جامـــعـة المـــسـتـقـبـل
AL MUSTAQBAL UNIVERSITY

قسـم الامــــن الـــــــسيبرانـــــــي

Department of Cyber Security

Subject:

Searching And Sorting Algorithms

Class:

Second

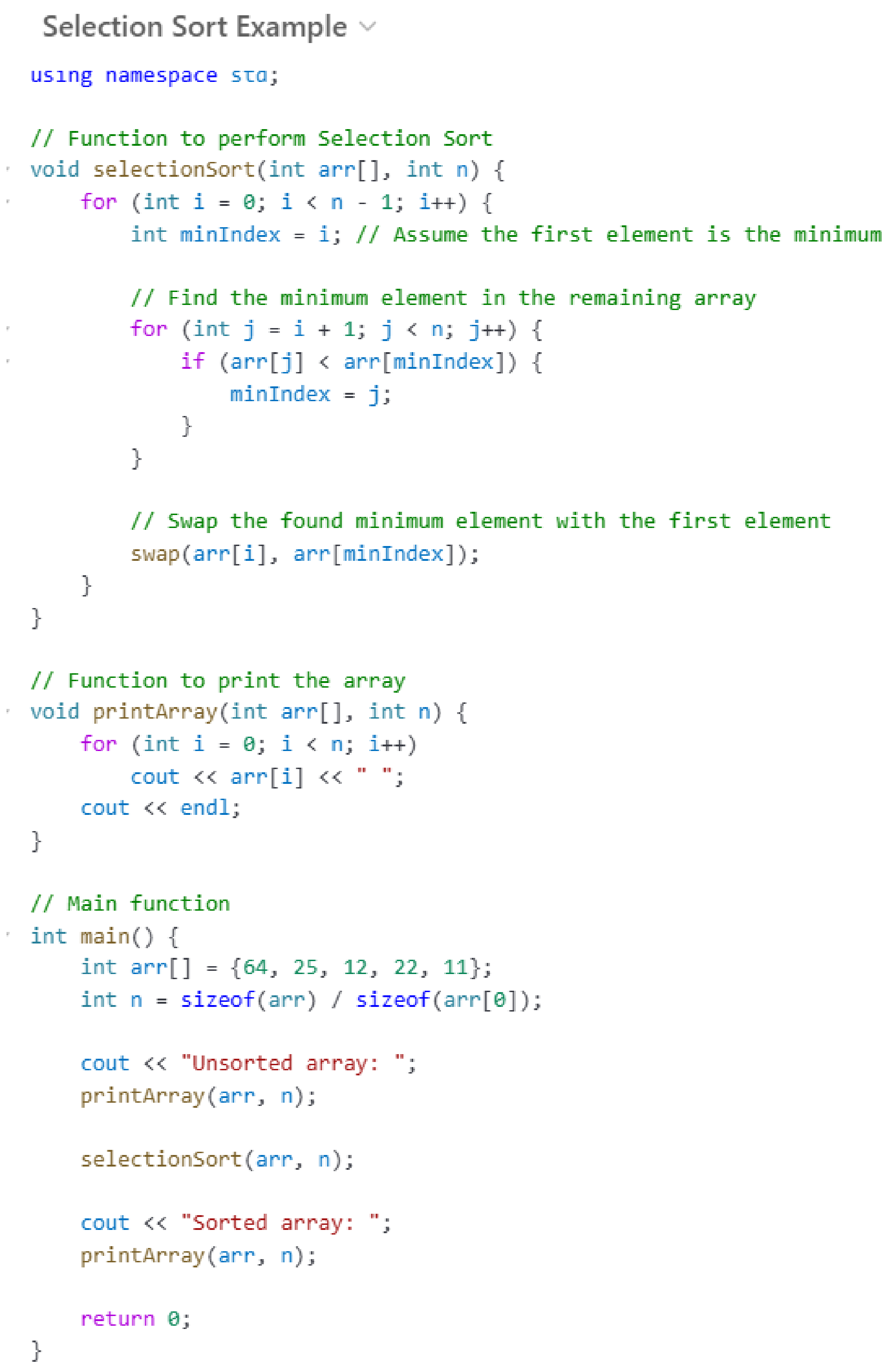
Lecturer:

