Graph

Algorithm	When to Use	Maximum Constraints
Dijkstra	Finding the shortest path from a single source in graphs. *Can't* detect negative weight cycles.	- Time Complexity: $O((V+E)\log V)$ with a priority queue (Fibonacci heap can reduce it further). Maximum Constraints $V \approx 10^5, E \approx 10^6$
Bellman-Ford	Best for finding shortest paths from a single source *Can* detect negative weight cycles.	Time Complexity: $O(VE)$ Maximum Constraints: $V \approx 5000$, $E \approx 10^6$
Floyd-Warshall	Best for all-pairs shortest paths in dense graphs. *Can* detect negative weight cycles.	Time Complexity: $O(V^3)$ Maximum Constraints: $V \approx 400 - 500$

Dijkstra Algorithm

Finding the shortest path from a single source to all nodes.
it's a normal BFS but with a priority queue.

```cpp
// Find shortest path from node 1 to all nodes.
void Ramez() {
    int n, m; cin >> n >> m;
    vector<vector<pii>> adj(n + 1);

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

    vi vis(n + 1), dis(n + 1);
    priority_queue<pii, vector<pii>, greater<pii>> pq; // {cost, node}
    pq.push({ 0, 1 });

    while (!pq.empty()) {
        auto [parentCost, parent] = pq.top(); pq.pop();
        if (vis[parent]) continue;
        vis[parent] = 1; dis[parent] = parentCost;

        for (auto [child, childCost] : adj[parent]) {
            if (!vis[child]) {
                pq.push({ parentCost + childCost, child });
            }
        }
    }

    for (int i = 1; i <= n; i++) {
        cout << dis[i] << " ";
    }
}
```

K-Dijkstra

```cpp
// Find the k shortest routes from 1 to n
void Ramez() {
    int n, m, k; cin >> n >> m >> k;
    vector<vector<pii>> adj(n + 1);

    for (int i = 0; i < m; i++) {
        int a, b, c; cin >> a >> b >> c;
        adj[a].push_back({ b, c });
    }
```

```
    vector<vi> dis(n + 1);
    priority_queue<pii, vector<pii>, greater<pii>> pq; // {cost, node}
    pq.push({ 0, 1 });

    while (!pq.empty()) {
        auto [parentCost, parent] = pq.top(); pq.pop();
        if (dis[parent].size() ⩾ k) continue;
        dis[parent].push_back(parentCost);

        for (auto [child, childCost] : adj[parent]) {
            pq.push({ parentCost + childCost, child });
        }
    }

    cout << dis[n] << "\n";
}
```

Floyd-Warshall Algorithm

Best for finding shortest paths from a single source to all nodes

```
// Find shortest path from all nodes to all nodes
void Ramez() {
    int n, m, q; cin >> n >> m >> q;

    vector<vi> dis(n + 1, vi(n + 1, LLONG_MAX));
    for (int i = 1; i ⩽ n; i++) {
        dis[i][i] = 0;
    }

    for (int i = 0; i < m; i++) {
        int a, b, c; cin >> a >> b >> c;
        dis[a][b] = min(dis[a][b], c);
        dis[b][a] = min(dis[b][a], c);
    }

    for (int k = 1; k ⩽ n; k++) {
        for (int i = 1; i ⩽ n; i++) {
            for (int j = 1; j ⩽ n; j++) {
                if (dis[i][k] < LLONG_MAX && dis[k][j] < LLONG_MAX)
                    dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
            }
        }
    }


    while (q--) {
        int a, b; cin >> a >> b;
        if (dis[a][b] == LLONG_MAX) cout << "-1\n";
        else cout << dis[a][b] << "\n";
    }
```

Bellman-Ford Algorithm (Maximum Score)

