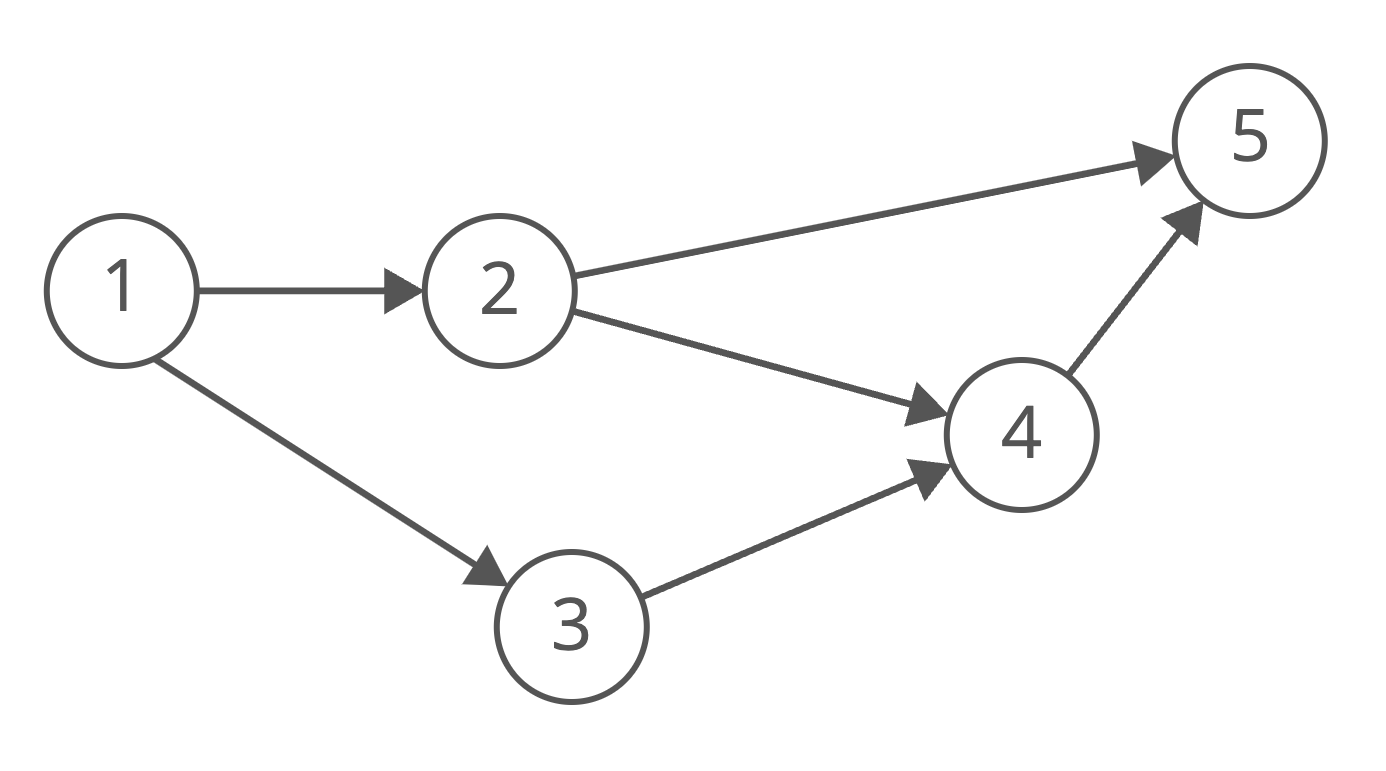
Week #4 (Graphs II)

# **| Topological Sort**

The topological sort algorithm takes a directed graph and returns an array of the nodes where each node appears before all the nodes it points to.

For Example:

Given a directed graph, print the topological sort of this graph.



Since node 1 points to nodes 2 and 3, node 1 appears before them in the ordering. And, since nodes 2 and 3 both point to node 4, they appear before it in the ordering.

So [1, 2, 3, 4, 5] would be a topological ordering of the graph.

| Input:  5 6 1 3 1 2 3 4 2 4  2 5 4 5 |
| --- |

| Output: 1 2 3 4 5 |
| --- |

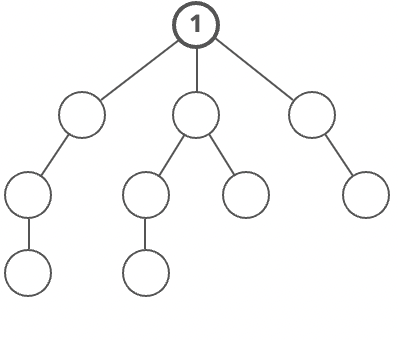
Code:

| #include <bits/stdc++.h>  using namespace std; const int N = 1e3 + 3; vector<int> graph[N]; stack<int> ans; bool vis[N];  void dfs(int node) {  for (auto i: graph[node]) {  if (vis[i])continue;  dfs(i);  }  vis[node] = true;  ans.push(node); }  int main() {  int n, m;  cin >> n >> m;  for (int i = 0; i < m; i++) {  int u, e;  cin >> u >> e;  graph[u].push\_back(e);  }  for (int i = 1; i <= n; i++) {  if (!vis[i])dfs(i);  }  while (!ans.empty()) {  cout << ans.top() << " ";  ans.pop();  }  return 0; } |
| --- |

