Important Algorithms

1. Graph Traversal
    - Breadth First Search (BFS)
    - 0-1 BFS
    - Depth First Search (DFS)
2. Shortest Path (From Source To Every Other Node In Graph)
    - Bellman Ford
    - Dijkstra
    - Shortest Path in DAG (Using Topological Sort)
3. Topological Sort
    - Using DFS
    - Kahn's Algorithm
4. Cycle Detection
    - Using DFS (Undirected graph & Directed graph)
    - Using Union-Find
5. Minimum Spanning Tree
    - Prim's Algorithm
    - Kruskal's Algorithm
6. Maximum Flow
    - Ford Fulkerson
7. Connectivity
    - Bipartite Graph
    - Cut Vertices
    - Bridges

1-A: Bellman Ford

Adjacency List Representation of Graph - Time: O(V*(V+E)) Space: O(V+E)

```
vector<int> bellman_ford(int n, int src, vector<vector<pair<int, int>>> adj) {
    vector<int> dist(n, INT_MAX);
    dist[src] = 0;

    for (int i = 0; i < n - 1; i++) {
        for (int u = 0; u < n; u++) {
            for (auto p : adj[u]) {
                int v = p.first, weight = p.second;
                if (dist[u] != INT_MAX && dist[v] > dist[u] + weight) {
                    dist[v] = dist[u] + weight;
                }
            }
        }
    }

    for (int u = 0; u < n; u++) {
        for (auto p : adj[u]) {
            int v = p.first, weight = p.second;
            if (dist[u] != INT_MAX && dist[v] > dist[u] + weight) {
                // Negative cycle detected
                dist.clear();
                return dist;
            }
        }
    }

    return dist;
}
```

```
int main() {
    int n, m, src;
    cin >> n >> m >> src;

    vector<vector<pair<int, int>>> adj(n);
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u].push_back({v, w});
    }

    vector<int> dist = bellman_ford(n, src, adj);

    if (dist.empty()) cout << "Negative cycle detected\n";
    else {
        for (int i = 0; i < n; i++) {
            cout << src << " -> " << i << " : "<< dist[i] << "\n";
        }
    }

    return 0;
}
```

1-A: Bellman Ford

Vector containing all the edges of Graph - Time: O(VE) Space: O(V+E)

```
struct Edge {
    int from, to, weight;
};
```

```
vector<int> bellman_ford(int n, int src, vector<Edge> edges) {
    vector<int> dist(n, INT_MAX);
    dist[src] = 0;

    for (int i = 0; i < n - 1; i++) {
        for (Edge e : edges) {
            if (dist[e.from] != INT_MAX && dist[e.to] > dist[e.from] + e.weight)
{
                dist[e.to] = dist[e.from] + e.weight;
            }
        }
    }

    for (Edge e : edges) {
        if (dist[e.from] != INT_MAX && dist[e.to] > dist[e.from] + e.weight) {
            // Negative cycle detected
            dist.clear();
            break;
        }
    }
    return dist;
}
```

```
int main() {
    int n, m, src;
    cin >> n >> m >> src;

    vector<Edge> edges(m);
    for (int i = 0; i < m; i++) {
        cin >> edges[i].from >> edges[i].to >> edges[i].weight;
    }

    vector<int> dist = bellman_ford(n, src, edges);

    if (dist.empty()) cout << "Negative cycle detected\n";
    else {
        for (int i = 0; i < n; i++) {
            cout << src << " -> " << i << " : "<< dist[i] << "\n";
        }
    }

    return 0;
}
```

1-B: Dijkstra

Adjacency list representation of Graph - Time: O((E+V)logV) Space: O(E+V)

```
void dijkstra(vector<vector<pair<int, int>>> &adj, int start, vector<int> &dist){
    int n = adj.size();
    vector<bool> visited(n, false);

    // To select the node with minimum distance
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;
    pq.push(make_pair(0, start));

    while (!pq.empty()) {
        int u = pq.top().second, d = pq.top().first;
        pq.pop();
        if(visited[u]) continue;
        visited[u] = true;

        // Update the distance to all the neighbors of 'u' from source
        for (auto &v : adj[u]) {
            if (!visited[v] && dist[u] + v.second < dist[v.first]) {
                dist[v.first] = dist[u] + v.second;
                pq.push(make_pair(dist[v.first], v.first));
            }
        }
    }
}
```

```
int main() {
    int n, m, src;
    cin >> n >> m >> src;

    vector<vector<pair<int, int>>> adj(n);
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u].push_back({v, w});
    }

    vector<int> dist(n, INT_MAX);
    dist[src] = 0;
    dijkstra(adj, src, dist);

    // Print shortest distances from source
    for (int i = 0; i < n; i++) {
         cout << src << " -> " << i << " : " << dist[i] << "\n";
    }
    return 0;
}
```