Best for all-pairs shortest paths in dense graphs.

```
// Find maximum score to travel from 1 to n (negative allowed, infinite cycles allowed)
struct edge {
    int u, v, w;
};

void Ramez() {
    int n, m; cin >> n >> m;

    vector<edge> edges;

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

    vi score(n + 1, LLONG_MIN);
    score[1] = 0;
```

```
// Relaxation (n - 1) times
for (int i = 1; i ≤ n - 1; i++){
    for (int j = 0; j < m; j++){
        auto [u, v, w] = edges[j];
        if(score[u] ≠ LLONG_MIN){
            score[v] = max(score[v], score[u] + w);
        }
    }
}


/*
After the initial relaxation steps, we check if any edge can still be relaxed.
If it can, that means there's a cycle (specifically a "positive cycle" for maximizing the score)
that can improve the score.
*/
vector<bool> hasPositiveCycle(n + 1, false);
for (int i = 0; i < m; i++){
    auto [u, v, w] = edges[i];
    if(score[u] ≠ LLONG_MIN && score[v] < score[u] + w){
        hasPositiveCycle[v] = true;
    }
}


    /*
    However, simply detecting an edge that can be relaxed doesn't tell us which vertices
    might be affected downstream by this cycle.
    The propagation loop iterates over all edges several times (in this case, n times)
    to "spread" the effect of the positive cycle
    */
    for (int i = 1; i ≤ n; i++) {
        for (int j = 0; j < m; j++) {
            auto [u, v, w] = edges[j];
            if(hasPositiveCycle[u]) hasPositiveCycle[v] = true;
        }
    }

    if(hasPositiveCycle[n]) cout << -1 << "\n";
    else cout << score[n] << "\n";
}
```

Finding a cycle in graph

```
// Searching for a cycle
void Ramez() {
    int n, m; cin >> n >> m;
    vector<vi> adj(n + 1);
    for (int i = 0; i < m; i++) {
        int a, b; cin >> a >> b;
        adj[a].push_back(b);
    }

    // Color: 0 = unvisited, 1 = in-stack, 2 = done
    vi color(n + 1, 0), parents(n + 1, -1), cycle;
    bool found = false;

    function<bool(int)> dfs = [&](int parent) → bool {
        color[parent] = 1;
        for (int child : adj[parent]) {
            if (color[child] == 0) {
                parents[child] = parent;
                if (dfs(child)) return true;
            }
            else if (color[child] == 1) {
                // back-edge u→v ⇒ found a cycle
                found = true;
                cycle.push_back(child);
                for (int x = parent; x ≠ child; x = parents[x])
                    cycle.push_back(x);
                cycle.push_back(child);
                reverse(all(cycle));
                return true;
            }
        }
```

```
        color[parent] = 2;
        return false;
    };

    for (int i = 1; i <= n && !found; i++) {
        if (color[i] == 0) dfs(i);
    }

    if (!found) {
        cout << "IMPOSSIBLE\n";
    } else {
        cout << cycle.size() << "\n";
        cout << cycle << "\n";
    }
}
```

Topoligical Sort

A topological sort takes a directed acyclic graph (DAG) and produces
a linear ordering of its vertices such that for every directed edge u → v, u comes before v in that order.
Returns a vector of nodes in a valid order; if a cycle exists, the size will be < n.
Useful for: Scheduling with Dependencies, Course Prerequisites,

```
vi topologicalSort(int n, vector<vi>& adj, vi& inDeg) {
    queue<int> q;
    for (int i = 1; i <= n; i++) {
        if (inDeg[i] == 0)
            q.push(i);
    }

    vi order;

    while (!q.empty()) {
        int parent = q.front(); q.pop();
        order.push_back(parent);
        for (int child : adj[parent]) {
            if (--inDeg[child] == 0)
                q.push(child);
        }
    }

    return order;
}
```

DAG longest-path DP

What is the maximum number of cities I can visit on any directed path from 1 to n in a graph with no cycles DAG?
In a DAG, once you've placed nodes in topological order, all ways to reach a city *u* must come from cities *before u* in that order. That means when you process *u*, you already know the best (longest) way to get there.

```
    vi order = topologicalSort(n, adj, inDeg);

    // dp[i] = the maximum number of cities you can visit on any path from city 1 ending at city i
    vi dp(n + 1, -1), parent(n + 1, -1);
    dp[1] = 1;

    for (int u : order) {
        if (dp[u] < 0)    // not reachable from 1
            continue;

        for (int v : adj[u]) {
            if (dp[u] + 1 > dp[v]) {
                dp[v] = dp[u] + 1;
                parent[v] = u;
            }
        }
    }

    if (dp[n] < 0) {
```

```
        cout << "IMPOSSIBLE\n";
        return;
    }
```