Asst. Prof. Dr. Ali Kadhum Al-Quraby

Lecture: (2)

Sorting Methods

❖ Program for selection sort



Selection Sort Example

```
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.

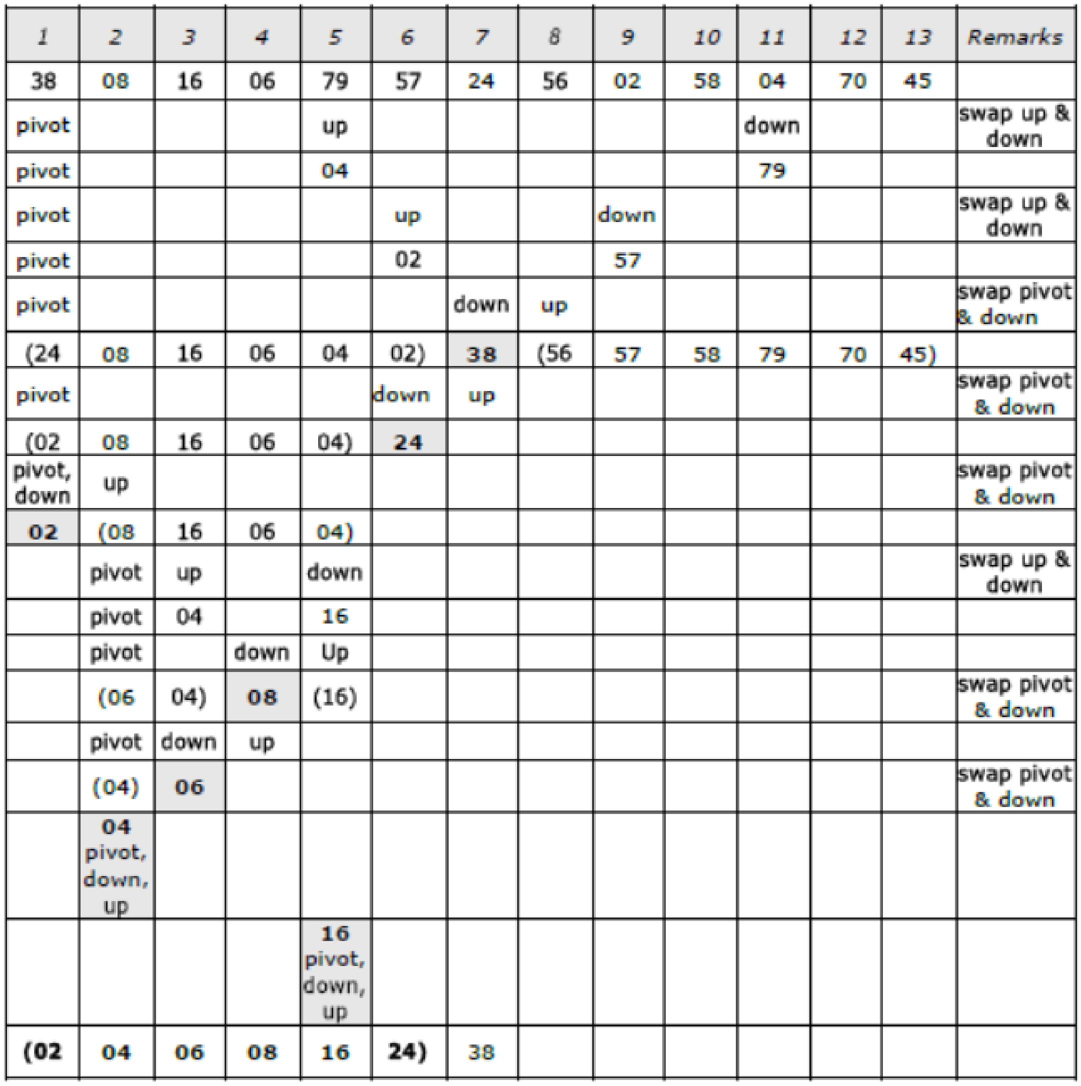
The time complexity of quick sort algorithm is of ***O(n log n)***.

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;
}
```