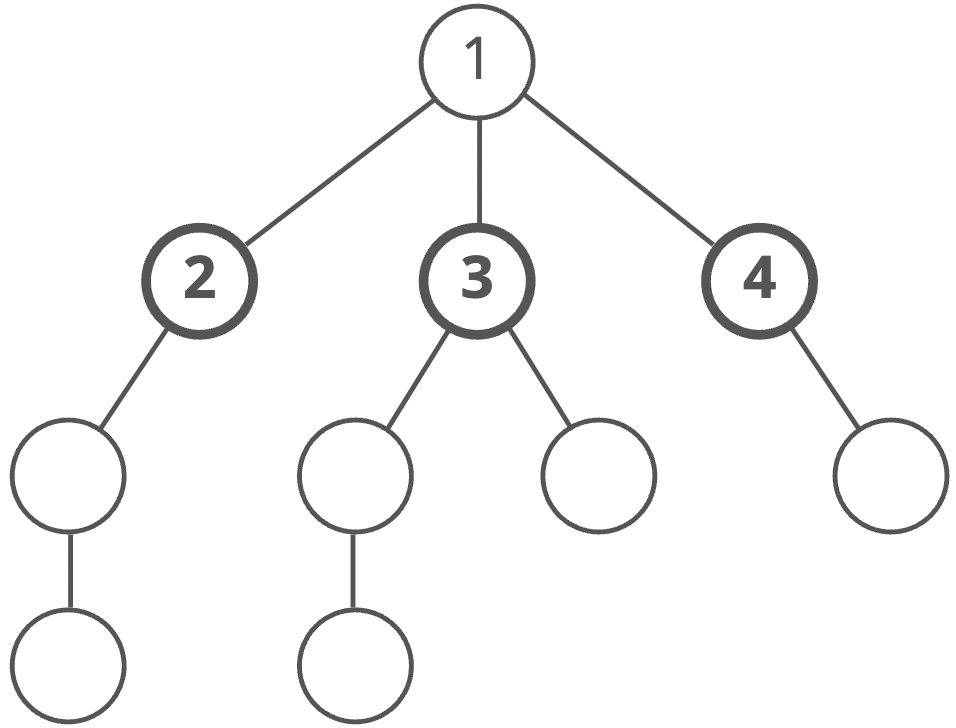
# **| BFS Recap**

In a BFS, you first explore all the nodes one step away, then all the nodes two steps away, etc.

