1. Write a program to print all the nodes reachable from a given starting node in a digraph using BFS method.

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>

using namespace std;

void bfs(vector<vector<int>>& graph, int start) {
    unordered_set<int> visited;
    queue<int> q;
    q.push(start);

    while (!q.empty()) {
        int node = q.front();
        q.pop();

        if (visited.find(node) == visited.end()) {
            cout << node << " ";
            visited.insert(node);
            for (int neighbor : graph[node]) {
                if (visited.find(neighbor) == visited.end()) {
                    q.push(neighbor);
                }
            }
        }
    }
}

int main() {
    vector<vector<int>> graph = {
        {1, 2},
        {2},
```

```
        {3},
        {1, 2}
    };
    bfs(graph, 0);
    return 0;
}
```

2a. Write a program to obtain the Topological ordering of vertices in a given digraph.(Vertex removal method)

```
import java.util.*;

public class TopologicalSort {
    public static List<Integer> topologicalSort(Map<Integer, List<Integer>> graph) {
        Map<Integer, Integer> inDegree = new HashMap<>();
        for (int u : graph.keySet()) {
            inDegree.putIfAbsent(u, 0);
            for (int v : graph.get(u)) {
                inDegree.put(v, inDegree.getOrDefault(v, 0) + 1);
            }
        }

        Queue<Integer> queue = new LinkedList<>();
        for (int u : inDegree.keySet()) {
            if (inDegree.get(u) == 0) {
                queue.add(u);
            }
        }

        List<Integer> topOrder = new ArrayList<>();
        while (!queue.isEmpty()) {
            int u = queue.poll();
            topOrder.add(u);

            for (int v : graph.get(u)) {
                inDegree.put(v, inDegree.get(v) - 1);
```

```
                if (inDegree.get(v) == 0) {
                    queue.add(v);
                }
            }
        }

        if (topOrder.size() != graph.size()) {
            throw new IllegalStateException("Graph has a cycle!");
        }
        return topOrder;
    }

    public static void main(String[] args) {
        Map<Integer, List<Integer>> graph = new HashMap<>();
        graph.put(0, Arrays.asList(1, 2));
        graph.put(1, Arrays.asList(2));
        graph.put(2, Arrays.asList(3));
        graph.put(3, Arrays.asList(4));
        graph.put(4, new ArrayList<>());

        System.out.println(topologicalSort(graph));
    }
}
```

2b. Write a program to obtain the Topological ordering of vertices in a given digraph.(DFS method)

```
import java.util.*;

public class TopologicalSortDFS {
    private int vertices;
    private LinkedList<Integer> adj[];

    // Constructor
    TopologicalSortDFS(int v) {
        vertices = v;
```

```
    adj = new LinkedList[v];
    for (int i = 0; i < v; ++i) {
        adj[i] = new LinkedList();
    }
}

// Function to add an edge into the graph
void addEdge(int v, int w) {
    adj[v].add(w);
}

// A recursive function used by topologicalSort
void topologicalSortUtil(int v, boolean visited[], Stack<Integer> stack) {
    // Mark the current node as visited
    visited[v] = true;
    Integer i;

    // Recur for all the vertices adjacent to this vertex
    for (Integer neighbor : adj[v]) {
        if (!visited[neighbor]) {
            topologicalSortUtil(neighbor, visited, stack);
        }
    }

    // Push current vertex to stack which stores result
    stack.push(v);
}

// The function to do Topological Sort. It uses recursive topologicalSortUtil()
void topologicalSort() {
    Stack<Integer> stack = new Stack<>();

    // Mark all the vertices as not visited
    boolean visited[] = new boolean[vertices];
    for (int i = 0; i < vertices; i++) {
        visited[i] = false;
    }
```

```
        // Call the recursive helper function to store Topological Sort starting from all vertices one by one
        for (int i = 0; i < vertices; i++) {
            if (!visited[i]) {
                topologicalSortUtil(i, visited, stack);
            }
        }

        // Print contents of stack
        while (!stack.empty()) {
            System.out.print(stack.pop() + " ");
        }
    }

    // Driver method
    public static void main(String args[]) {
        TopologicalSortDFS g = new TopologicalSortDFS(6);
        g.addEdge(5, 2);
        g.addEdge(5, 0);
        g.addEdge(4, 0);
        g.addEdge(4, 1);
        g.addEdge(2, 3);
        g.addEdge(3, 1);

        System.out.println("Topological sort of the given graph:");
        g.topologicalSort();
    }
}
```

OR

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;
```

```
void topo(vector<vector<int>>&graph, vector<bool>&visited, stack<int>&s,int start){
    visited[start] = true;

    for(int node : graph[start]){
        if(!visited[node]){
            topo(graph, visited, s, node);
        }
    }
    s.push(start);
}

int main(){
    stack<int> st;
    vector<vector<int>> graph = {{},{},{3},{1},{0,1},{0,2}};
    vector<bool> visited(6,false);
    for(int i = 0; i < 6; i++){
        if(!visited[i])
            topo(graph,visited,st,i);
    }
    while(!st.empty()){
        cout << st.top() << " ";
        st.pop();
    }

}
```