1-C: DAG: Shortest Path

Adjacency list representation of Graph- Time: O(V+E) Space: O(V+E)

```
void topsort(int u, stack<int>& s, vector<bool>& vis, vector<vector<pair<int,
int>>> &adj) {
    vis[u] = true;
    for (auto v : adj[u]) {
        if (!vis[v.first])
            topsort(v.first, s, vis, adj);
    }
    s.push(u);
}
```

```
void shortestPath(int start, vector<int> &dis, int n, vector<vector<pair<int,
int>>> &adj) {
    vector<bool> vis(n, false);
    stack<int> s;

    for (int i = 0; i < n; i++) if (!vis[i]) topsort(i, s, vis, adj);

    while (!s.empty()) {
        int u = s.top();
        s.pop();
        if (dis[u] != INF) {
            for (auto v : adj[u]) {
                if (dis[v.first] > dis[u] + v.second)
                    dis[v.first] = dis[u] + v.second;
            }
        }
    }
}
```

```
int main() {
    int n, m, src;
    cin >> n >> m >> src;
    vector<vector<pair<int, int>>> adj(n);
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u].push_back({v, w});
    }
    vector<int> dist(n, INF);
    dis[src] = 0;
    shortest_path(src, dist, n, adj);
    // Print shortest distances from source
    for (int i = 0; i < n; i++) {
        cout << src << " -> " << i << “ : ”<< dist[i] << "\n";
    }
    return 0;
}
```

2-B: Kahn’s Algorithm: Topological Sort

Adjacency list representation of Graph- Time: O(E+V) Space: O(E+V)

```
vector<int> kahn_algorithm(vector<vector<int>>& adj, vector<int>& indegrees) {
    vector<int> result;

    queue<int> q;
    // add nodes with zero indegree to the queue
    for (int i = 0; i < indegrees.size(); i++) {
        if (indegrees[i] == 0) q.push(i);
    }
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        result.push_back(u);
        // decrease the indegree of neighbors
        for (auto v : adj[u]) {
            indegrees[v]--;
            if (indegrees[v] == 0) {
                q.push(v);
            }
        }
    }
    // check for cycle (All the nodes in a cycle will have indegree > 0)
    if (result.size() != indegrees.size()) {
        // return empty vector to indicate cycle
        result.clear();
    }
    return result;
}
```

```
int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> graph(n);
    vector<int> indegrees(n, 0);
    // build the graph and compute the indegrees
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        graph[a].push_back(b);
        indegrees[b]++;
    }
    vector<int> result = kahn_algorithm(graph, indegrees);

    if (result.empty()) {
        cout << "Cycle detected!" << endl;
    } else {
        cout << "Topological order:" << endl;
        for (int i = 0; i < result.size(); i++) cout << result[i] << " ";
        cout << endl;
    }
}
```

3-A: Detect Cycle - <u>Undirected</u> Graph (Using DFS)

Adjacency list representation of Graph - Time: O(E+V)   Space: O(E+V)

```
bool dfs(vector<vector<int>> adj, int u, vector<bool>& visited, int parent){
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v]) {
            if (dfs(adj, v, visited, u)) return true;
        }
        // If adjacent node is visited and not parent, cycle found
        else if (v != parent) return true;
    }
    // No cycle found
    return false;
}
```

```
int main() {
    int V, E;
    // Take input of number of vertices and edges
    cin >> V >> E;

    vector<vector<int>> adj(V);

    int u, v;
    for (int i = 0; i < E; i++) {
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    vector<bool> visited(V, false);

    for (int u = 0; u < V; u++) {
        if (!visited[u]) {
            if (dfs(adj, u, visited, -1)){
                cout << "Cycle detected\n";
                return 0;
            }
        }
    }

    cout << "No cycle detected\n";
    return 0;
}
```

3-B: Detect Cycle - Directed Graph (Using DFS)

Adjacency list representation of Graph- Time: O(E+V)   Space: O(E+V)

