**Algorithm**

**Design**

**Created By:**

**Susmoy Barman**

**Roll: 1610576113**

**Dept. of CSE**

**University of Rajshahi**

**List of algorithms:**

* **Sorting:**

1. **Insertion sort**
2. **Selection sort**
3. **Bubble sort**
4. **Merge sort**
5. **Quick sort**

* **Searching:**

1. **Linear search**
2. **Binary search**

* **Graph:**

1. **DFS**
2. **BFS**
3. **Iterative DFS (IDFS)**
4. **Dijkstra’s shortest path**
5. **A\* algorithm**

* **Insertion Sort:**

**INSERTION(A[])**

**{**

**For j=2 to A.length do**

**Key = A[j]**

**I = j – 1**

**While i>0 and A[j]>key do**

**A[i+1] = A[i]**

**i=i-1**

**end while**

**A[i+1] = key**

**end for**

**}**

* **Complexity:**

**Time: O(n2)**

**Space: O(1)**

* **Selection sort:**

**void selectionSort(int arr[], int n)**

**{**

**int i, j, min\_idx;**

**// One by one move boundary of unsorted subarray**

**for (i = 0; i < n-1; i++)**

**{**

**// Find the minimum element in unsorted array**

**min\_idx = i;**

**for (j = i+1; j < n; j++)**

**if (arr[j] < arr[min\_idx])**

**min\_idx = j;**

**// Swap the found minimum element with the first element**

**swap(&arr[min\_idx], &arr[i]);**

**}**

**}**

**Complexity:**

**Time: Worst Case: O(n2)**

**Best Case: O(n2)**

**Average Case: O(n2)**

**Space: O(1)**

* **Bubble sort:**

**void bubbleSort(int arr[], int n)**

**{**

**int i, j;**

**for (i = 0; i < n-1; i++)**

**// Last i elements are already in place**

**for (j = 0; j < n-i-1; j++)**

**if (arr[j] > arr[j+1])**

**swap(&arr[j], &arr[j+1]);**

**}**

**Complexity:**

**Time:**

**Worst Case: O(n2)**

**Average Case: O(n2)**

**Best Case: O(n)**

**Space: O(1)**

* **Merge sort:**

**void merge(int arr[], int l, int m, int r)**

**{**

**int i, j, k;**

**int n1 = m - l + 1;**

**int n2 =  r - m;**

**int L[n1];**

**int R[n2];**

**for (i = 0; i < n1; i++)**

**L[i] = arr[l + i];**

**for (j = 0; j < n2; j++)**

**R[j] = arr[m + 1+ j];**

**i = 0;**

**j = 0;**

**k = l;**

**while (i < n1 && j < n2)**

**{**

**if (L[i] <= R[j])**

**{**

**arr[k] = L[i];**

**i++;**

**}**

**else**

**{**

**arr[k] = R[j];**

**j++;**

**}**

**k++;**

**}**

**while (i < n1)**

**{**

**arr[k] = L[i];**

**i++;**

**k++;**

**}**

**while (j < n2)**

**{**

**arr[k] = R[j];**

**j++;**

**k++;**

**}**

**}**

**void mergeSort(int arr[], int l, int r)**

**{**

**if (l < r)**

**{**

**int m = l+(r-l)/2;**

**mergeSort(arr, l, m);**

**mergeSort(arr, m+1, r);**

**merge(arr, l, m, r);**

**}**

**}**

**Complexity:**

**Time:**

**Worst case: O(n\*logn)**

**Average case: O(n\*logn)**

**Best case: O(n\*logn)**

**Space: O(n)**

* **Quick sort**

**int partition (int arr[], int low, int high)**

**{**

**int pivot = arr[high];**

**int i = (low - 1);**

**for (int j = low; j <= high- 1; j++)**

**{**

**if (arr[j] <= pivot)**

**{**

**i++;**

**swap(&arr[i], &arr[j]);**

**}**

**}**

**swap(&arr[i + 1], &arr[high]);**

**return (i + 1);**

**}**

**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);**

**}**

**}**

**Complexity:**

* **Time:**

**Worst case: O(n2), Average case: O(n\*logn), Best case: O(n\*logn)**

* **Linear Search**

**int search(int arr[], int n, int x)**

**{**

**int i;**

**for (i = 0; i < n; i++)**

**if (arr[i] == x)**

**return i;**

**return -1;**

**}**

**Complexity:**

* **Time:**

**Worst case: O(n)**

**Average case: O(n)**

* **Space:**

**Normal case: O(1)**

**Worst-case space complexity: O(1) iterative**

* **Binary Search:**
* **The array must be in sorted order.**

**int binarySearch(int arr[], int l, int r, int x)**

**{**

**if (r >= l)**

**{**

**int mid = l + (r - l)/2;**

**if (arr[mid] == x)**

**return mid;**

**if (arr[mid] > x)**

**return binarySearch(arr, l, mid-1, x);**

**return binarySearch(arr, mid+1, r, x);**

**}**

**return -1;**

**}**

**Complexity:**

* **Time:**

**Worst case: O(logn)**

**Average case: O(logn)**

**Best case: O(1)**

* **Space: O(1)**

**Graph Algorithms**

1. **Depth**-**first search** (DFS):