3. Implement Johnson Trotter algorithm to generate permutations

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;
```

```
void printPermutation(const vector<int>& perm) {
    for (int i : perm) {
        cout << i << " ";
    }
    cout << endl;
}

void johnsonTrotter(int n) {
    vector<int> perm(n), dir(n, -1);
    iota(perm.begin(), perm.end(), 1); // Fill perm with 1, 2, 3, ..., n

    while (true) {
        printPermutation(perm);

        int mobile = -1, mobileIndex = -1;
        for (int i = 0; i < n; ++i) {
            if ((dir[i] == -1 && i > 0 && perm[i] > perm[i - 1]) || // Left mobile
                (dir[i] == 1 && i < n - 1 && perm[i] > perm[i + 1]))// Right mobile
                {
                if (perm[i] > mobile) {
                    mobile = perm[i];
                    mobileIndex = i;
                }
            }
        }

        if (mobileIndex == -1) break;

        int swapIndex = mobileIndex + dir[mobileIndex]; // Element to swap with mobile element
        swap(perm[mobileIndex], perm[swapIndex]); // Swap mobile element with the element it is looking at
        swap(dir[mobileIndex], dir[swapIndex]); // Swap the directions

        for (int i = 0; i < n; ++i) { // Update directions of elements greater than the mobile element
            if (perm[i] > mobile) {
                dir[i] = -dir[i];
            }
        }
    }
```

```
}

int main() {
    int n = 3;
    johnsonTrotter(n);
    return 0;
}
```

4. Sort a given set of N integer elements using Merge Sort technique and compute its time taken. Run the program for different values of N and analyze its time complexity.

```
import java.util.Arrays;

public class MergeSort {
    public static void mergeSort(int[] arr) {
        if (arr.length > 1) {
            int mid = arr.length / 2;
            int[] left = Arrays.copyOfRange(arr, 0, mid);
            int[] right = Arrays.copyOfRange(arr, mid, arr.length);

            mergeSort(left);
            mergeSort(right);

            merge(arr, left, right);
        }
    }

    private static void merge(int[] arr, int[] left, int[] right) {
        int i = 0, j = 0, k = 0;

        while (i < left.length && j < right.length) {
            if (left[i] <= right[j]) {
                arr[k++] = left[i++];
            } else {
```

```
            arr[k++] = right[j++];
        }
    }

    while (i < left.length) {
        arr[k++] = left[i++];
    }

    while (j < right.length) {
        arr[k++] = right[j++];
    }
}

public static void main(String[] args) {
    int[] arr = {64, 34, 25, 12, 22, 11, 90};
    long startTime = System.nanoTime();
    mergeSort(arr);
    long endTime = System.nanoTime();

    System.out.println("Sorted array: " + Arrays.toString(arr));
    System.out.println("Time taken: " + (endTime - startTime) + " ns");
}
}
```

5a. Sort a given set of N integer elements using Quick Sort technique and compute its time taken. Run the program for different values of N and analyze its time complexity. (Last element as pivot OR single pointer method)

```
import java.util.Arrays;

public class QuickSort {
    public static void quickSort(int[] arr) {
        quickSort(arr, 0, arr.length - 1);
    }
```

```
private static 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);
    }
}

private static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
}

public static void main(String[] args) {
    int[] arr = {64, 34, 25, 12, 22, 11, 90};
    long startTime = System.nanoTime();
    quickSort(arr);
    long endTime = System.nanoTime();

    System.out.println("Sorted array: " + Arrays.toString(arr));
    System.out.println("Time taken: " + (endTime - startTime) + " ns");
```

```
    }
}
```

5b. Sort a given set of N integer elements using Quick Sort technique and compute its time taken. Run the program for different values of N and analyze its time complexity.(First element as pivot OR two pointer method)

```
import java.util.Arrays;

public class QuickSortFirstPivot {
    public static void quickSort(int[] arr) {
        quickSort(arr, 0, arr.length - 1);
    }

    private static 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);
        }
    }

    private static int partition(int[] arr, int low, int high) {
        int pivot = arr[low];
        int left = low + 1;
        int right = high;

        while (left <= right) {
            while (left <= right && arr[left] <= pivot) {
                left++;
            }

            while (left <= right && arr[right] >= pivot) {
```

```
            right--;
        }

        if (left < right) {
            int temp = arr[left];
            arr[left] = arr[right];
            arr[right] = temp;
        }
    }

    int temp = arr[low];
    arr[low] = arr[right];
    arr[right] = temp;

    return right;
}

public static void main(String[] args) {
    int[] arr = {64, 34, 25, 12, 22, 11, 90};
    long startTime = System.nanoTime();
    quickSort(arr);
    long endTime = System.nanoTime();

    System.out.println("Sorted array: " + Arrays.toString(arr));
    System.out.println("Time taken: " + (endTime - startTime) + " ns");
}
}
```

6. Sort a given set of N integer elements using Heap Sort technique and analyze its time complexity.