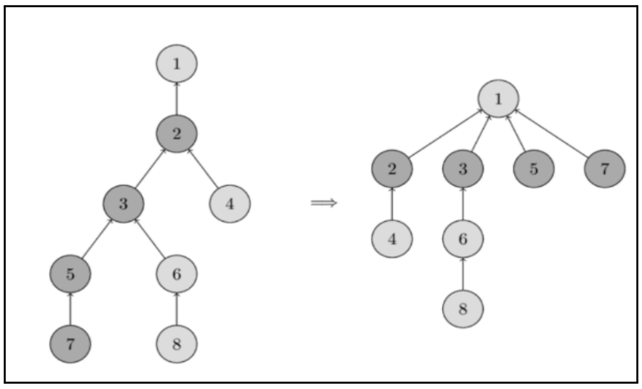
```
bool dfs(int src, vector> &graph, vector &visited, vector &dfsVis){
    visited[src] = 1;
    dfsVis[src] = 1;

    for(auto e: graph[src]){
        if(!visited[e]){
            if(dfs(e, graph, visited, dfsVis)) return true;
        // By the following condition we are checking is it part of
        // the current path being traced or not?
        // If YES then we have reached the node again suggesting cycle
        }else if(dfsVis[e]) return true;
    }

    // Remove the vertex to show that this node is no longer part of the path being traced
    dfsVis[src] = 0;
    return false;
}
```

```
int Solution::solve(int A, vector<int> &B) {
    vector visited(A+1, 0);
    // 'dfsVis' will have elements of current path being traced
    vector dfsVis(A+1, 0);

    // Create Adjacency list representation of B
    vector<vector<int>> graph(A+1);
    for(auto v: B){
        graph[v[0]].push_back(v[1]);
    }

    for(int i=1; i<=A; i++){
        if(dfs(i, graph, visited, dfsVis)) return 1;
    }

    return 0;
}
```

3-C: Detect Cycle - <u>Undirected</u> Graph (Using Union Find Algorithm)

Adjacency list representation of Graph- Time: O(ElogV) Space: O(E+V)

```
int find(int x, vector<int> &parent) {
    if (parent[x] == x) {
        return x;
    }
    return find(parent[x], parent);
    // return parent[x] = find(parent[x], parent); // This line does the following
}
```



```
void union(int Px, int Py, vector<int> &parent, vector<int> &rank) {
    if (rank[px] < rank[py]) {
        parent[px] = py;
        rank[py]++;
    }else if (rank[py] < rank[px]) {
        parent[py] = px;
        rank[px]++;
    }else{
        parent[px] = py;
        rank[py]++;
    }
}
```

```
bool isCyclic(vector<pair<int, int>> edges, int V) {
    vector<int> parent(V);
    vector<int> rank(V, 0);
    for (int i = 0; i < V; i++) parent[i] = i;
    for(auto edge : edges) {
        int px = find(edge.first, parent);
        int py = find(edge.second, parent);
        if(px == py) return true;
        union(px, py, parent, rank);
    }
    return false;
}
```

```cpp
int main() {
    int V, E;
    cout << "Enter the number of vertices in the graph: ";
    cin >> V;
    cout << "Enter the number of edges in the graph: ";
    cin >> E;
    vector<pair<int, int>> edges;
    cout << "Enter the edges of the graph (source, destination): " << endl;
    for(int i = 0; i < E; i++) {
        int u, v;
        cin >> u >> v;
        edges.push_back(make_pair(u, v));
    }
    if(isCyclic(edges, V))
        cout << "The graph contains a cycle." << endl;
    else
        cout << "The graph does not contain a cycle." << endl;
    return 0;
}
```

4-A: Minimum Spanning Tree - Prim's algorithm

Adjacency list representation of Graph - Time: O(ElogV)   Space: O(E+V)

```
int prim(int start, vector<vector<pair<int, int>>> &adj, vector<bool> &visited){
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;
    pq.push(make_pair(0, start));
    int mst_cost = 0;

    while (!pq.empty()) {
        auto p = pq.front();
        pq.pop();
        int w = p.first, u = p.second,;
        if (vis[u]) continue;
        vis[u] = true;
        mst_cost += w;

        // following step is executed degree(u) times
        // Sum of degree of all the vertices is upper bounded by E
        for (auto x : adj[u]) {
            int v = x.second, weight = x.first;
            if (!vis[v])
                pq.push(make_pair(weight, v));
        }
    }
    return mst_cost;
}
```

```
int main() {
    int n, m;
    cin >> n >> m;

    vector<vector<pair<int, int>>> adj(n);
    vector<bool> visited(n);

    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u].push_back({w, v});
        adj[v].push_back({w, u});
    }

    int start_node = 0;
    int mst_cost = prim(start_node, adj, visited);
    cout << "Minimum Spanning Tree cost: " << mst_cost << endl;
    return 0;
}
```