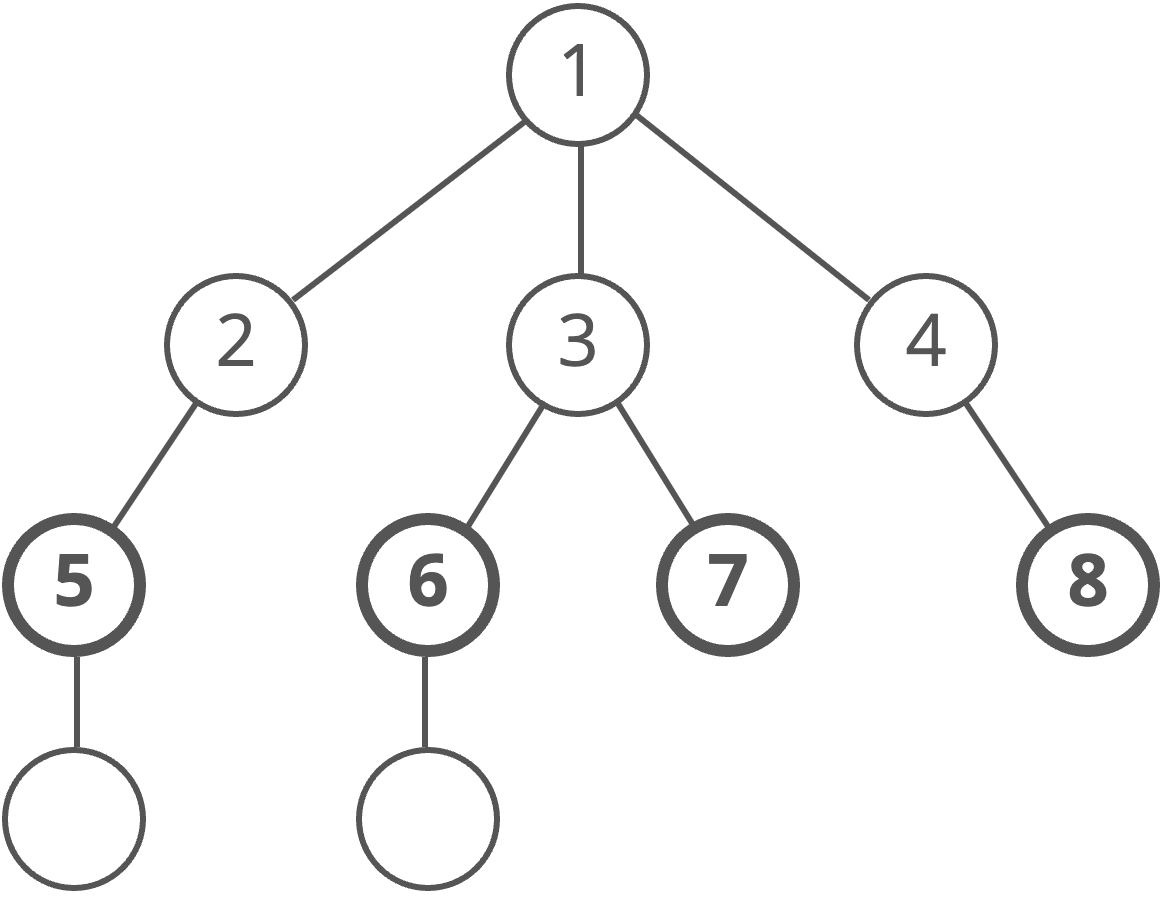
Here's a how a BFS would traverse this tree, starting with the root:



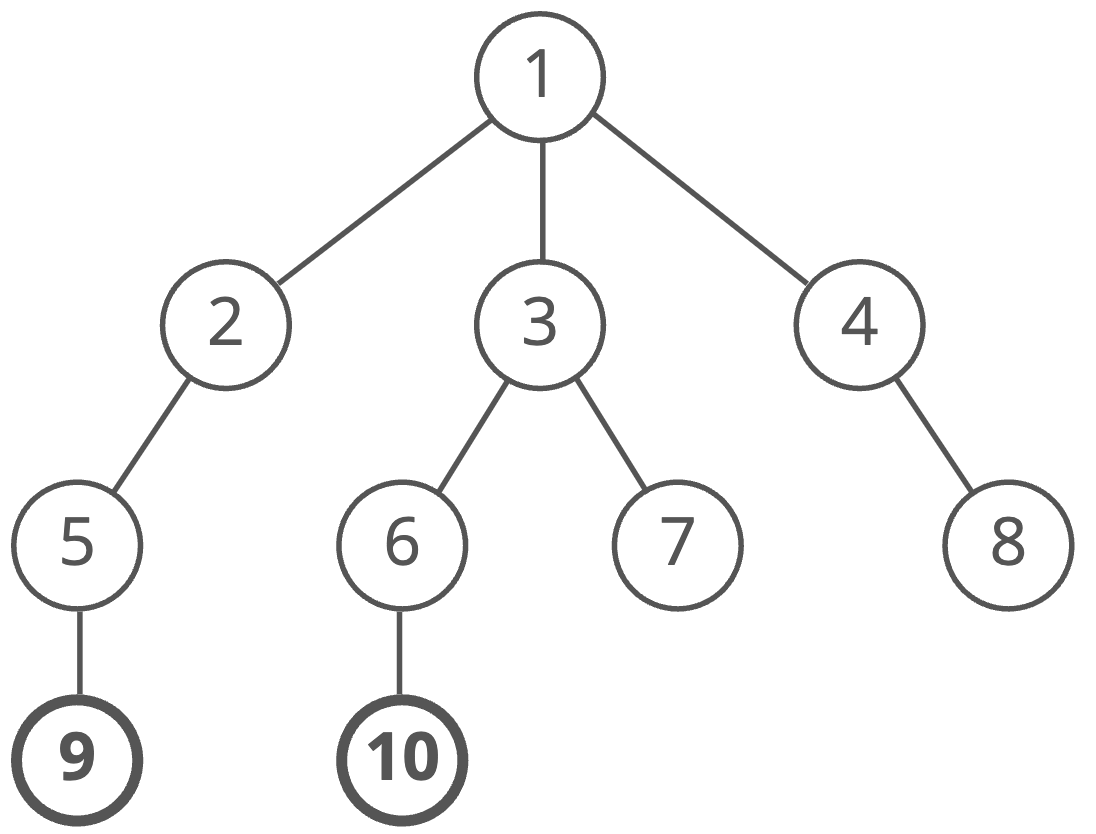
We'd visit all the immediate children (all the nodes that're one step away from our starting node):



Then we'd move on to all those nodes' children (all the nodes that're two steps away from our starting node):



And so on:



Until we reach the end.