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**vector <int> adj[10];**

**bool visited[10];**

**void dfs(int s) {**

**visited[s] = true;**

**for(int i = 0;i < adj[s].size();++i) {**

**if(visited[adj[s][i]] == false)**

**dfs(adj[s][i]);**

**}**

**}**

**void initialize() {**

**for(int i = 0;i < 10;++i)**

**visited[i] = false;**

**}**

**int main() {**

**int nodes,edges,x,y,connectedComponents = 0;**

**cin >> nodes; //Number of nodes**

**cin >> edges; //Number of edges**

**for(int i = 0;i < edges;++i) {**

**cin >> x >> y;**

**//Undirected Graph**

**adj[x].push\_back(y); //Edge from vertex x to vertex y**

**adj[y].push\_back(x); //Edge from vertex y to vertex x**

**}**

**initialize(); //Initialize all nodes as not visited**

**for(int i = 1;i <= nodes;++i) {**

**if(visited[i] == false) {**

**dfs(i);**

**connectedComponents++;**

**}**

**}**

**cout <<"Number of connected components: "<<connectedComponents << endl;**

**return 0;**

**}**

**Complexity:**

* **Time:**

**Worst case: O( |V| + |E| )**

* **Space:**

**Worst case: O( |V|)**

**Input File**  
6  
4  
1 2  
2 3  
1 3  
4 5

1. **Breadth First Search(BFS)**

**vector <int> v[10] ; //Vector for maintaining adjacency list explained above**

**int level[10]; //To determine the level of each node**

**bool vis[10]; //Mark the node if visited**

**void bfs(int s) {**

**queue <int> q;**

**q.push(s);**

**level[ s ] = 0 ; //Setting the level of the source node as 0**

**vis[ s ] = true;**

**while(!q.empty())**

**{**

**int p = q.front();**

**q.pop();**

**for(int i = 0;i < v[ p ].size() ; i++)**

**{**

**if(vis[ v[ p ][ i ] ] == false)**

**{**

**//Setting the level of each node with an increment in the level of parent node**

**level[ v[ p ][ i ] ] = level[ p ]+1;**

**q.push(v[ p ][ i ]);**

**vis[ v[ p ][ i ] ] = true;**

**}**

**}**

**}**

**}**

**Complexity:**

* **Time: Worst Case: O(|E|) = O(bd)**
* **Space: O(|V|) = O(bd)**

1. **Iterative DFS (IDFS)**

**#include<bits/stdc++.h>**

**using namespace std;**

**// Graph class represents a directed graph using adjacency**

**// list representation.**

**class Graph**

**{**

**int V;    // No. of vertices**

**// Pointer to an array containing**

**// adjacency lists**

**list<int> \*adj;**

**// A function used by IDDFS**

**bool DLS(int v, int target, int limit);**

**public:**

**Graph(int V);   // Constructor**

**void addEdge(int v, int w);**

**// IDDFS traversal of the vertices reachable from v**

**bool IDDFS(int v, int target, int max\_depth);**

**};**

**Graph::Graph(int V)**

**{**

**this->V = V;**

**adj = new list<int>[V];**

**}**

**void Graph::addEdge(int v, int w)**

**{**

**adj[v].push\_back(w); // Add w to v’s list.**

**}**

**// A function to perform a Depth-Limited search**

**// from given source 'src'**

**// Start**

**bool Graph::DLS(int src, int target, int limit)**

**{**

**if (src == target)**

**return true;**

**// If reached the maximum depth, stop recursing.**

**if (limit <= 0)**

**return false;**

**// Recur for all the vertices adjacent to source vertex**

**for (auto i = adj[src].begin(); i != adj[src].end(); ++i)**

**if (DLS(\*i, target, limit-1) == true)**

**return true;**

**return false;**

**}**

**// IDDFS to search if target is reachable from v.**

**// It uses recursive DFSUtil().**

**bool Graph::IDDFS(int src, int target, int max\_depth)**

**{**

**// Repeatedly depth-limit search till the**

**// maximum depth.**

**for (int i = 0; i <= max\_depth; i++)**

**if (DLS(src, target, i) == true)**

**return true;**

**return false;**

**}**

**// End**

**// Driver code**

**int main()**

**{**

**// Let us create a Directed graph with 7 nodes**

**Graph g(7);**

**g.addEdge(0, 1);**

**g.addEdge(0, 2);**

**g.addEdge(1, 3);**

**g.addEdge(1, 4);**

**g.addEdge(2, 5);**

**g.addEdge(2, 6);**

**int target = 6, maxDepth = 3, src = 0;**

**if (g.IDDFS(src, target, maxDepth) == true)**

**cout << "Target is reachable from source "<<"within max depth";**

**else**

**cout << "Target is NOT reachable from source "<<"within max depth";**

**return 0;**

**}**

**Complexity:**

* **Time: Worst Case: O(bd)**
