Graphs

DFS

- Recursive way of Graph Exploration
- Notice *Level, Choices, Check, Move*

```
void dfs(int node) {
    visited[node] = 1;

    for(auto v: adj[node]){
        if(!visited[v]){
            dfs(v);
        }
    }
}
```

BFS

- **Mark node as visited before adding to queue**

```
queue<int> q;
visited[sc] = 1;
q.push(sc);

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

    for (auto v: adj[node]) {
        if (!visited[v]) {
            visited[v] = true;
            q.push(v);
        }
    }
}
```

0-1 BFS

Edge weights are either 0 or 1. We use a deque to ensure that the 0-weight nodes are processed first.

- Implemented using a **deque**
- Which side to push node to depends upon weight

Multisource BFS

Dijkstra

We've weighted graphs now.
We use priority_queue to process the node with smallest distance at every step.

But there could be same node in the pq with multiple distances (multiple paths to that node exists), but you've to ensure that the node gets explored only once with the smallest distance.

- **Remember to check vis after pop from pq**
- and mark it visited if not already

```
vector<vector<pair<int, int>>> adj;
vector<int> dist(n+1, 1e9);
vector<bool> visited(n+1, false);

void dijkstra(int sc){
    priority_queue<pair<int, int>> pq;   // {-dist, node} for min heap

    pq.push({0, sc});
    dist[sc]=0;

    while(!pq.empty()){
        int curr = pq.top().second; pq.pop();

        // remember to check visited
        if(visited[curr]) continue;
        visited[curr]=1;

        for(auto v: adj[curr]){
            if(dist[v.first] > dist[curr] + v.second){
                dist[v.first] = dist[curr] + v.second;
                pq.push({-dist[v.first], v.first});
            }
        }
    }
}
```

Topological Ordering

Ordering nodes so that edge always go forward in direction.

DFS Way

- Push at the end of dfs recursion, reverse array later.

```
vector<int> topo;
void dfs(int node) {
    visited[node] = 1;

    for(auto v: adj[node]){
```

```
        if(!visited[v]){
            dfs(v);
        }
    }

    topo.push_back(node);
}

reverse(topo.begin(), topo.end());
```

Kahn's Algo (BFS Way)

- We maintain indeg of each node
- Push nodes with indeg=0 to queue (topo starts with 0 indeg)
- **Use a priority_queue for lexicographical order**, pushing `-node` to pq

```
vector<int> indeg;
vector<int> topo;

void kahn(){
    queue<int> q;
    for(int i=1; i<=n; i++){
        if(indeg[i]==0) q.push(i);
    }

    while(!q.empty()){
        int curr = -q.front(); q.pop();
        topo.push_back(curr);       // topo starts with a node with indeg=0

        // del curr - update indeg of its neighbours
        for(auto v: adj[curr]){
            indeg[v]--;
            if(indeg[v]==0) q.push(v);
        }
    }
}
```

Bellman Ford

- Graphs with negative weight
- Relax each edge n-1 times
- **Cycle Finding:** Relax once more. If any distance decreases, there exists a cycle.

```
vector<pair<pair<int, int>, int>> edges;    // {u, v}, w
vector<ll> dist;

void bf(){
    dist.resize(n+1, INF);
```

```
    dist[1] = 0;

    for(int i=0; i<n-1; i++){       // n-1 times
        for(auto e: edges){         // relax all edges
            int u = e.first.first, v = e.first.second, w = e.second;
            dist[v] = min(dist[v], dist[u]+w);
        }
    }

    ///// cycle check /////
    bool negCycle = false;
    for(auto e: edges){                 // relax once again
        int u = e.first.first, v = e.first.second, w = e.second;
        if(dist[v] > dist[u]+w){        // if dist dec further, so -ve cycle
            negCycle = true;
            break;
        }
    }
}
```

Floyd Warshall [All pairs Shortest Path]

```
vector<vector<ll>> dist;    // adjacency matrix

void floydWarshall(){
    for(int k=1; k<=n; k++){        // via kth intermediate node
        for(int i=1; i<=n; i++){    // relaxing each pair of node
            for(int j=1; j<=n; j++){
                dist[i][j] = min(dist[i][j], dist[i][k]+dist[k][j]);
            }
        }
    }
}
```

- At each iter k, we calculate APSP USING NODES 1.....k for all pairs.
  The core logic is to check if using vertex k as an intermediate node provides a shorter path from i to j. If it does, we update dist[i][j].