# **| Dijkstra**

* What Is Dijkstra:

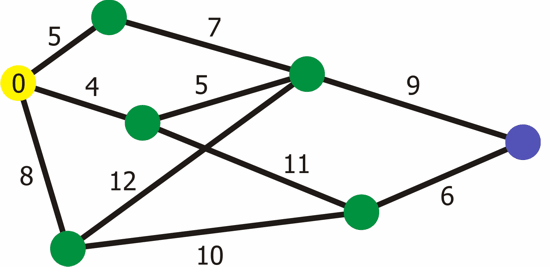
Dijkstra's original algorithm found the shortest path between two given nodes, but a more common variant fixes a single node as the "source" node and finds the shortest paths from the source to all other nodes in the graph, producing a shortest path [tree](https://en.wikipedia.org/wiki/Shortest-path_tree).

* How It Works:

It is similar to BFS with some differences. First, instead of using a regular queue, you use a priority queue storing the node and the cost to get to this node from some direction and sort it ascendingly by cost.

Also, we need an array to save the minimum cost to reach each node from the

Source. By sorting the queue ascendingly, we pick the node with the lowest cost each time.



Code:

| //node,cost vector<pair<int,int>> adj[N]; vector<ll> dist(N,1e18); void dijkstra(int n) {  priority\_queue<pair<ll,int>> pq;  pq.push({0,1});  dist[1] = 0;  while (!pq.empty())  {  int node = pq.top().second;  ll cost = -pq.top().first;  pq.pop();  if ( cost > dist[node])  continue;  for( auto &[u,c] : adj[node])  {  ll newcost = cost +c;  if ( newcost < dist[u])  {  dist[u] = newcost;  pq.push({-dist[u],u});  }  }  }  } |
| --- |

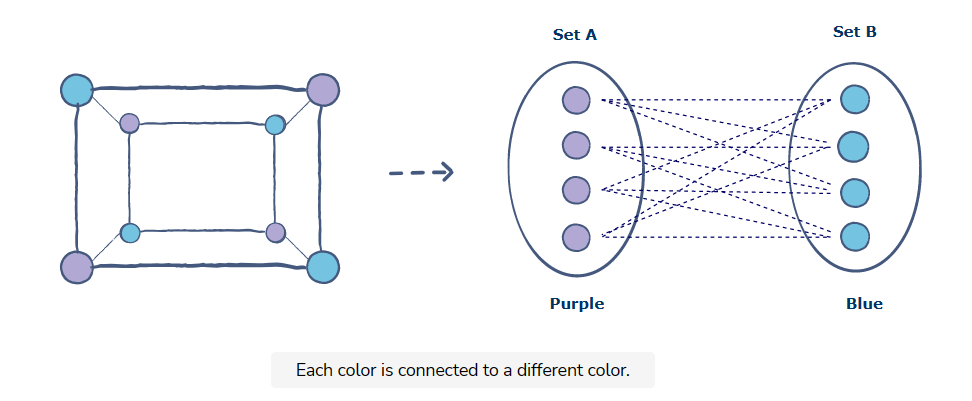
The time complexity for this algorithm: , where is the number of nodes and is the number of edges. is the complexity of BFS and is the complexity of adding nodes to the priority queue.

**Note:** Dijkstra doesn’t work with negative weights.

# **| Bipartite Graph**

A bipartite graph is a graph which its nodes can be divided into two sets such that no two adjacent nodes belong to the same set.

Bipartite graphs are equivalent to two-colorable graphs i.e., coloring of the vertices using two colors in such a way that vertices of the same color are never adjacent along an edge. All Acyclic graphs are bipartite. A cyclic graph is bipartite if all its cycles are of even length.



Now, to check if a given graph is bipartite we will do the following :

* Assign color 1 to the source node.
* Assign color 2 to the neighbors of the source nodes.
* Assign color 1 to the neighbors of the neighbors of the source nodes and so on…
* If we find that the color of a node is the same as its neighboring node that means that the graph is not bipartite.

| vector<int> adj[100]; int color[100]; bool vis[100];  bool dfs(int node) {  vis[node] = true;  for (auto child : adj[node]) {  if (color[child] == color[node]) return false;  else if (!vis[child]) {  if (!dfs(child)) return false;  }  } } |
| --- |