Topological sorting order

```
#include <stdio.h>
#include <stdlib.h>

// Define the maximum number of vertices
#define MAX_VERTICES 6

// Function to perform a Depth First Search (DFS) from a given vertex
void dfs(int vertex, int adjMatrix[MAX_VERTICES][MAX_VERTICES], int visited[], int *stack, int
*top) {
    visited[vertex] = 1;

    // Explore all adjacent vertices
    for (int i = 0; i < MAX_VERTICES; i++) {
        if (adjMatrix[vertex][i] == 1 && !visited[i]) {
            dfs(i, adjMatrix, visited, stack, top);
        }
    }

    // Push the vertex to the stack
    stack[(*top)++] = vertex;
}

// Function to perform Topological Sort
void topologicalSort(int adjMatrix[MAX_VERTICES][MAX_VERTICES], int startVertex) {
    int visited[MAX_VERTICES] = {0}; // Visited array to keep track of visited vertices
    int stack[MAX_VERTICES];         // Stack to store the topological order
    int top = 0;                     // Stack top pointer

    // Perform DFS starting from the given startVertex
    dfs(startVertex, adjMatrix, visited, stack, &top);

    // Perform DFS for any unvisited vertex
    for (int i = 0; i < MAX_VERTICES; i++) {
        if (!visited[i]) {
            dfs(i, adjMatrix, visited, stack, &top);
        }
    }

    // Print the topological order
    printf("Topological Sort Order: ");
    for (int i = top - 1; i >= 0; i--) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}

int main() {
    // Define the adjacency matrix for the given graph
    int adjMatrix[MAX_VERTICES][MAX_VERTICES] = {
        {0, 0, 0, 0, 0, 0}, // Vertex 0
        {0, 0, 0, 0, 0, 0}, // Vertex 1
        {0, 1, 0, 1, 0, 0}, // Vertex 2
        {0, 1, 0, 0, 0, 0}, // Vertex 3
        {1, 1, 0, 0, 0, 0}, // Vertex 4
        {1, 0, 1, 0, 0, 0}  // Vertex 5
    };

    int startVertex = 5; // Starting vertex for the topological sort

    // Perform the Topological Sort
    topologicalSort(adjMatrix, startVertex);

    return 0;
}
```

Quick sort :

```
#include <stdio.h>

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

// Partition function using the first element as pivot
int partition(int arr[], int low, int high) {
    int pivot = arr[low]; // Taking the first element as pivot
    int left = low + 1;
    int right = high;

    while (left <= right) {
        // Move left index to the right as long as elements are less than or equal to pivot
        while (left <= right && arr[left] <= pivot) {
            left++;
        }
        // Move right index to the left as long as elements are greater than pivot
        while (left <= right && arr[right] > pivot) {
            right--;
        }
        // If left is less than right, swap the elements
        if (left < right) {
            swap(&arr[left], &arr[right]);
        }
    }
    // Finally, swap the pivot element with the right index element
    swap(&arr[low], &arr[right]);
    return right; // Return the pivot index
}

// QuickSort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partition the array and get the pivot index
        int pivotIndex = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {54, 26, 93, 17, 77, 31, 44, 55, 20};
    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;
}
```