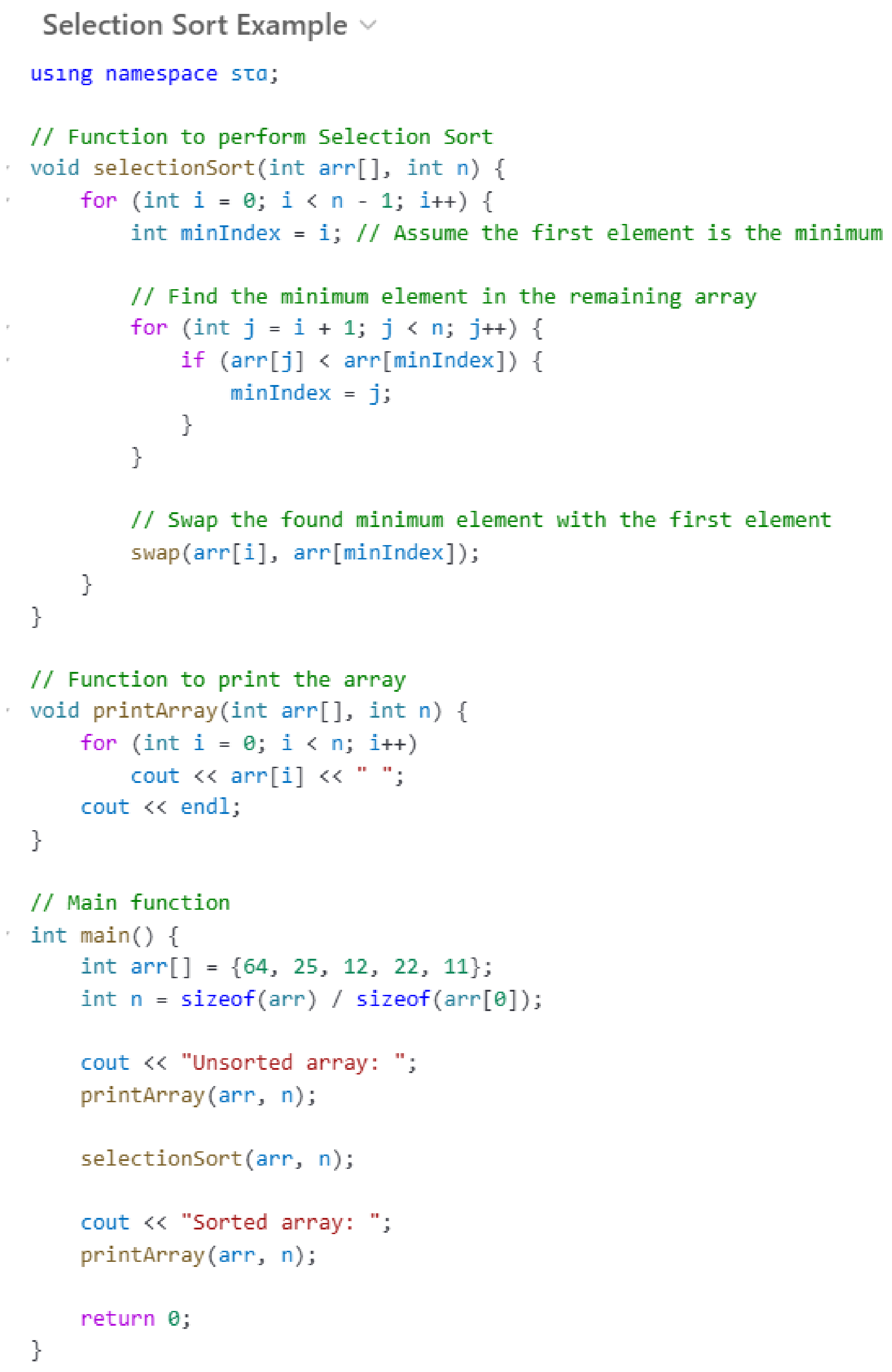
In general, we prefer selection sort in case where the insertion sort or the bubble sort requires exclusive swapping. In spite of superiority of the selection sort over bubble sort and the insertion sort (there is significant decrease in run time), its efficiency is also **O(n^2)** for n data items.