```
#include <iostream>
#include <vector>
#include <ctime>

using namespace std;
```

```
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(vector<int>& arr) {
    int n = arr.size();

    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

int main() {
    vector<int> arr = {64, 34, 25, 12, 22, 11, 90};
    clock_t startTime = clock();
    heapSort(arr);
    clock_t endTime = clock();
```

```
    cout << "Sorted array: ";
    for (int i : arr) {
        cout << i << " ";
    }
    cout << endl;

    cout << "Time taken: " << (double)(endTime - startTime) / CLOCKS_PER_SEC << " s" << endl;
    return 0;
}
```

7. Implement 0/1 Knapsack problem using dynamic programming.

```
public class Knapsack {
    public static int knapsack(int[] weights, int[] values, int capacity) {
        int n = values.length;
        int[][] dp = new int[n + 1][capacity + 1];

        for (int i = 0; i <= n; i++) {
            for (int w = 0; w <= capacity; w++) {
                if (i == 0 || w == 0) {
                    dp[i][w] = 0;
                } else if (weights[i - 1] <= w) {
                    dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
                } else {
                    dp[i][w] = dp[i - 1][w];
                }
            }
        }

        return dp[n][capacity];
    }

    public static void main(String[] args) {
        int[] values = {60, 100, 120};
        int[] weights = {10, 20, 30};
```

```
        int capacity = 50;
        System.out.println(knapsack(weights, values, capacity));
    }
}
```

8. Implement All Pair Shortest paths problem using Floyd's algorithm.

```
#include <iostream>
#include <vector>
#define INF 99999

using namespace std;

void floydWarshall(vector<vector<int>>& graph) {
    int V = graph.size();

    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (graph[i][k] + graph[k][j] < graph[i][j]) {
                    graph[i][j] = graph[i][k] + graph[k][j];
                }
            }
        }
    }

    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (graph[i][j] == INF) {
                cout << "INF ";
            } else {
                cout << graph[i][j] << " ";
            }
        }
        cout << endl;
    }
```

```
}

int main() {
    vector<vector<int>> graph = {
        {0, 5, INF, 10},
        {INF, 0, 3, INF},
        {INF, INF, 0, 1},
        {INF, INF, INF, 0}
    };
    floydWarshall(graph);
    return 0;
}
```

9. Find Minimum Cost Spanning Tree of a given undirected graph using Prim algorithm.

```
#include <iostream>
#include <vector>
#include <climits> // For INT_MAX

using namespace std;

// Function to find the vertex with the minimum key value that is not yet included in the MST
int minKey(const vector<int>& key, const vector<bool>& mstSet, int V) {
    int min = INT_MAX;
    int minIndex = -1; // Initialize with -1 to handle edge cases

    for (int i = 0; i < V; i++) {
        if (!mstSet[i] && key[i] < min) {
            min = key[i];
            minIndex = i;
        }
    }
    return minIndex;
}
```

```
// Function to print the constructed MST
 void printMST(const vector<int>& parent, const vector<vector<int>>& graph) {
    int total_weight = 0;
    cout << "Edge \tWeight\n";
    for (int i = 1; i < graph.size(); i++) {
        cout << parent[i] << " - " << i << " \t" << graph[i][parent[i]] << " \n";
        total_weight += graph[i][parent[i]];
    }
    cout << "Total Weight = " << total_weight;
 }

// Function to implement Prim's MST algorithm
void primMST(const vector<vector<int>>& graph) {
    int V = graph.size();
    vector<int> key(V, INT_MAX);     // Initialize all keys as INFINITE
    vector<int> parent(V, -1);      // Array to store the constructed MST
    vector<bool> mstSet(V, false);  // To check if a vertex is included in the MST

    key[0] = 0; // Start from the first vertex

    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet, V); // Pick the minimum key vertex from the set of vertices not yet processed
        mstSet[u] = true; // Add the picked vertex to the MST set

        for (int i = 0; i < V; i++) {
            // Update the key value and parent index of the adjacent vertices of the picked vertex
            if (graph[u][i] && !mstSet[i] && graph[u][i] < key[i]) {
                parent[i] = u;
                key[i] = graph[u][i];
            }
        }
    }

    printMST(parent, graph); // Print the MST
}

int main() {
```

```
    vector<vector<int>> graph = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };

    primMST(graph); // Call Prim's MST algorithm
    return 0;
}
```

10. Write program to check whether a given graph is connected or not using DFS method

```
#include <iostream>
#include <vector>
#include <unordered_set>

using namespace std;

void dfs(const vector<vector<int>>& graph, int node, unordered_set<int>& visited) {
    visited.insert(node);
    for (int neighbor : graph[node]) {
        if (visited.find(neighbor) == visited.end()) {
            dfs(graph, neighbor, visited);
        }
    }
}

bool isConnected(const vector<vector<int>>& graph) {
    unordered_set<int> visited;
    dfs(graph, 0, visited);
    return visited.size() == graph.size();
}
```

```
int main() {
    vector<vector<int>> graph = {
        {1, 2},
        {0, 2},
        {0, 1, 3},
        {2}
    };
    cout << (isConnected(graph) ? "Connected" : "Not Connected") << endl;
    return 0;
}
```