4-B: Minimum Spanning Tree - Kruskal's Algorithm

Adjacency list representation of Graph- Time: O(ElogE) Space: O(E+V)

```
// Define a structure to represent edges in the graph
struct Edge {
    int src, dest, weight; };
```

```
// Define a 'comparison function' for sorting the edges based on their weight
bool compareEdges(Edge a, Edge b) {
    return a.weight < b.weight;   }
```

```
// Find parent of a vertex in the Union-Find data structure
int find(int x, vector<int> &parent) {
    if (parent[x] == x) return x;
    return find(parent[x], parent);
}
```

```
// Union two sets in the Union-Find data structure
void union(int px, int py, vector<int> &parent, vector<int> &rank) {
    if (rank[px] < rank[py]) {
        parent[px] = py;
    } else if (rank[py] < rank[px]) {
        parent[py] = px;
    } else {
        parent[px] = py;
        rank[py]++;
    }
}
```

```
void kruskalMST(vector<Edge> &edges, int V, int E) {
    // Sort edges in increasing order of their weight
    sort(edges, edges + E, compareEdges);

    // Initialize parent and rank arrays for Union-Find data structure
    vector<int> parent(V, -1);
    vector<int> rank(V, 0);
    for (int i = 0; i < V; i++) parent[i] = i;

    // Initialize MST array to store edges of minimum spanning tree
    vector<Edge> MST(V-1, -1);
    int mstIndex = 0;
    int no_of_edges = 0;

   // Add edges to MST until V-1 edges have been added or all edges have been considered
    while (mstIndex < V - 1 && no_of_edges < E) {
        Edge currEdge = edges[no_of_edges++];
        int srcParent = find(currEdge.src, parent);
        int destParent = find(currEdge.dest, parent);
        if (srcParent != destParent) {
            MST[mstIndex++] = currEdge;
            union(srcParent, destParent, parent, rank);
        }
    }

    // Display edges of MST and their weights
    for (int i = 0; i < V - 1; i++) {
        cout << MST[i].src << " - " << MST[i].dest << " : " << MST[i].weight << endl;
    }
}
```

```
int main() {
    int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;
    vector<Edge> edges(E);
    for (int i = 0; i < E; i++) {
        cout << "Enter source, destination and weight of edge " << i+1 << ": ";
        cin >> edges[i].src >> edges[i].dest >> edges[i].weight;
    }
    kruskalMST(edges, V, E);
    return 0;
}
```

while(mstIndex < V-1 && edgeIndex < E): The condition "mstIndex < V - 1" ensures that the loop continues until the desired number of edges is added to the MST. The condition "no_of_edges < E" ensures that the loop terminates if there are no more edges left to consider. The condition "no_of_edges < E" serves as a safety check to prevent an infinite loop in case the number of edges (E) is smaller than the expected number of edges needed to form a minimum spanning tree (V - 1).

Maximum Flow: **Ford Fulkerson**'s Algorithm

Adjacency matrix representation of Graph- Time: O(f_max*|V^2|) Space: O(V*V)

```
const int INF = 1e9;
// capacity and flow matrices
vector<vector<int>> c(n, vector<int>(n,0));
vector<vector<int>> f(n, vector<int>(n,0));
vector<bool> visited(n, false);
```

```
// 'flow' keeps track of Δ_f(π)
int dfs(int s, int t, int flow) {
    if (s == t) return flow;
    visited[s] = true;
    for (int v = 0; v < V; v++) {
        if (!visited[v] && c[s][v] - f[s][v] > 0) {
            int aug = dfs(v, t, min(flow, c[s][v] - f[s][v]));
            if (aug > 0) {
                f[s][v] += aug;
                f[v][s] -= aug;
                return aug;
            }
        }
    }
    return 0;
}
```

```
int maxFlow(int s, int t) {
    int flow = 0;
    while (true) {
        visited.resize(n, false);
        int aug = dfs(s, t, INF);
        if (aug == 0) break;
        flow += aug;
    }
    return flow;
}
```

```
int main() {
    int V, E;
    cin >> V >> E;
    for (int i = 0; i < E; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        c[u][v] += w;
        c[v][u] += w; // if the graph is undirected
    }
    int s, t;
    cin >> s >> t;
    cout << "Maximum flow: " << maxFlow(s, t) << endl;
    return 0;
}
```

```
// Time complexity explanation:
//    If each augmenting path increases the flow by 1. In the worst case DFS
//    will take O(V^2) to find an augmenting path.

// This time complexity is not polynomial because it’s dependent on f_max which
// can be any amount. Though the time complexity is not polynomial it guarantees
// termination as long as all the capacities are integers. There is no guarantee
// on termination if the capacities are ‘non-integers’.

// Edmonds-Karp algorithm
//    A variation of ford fulkerson guarantees termination because the time
//    complexity O(VE²) is independent of the “f_max”.
```