❖ EXAMPLE:

Let us consider the following example with 9 elements to analyze selection Sort:

1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***Remarks***
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	**50**	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	**50**	**55**	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	**50**	**55**	**60**	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	85	80	Find the eighth smallest element
							i	j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	The outer loop ends.

❖ PROGRAM FOR SELECTION SORT



Selection Sort Example

```cpp
using namespace std;

// Function to perform Selection Sort
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i; // Assume the first element is the minimum

        // Find the minimum element in the remaining array
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        // Swap the found minimum element with the first element
        swap(arr[i], arr[minIndex]);
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Main function
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Unsorted array: ";
    printArray(arr, n);

    selectionSort(arr, n);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```

❖ Recursive Program for selection sort

```
#include <iostream>
using namespace std;

// Function to find the index of the minimum element in the array
int findMinIndex(int arr[], int start, int n) {
    int minIndex = start;
    for (int i = start + 1; i < n; i++) {
        if (arr[i] < arr[minIndex])
            minIndex = i;
    }
    return minIndex;
}

// Recursive function for Selection Sort
void recursiveSelectionSort(int arr[], int start, int n) {
    // Base case: If start index reaches the last element, return
    if (start >= n - 1)
        return;

    // Find the minimum element in the remaining array
    int minIndex = findMinIndex(arr, start, n);

    // Swap the found minimum element with the first element of the unsorted part
    swap(arr[start], arr[minIndex]);

    // Recursively call the function for the remaining unsorted array
    recursiveSelectionSort(arr, start + 1, n);
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Main function
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Unsorted array: ";
    printArray(arr, n);

    recursiveSelectionSort(arr, 0, n);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```

Quick Sort

The quick sort was invented in 1960. It was one of the first most efficient sorting algorithms. It is an example of a class of algorithms that work by “divide and conquer” technique. The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the pivot element. Once the array has been rearranged in this way with respect to the pivot, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted. The function **partition()** makes use of two pointers up and down which are moved toward each other in the following fashion:

1. Repeatedly increase the pointer ‘up’ until a[up] >= pivot.
2. Repeatedly decrease the pointer ‘down’ until a[down] <= pivot.
3. If down > up, interchange a[down] with a[up]
4. Repeat the steps 1, 2 and 3 till the ‘up’ pointer crosses the ‘down’ pointer. If ‘up’ pointer crosses ‘down’ pointer, the position for pivot is found and place pivot element in ‘down’ pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

1. It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

2. Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . . x[high].