* **Space: O(d)**

1. **Dijkstra’s shortest path**

**#include <stdio.h>**

**#include <limits.h>**

**// Number of vertices in the graph**

**#define V 9**

**// A utility function to find the vertex with minimum distance value, from**

**// the set of vertices not yet included in shortest path tree**

**int minDistance(int dist[], bool sptSet[])**

**{**

**// Initialize min value**

**int min = INT\_MAX, min\_index;**

**for (int v = 0; v < V; v++)**

**if (sptSet[v] == false && dist[v] <= min)**

**min = dist[v], min\_index = v;**

**return min\_index;**

**}**

**// A utility function to print the constructed distance array**

**int printSolution(int dist[], int n)**

**{**

**printf("Vertex   Distance from Source\n");**

**for (int I = 0; I < V; i++)**

**printf(“%d tt %d\n”, I, dist[i]);**

**}**

**// Function that implements Dijkstra’s single source shortest path algorithm**

**// for a graph represented using adjacency matrix representation**

**void dijkstra(int graph[V][V], int src)**

**{**

**int dist[V];     // The output array.  Dist[i] will hold the shortest**

**// distance from src to I**

**bool sptSet[V]; // sptSet[i] will true if vertex I is included in shortest**

**// path tree or shortest distance from src to I is finalized**

**// Initialize all distances as INFINITE and stpSet[] as false**

**for (int I = 0; I < V; i++)**

**dist[i] = INT\_MAX, sptSet[i] = false;**

**// Distance of source vertex from itself is always 0**

**dist[src] = 0;**

**// Find shortest path for all vertices**

**for (int count = 0; count < V-1; count++)**

**{**

**// Pick the minimum distance vertex from the set of vertices not**

**// yet processed. U is always equal to src in the first iteration.**

**Int u = minDistance(dist, sptSet);**

**// Mark the picked vertex as processed**

**sptSet[u] = true;**

**// Update dist value of the adjacent vertices of the picked vertex.**

**For (int v = 0; v < V; v++)**

**// Update dist[v] only if is not in sptSet, there is an edge from**

**// u to v, and total weight of path from src to  v through u is**

**// smaller than current value of dist[v]**

**if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX**

**&& dist[u]+graph[u][v] < dist[v]) {**

**dist[v] = dist[u] + graph[u][v]; }**

**}**

**// print the constructed distance array**

**printSolution(dist, V);**

**}**

**// driver program to test above function**

**int main()**

**{**

**/\* Let us create the example graph discussed above \*/**

**int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},**

**{4, 0, 8, 0, 0, 0, 0, 11, 0},**

**{0, 8, 0, 7, 0, 4, 0, 0, 2},**

**{0, 0, 7, 0, 9, 14, 0, 0, 0},**

**{0, 0, 0, 9, 0, 10, 0, 0, 0},**

**{0, 0, 4, 14, 10, 0, 2, 0, 0},**

**{0, 0, 0, 0, 0, 2, 0, 1, 6},**

**{8, 11, 0, 0, 0, 0, 1, 0, 7},**

**{0, 0, 2, 0, 0, 0, 6, 7, 0}**

**};**

**dijkstra(graph, 0);**

**return 0;**

**}**

* **Time complexity:** It depends on the implementation of **Dijkstra's** Algorithm. Simple algorithm’s Time **complexity** of O(V^2). There are also some time-efficient Algorithms: Graph represented using adjacency list can be reduced to O(E log V) with the help of binary heap.