Connectivity: **Bipartite/ Two Color Graph**

Adjacency list representation of Graph- Time: O(E+V) Space: O(E+V)

```
bool isBipartite(int u, vector<bool> &visited, vector<int> &color,
vector<vector<int>> &adj){
    visited[u]=true;
    for(auto v: adj[u]){
        if(!visited[v]){
            color[v]= !color[u];
            if(!isBipartite(v, visited, color, adj)) return false;
        }else if(color[u]==color[v]) return false;
    }
    return true;
}
```

```
int Solution::solve(int A, vector<vector<int>> &B) {
    vector<bool> visited(A+1, 0);
    vector<int> color(A+1, 0);
    vector<vector<int>> adj(A+1);

    // Creating adjacency list representation for B
    for(auto p: B){
        adj[p[0]].push_back(p[1]);
        adj[p[1]].push_back(p[0]);
    }

    for(int i=1; i<=A; i++){
        if(!visited[i]){
            // NOTE:
            //   We don't do 'if(isBipartite())' because then if all the elements in
            //   one connected component can be divided into two teams we don't know
            //   about the other components.
            if(!isBipartite(i, visited, color, adj)) return 0;
        }
    }
    return 1;
}
```

Connectivity: **Articulation Points / Cut Vertices** (**Tarjan's** Algorithm)

Adjacency list representation of Graph- Time: O(V+E) Space: O(V+E)

```
void tarjan(int u, int parent, vector<int> adj[], vector<int>& disc, vector<int>&
low, vector<bool>& ap, int& time) {
    disc[u] = low[u] = ++time;
    int children = 0;

    // If 'u' is not the root of the DFS tree
    for (int v : adj[u]) {
        if (v == parent) continue;
        if (disc[v] == -1) {
            children++;
            tarjan(v, u, adj, disc, low, ap, time);
            // Since 'v' is descendant of 'u' and any vertex reachable from 'v'
            // is also reachable form 'u'
            low[u] = min(low[u], low[v]);
            // low[v] >= disc[u] suggest no back edge from 'v' to any ancestor of 'u'
            if (parent != -1 && low[v]  >= disc[u]) ap[u] = true;
        } else {
            low[u] = min(low[u], low[v]);
        }
    }

    // If 'u' is the root of the DFS tree
    if (parent == -1 && children > 1) ap[u] = true;
}
```

```
int main() {
    int n, m;
    cin >> n >> m;
    vector<int> adj[n];

    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    vector<int> disc(n, -1), low(n, 0);
    vector<bool> ap(n, false);
    int time = 0;

    for (int i = 0; i < n; i++) {
        if (disc[i] == -1) {
            tarjan(i, -1, adj, disc, low, ap, time);
        }
    }

    cout << "Articulation points:\n";
    for (int u : ap) {
        cout << u << " ";
    }
    return 0;
}
```

Connectivity: **Bridges** (Tarjan's algorithm)

Adjacency list representation of Graph- Time: O(V+E) Space: O(V+E)

```
void tarjan(int u, int parent, vector<int> adj[], vector<int>& disc, vector<int>&
low, vector<pair<int, int>> &bridges, int& time) {
    disc[u] = low[u] = ++time;

    for (int v : adj[u]) {
        if (v == parent) continue;
        if (disc[v] == -1) {
            tarjan(v, u, adj, disc, low, bridges, time);
            low[u] = min(low[u], low[v]);
            // If the following condition is true then we can't reach 'v' without 'u'
            //    low[v] >= disc[u] means any vertex reachable from 'v' was discovered
            //     after 'u' was discovered. So, you cannot visit 'v' or any vertex reachable
            //    from 'v' without visiting 'u'
            if (low[v] >= disc[u]) bridges.push_back({u,v});
        } else {
            low[u] = min(low[u], low[v]);
        }
    }
}
```

```
int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> adj(n);

    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    vector<int> disc(n, -1), low(n, 0);
    vector<pair<int, int>> bridges;
    int time = 0;

    for (int i = 0; i < n; i++) {
        if (disc[i] == -1) {
            tarjan(i, -1, adj, disc, low, bridges, time);
        }
    }

    cout << "Bridges:\n";
    for (auto p : bridges) {
        cout << p.first << "->" << p.second;
        cout << "\n";
    }
    return 0;
}
// low[u]: Represents the smallest discovery time of any node reachable from 'u',
// including 'u' itself. It helps in identifying the lowest ancestor that can be
// reached from the subtree rooted at 'u'
```