3. It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).

4. It calls itself recursively to sort the right sub-array x[j+1], x[j+2], . . x[high] between positions j+1 and high.

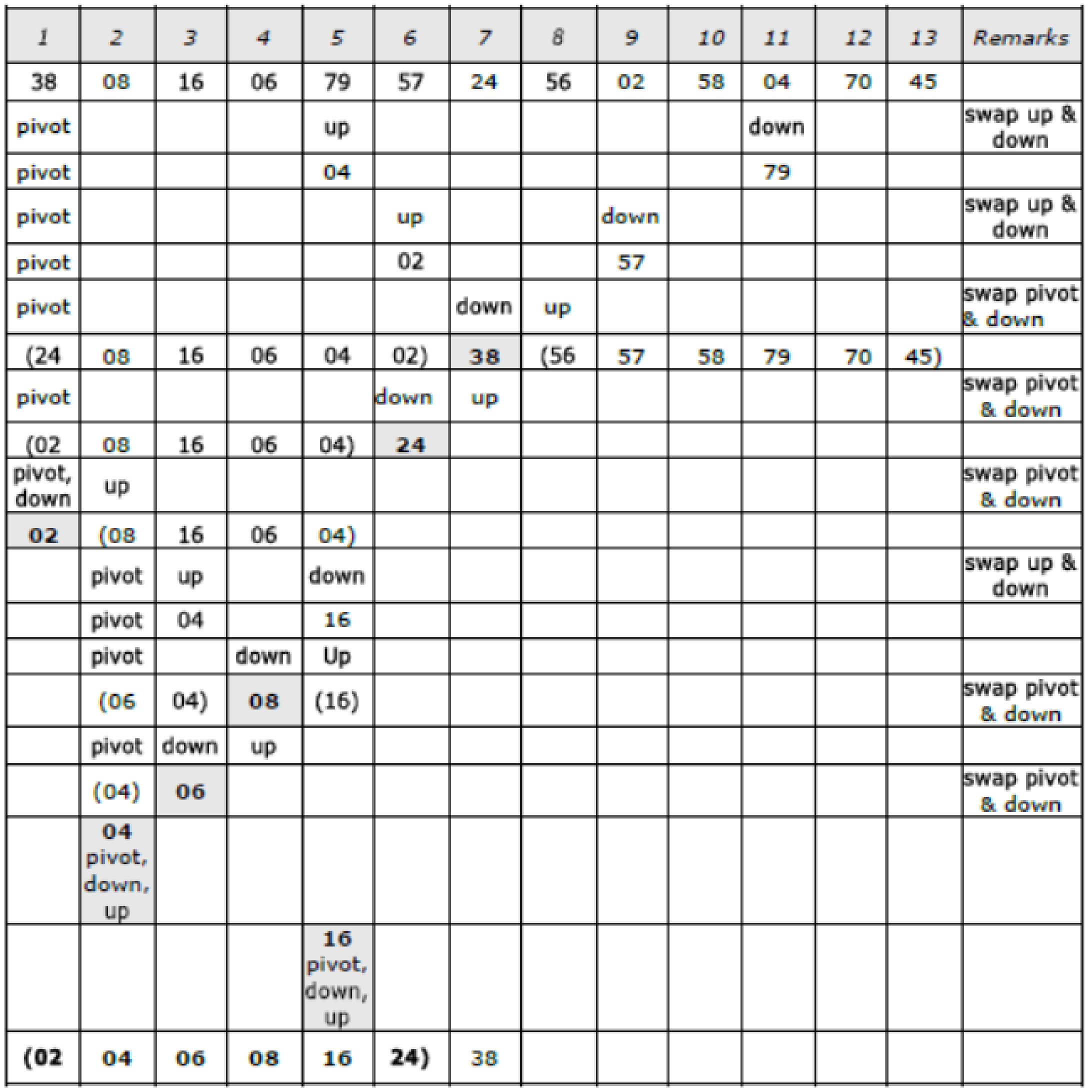
The time complexity of quick sort algorithm is of ***O(n log n)***.

Example:

Select first element as the pivot element. Move 'up' pointer from left to right in search of an element larger than pivot. Move the 'down' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped.

This process continues till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and interchange pivot and element at 'down' position.

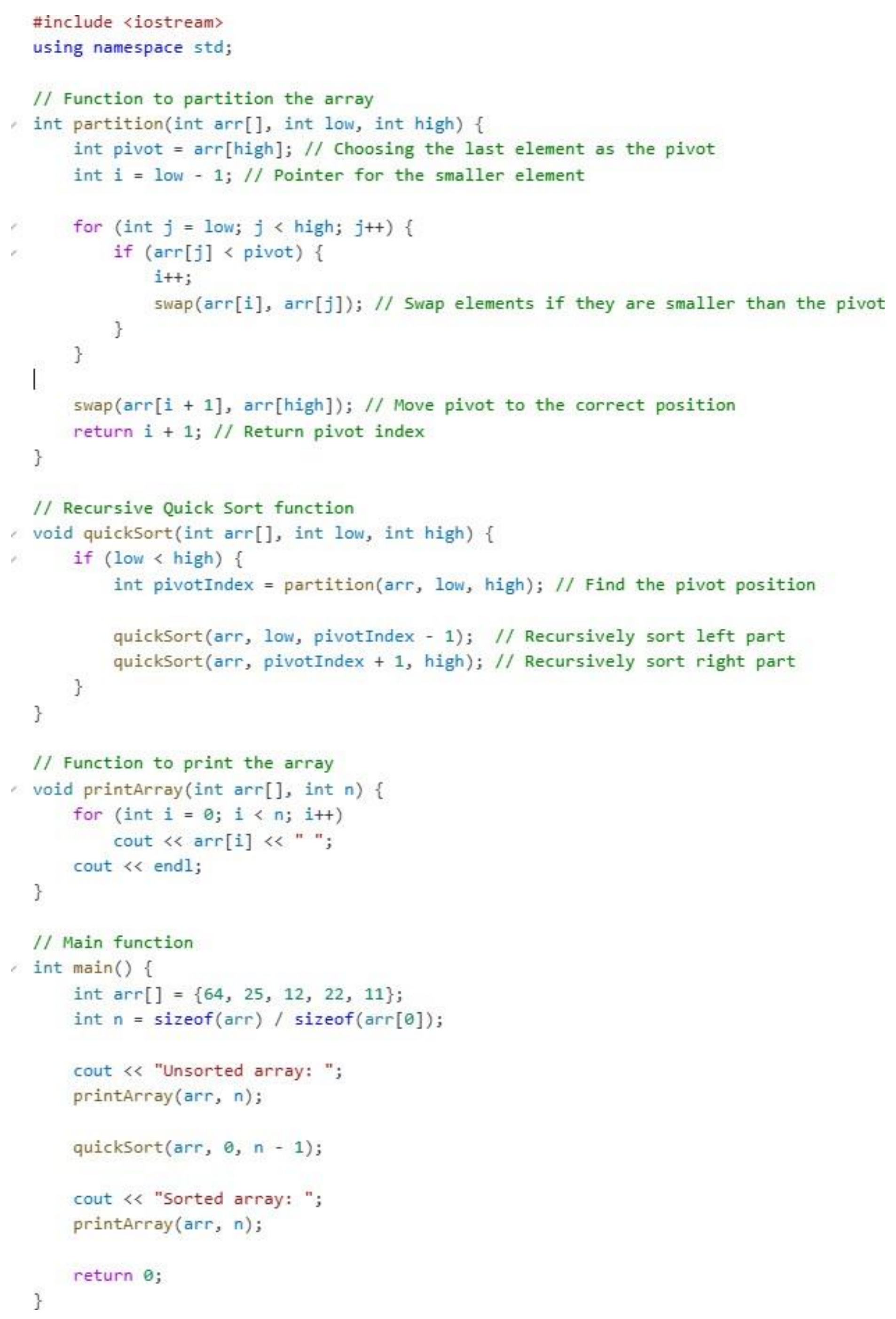
Let us consider the following example with 13 elements to analyze quick sort:



1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						down			swap up & down
pivot				04						79			
pivot					up			down					swap up & down
pivot					02			57					
pivot						down	up						swap pivot & down
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	
pivot					down	up							swap pivot & down
(02	08	16	06	04)	**24**								
pivot, down	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	**08**	(16)									swap pivot & down
	pivot	down	up										
	(04)	**06**											swap pivot & down
	04 pivot, down, up												
				16 pivot, down, up									
(02	**04**	**06**	**08**	**16**	**24)**	38							

							(56	57	58	79	70	45)	
							pivot	up				down	swap up & down
							pivot	45				57	
							pivot	down	up				swap pivot & down
							(45)	**56**	(58	79	70	57)	
							45 pivot, down, up						swap pivot & down
									(58 pivot	79 up	70	57) down	swap up & down
										57		79	
										down	up		
									(57)	**58**	(70	79)	swap pivot & down
									57 pivot, down, up				
											(70	79)	
											pivot, down	up	swap pivot & down
											70		
												79 pivot, down, up	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

❖ Recursive program for Quick Sort:



```cpp
#include <iostream>
using namespace std;

// Function to partition the array
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing the last element as the pivot
    int i = low - 1; // Pointer for the smaller element

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]); // Swap elements if they are smaller than the pivot
        }
    }

    swap(arr[i + 1], arr[high]); // Move pivot to the correct position
    return i + 1; // Return pivot index
}

// Recursive Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high); // Find the pivot position

        quickSort(arr, low, pivotIndex - 1);  // Recursively sort left part
        quickSort(arr, pivotIndex + 1, high); // Recursively sort right part
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Main function
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Unsorted array: ";
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```