# LeetCode Training Day 17 Dijkstra

When we discuss shortest path in the graph, the most common algorithm is Dijkstra. Here is the Wiki page:

<https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>

There is a animation as below:

|  |  |
| --- | --- |
| [Chart, diagram  Description automatically generated](https://en.wikipedia.org/wiki/File:Dijkstra_Animation.gif)  Dijkstra's algorithm to find the shortest path between *a* and *b*. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors. | |
| **Class** | [Search algorithm](https://en.wikipedia.org/wiki/Search_algorithm) [Greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) [Dynamic programming](https://en.wikipedia.org/wiki/Dynamic_programming)[[1]](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#cite_note-1) |
| **Data structure** | [Graph](https://en.wikipedia.org/wiki/Graph_(data_structure)) Usually used with [Priority queue](https://en.wikipedia.org/wiki/Priority_queue)/[Heap](https://en.wikipedia.org/wiki/Heap_(data_structure)) for optimization[[2]](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#cite_note-Intro-2)[[3]](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#cite_note-FibonacciH-3) |

The problem is normally we give a series of weighted edge, (ui, vi, wi), which means from ui to vi the distance is wi. And start from origin node, say 0, and go to desination node say n – 1, what is the shortest path.

The idea of Dijkstra is that we try to find the shortest path to every node along the way to the destination. We start from origin node with distance 0, then from origin point, we see what are the neighbors the origin node can reach, get distance for all them and insert into a priority queue, with shortest distance as top() (In C++ we need to make it negative since the original Priority Queue is large number first). At same time we track the distances we need to travel to these nodes. Then we use BFS algorithm, fetch the shortest distance node from the priority queue, check all its neigbours with accumulated distance, if the accumulated distance to these nodes are shorter than the ones we tracked on these nodes, then override the distance to the node, and add it as a new path in priority queue, otherwise we ignore the new paths since it is not better than what we currently found. By default, from origin node to every node, the distance is infinite. (We use INT\_MAX, for example).

During the process, you may have add the same node to the priority queue multiple times, but do not worry, since we check the accumulated distance, they will be ignored automatically.

Like what we traverse in graph using BFS or DFS, we should convert the input from edges to neighbors, so we can easily find the next node to visit.

Please note the limitation of Dijkstra algorithm. It can only process the weight graph with positive weight on the edge, if the weight can be negative, we should use Bellman -Ford algorithm and if we have multiple sources, we have to use Floyd-Warshall algorithm. Luckily the later are out of scope in interview.

## 743. Network Delay Time

Medium

You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target.

We will send a signal from a given node k. Return the time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

**Example 1:**

A picture containing clock

Description automatically generated

**Input:** times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2

**Output:** 2

**Example 2:**

**Input:** times = [[1,2,1]], n = 2, k = 1

**Output:** 1

**Example 3:**

**Input:** times = [[1,2,1]], n = 2, k = 2

**Output:** -1

**Constraints:**

* 1 <= k <= n <= 100
* 1 <= times.length <= 6000
* times[i].length == 3
* 1 <= ui, vi <= n
* ui != vi
* 0 <= wi <= 100
* All the pairs (ui, vi) are **unique**. (i.e., no multiple edges.)

### Analysis:

We use Dijkstra algorithm to process the node as the earliest time the signal can reach the node.

/// <summary>

/// Leet Code 743. Network Delay Time

///

/// Medium

///

/// You are given a network of n nodes, labeled from 1 to n. You are also

/// given times, a list of travel times as directed edges times[i] =

/// (ui, vi, wi), where ui is the source node, vi is the target node, and

/// wi is the time it takes for a signal to travel from source to target.

///

/// We will send a signal from a given node k. Return the time it takes

/// for all the n nodes to receive the signal. If it is impossible for

/// all the n nodes to receive the signal, return -1.

///

/// Example 1:

/// Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2

/// Output: 2

///

/// Example 2:

/// Input: times = [[1,2,1]], n = 2, k = 1

/// Output: 1

///

/// Example 3:

/// Input: times = [[1,2,1]], n = 2, k = 2

/// Output: -1

///

/// Constraints:

/// 1. 1 <= k <= n <= 100

/// 2. 1 <= times.length <= 6000

/// 3. times[i].length == 3

/// 4. 1 <= ui, vi <= n

/// 5. ui != vi

/// 6. 0 <= wi <= 100

/// 7. All the pairs (ui, vi) are unique. (i.e., no multiple edges.)

/// </summary>

int LeetCodeGraph::networkDelayTime(vector<vector<int>>& times, int n, int k)

{

vector<int> node\_times(n+1, INT\_MAX);

priority\_queue<pair<int, int>> pq;

vector<vector<pair<int, int>>> neighbors(n+1);

for (size\_t i = 0; i < times.size(); i++)

{

neighbors[times[i][0]].push\_back(make\_pair(times[i][1], times[i][2]));

}

pq.push(make\_pair(0, k));

node\_times[k] = 0;

while (!pq.empty())

{

pair<int, int> pair = pq.top();

pq.pop();

int time = -pair.first;

int node = pair.second;

for (size\_t i = 0; i < neighbors[node].size(); i++)

{

int next\_node = neighbors[node][i].first;

int next\_time = neighbors[node][i].second;

if (time + next\_time < node\_times[next\_node])

{

node\_times[next\_node] = time + next\_time;

pq.push(make\_pair(-node\_times[next\_node], next\_node));

}

}

}

int result = 0;

for (size\_t i = 1; i < node\_times.size(); i++)

{

if (node\_times[i] == INT\_MAX) return -1;

result = max(result, node\_times[i]);

}

return result;

}

## 1102. Path With Maximum Minimum Value

Medium

Given an m x n integer matrix grid, return *the maximum****score****of a path starting at*(0, 0)*and ending at*(m - 1, n - 1) moving in the 4 cardinal directions.

The **score** of a path is the minimum value in that path.

* For example, the score of the path 8 → 4 → 5 → 9 is 4.

**Example 1:**

Calendar

Description automatically generated

**Input:** grid = [[5,4,5],[1,2,6],[7,4,6]]

**Output:** 4

**Explanation:** The path with the maximum score is highlighted in yellow.

**Example 2:**

A picture containing text, clock, orange

Description automatically generated

**Input:** grid = [[2,2,1,2,2,2],[1,2,2,2,1,2]]

**Output:** 2

**Example 3:**

A picture containing table

Description automatically generated

**Input:** grid = [[3,4,6,3,4],[0,2,1,1,7],[8,8,3,2,7],[3,2,4,9,8],[4,1,2,0,0],[4,6,5,4,3]]

**Output:** 3

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 100
* 0 <= grid[i][j] <= 109

### Analysis:

Find the maximum value in the next step, use Dijkstra algorithm to reach the destination.

/// <summary>

/// Leet code #1102. Path With Maximum Minimum Value

///

/// Given a matrix of integers A with R rows and C columns, find the maximum

/// score of a path starting at [0,0] and ending at [R-1,C-1].

///

/// The score of a path is the minimum value in that path. For example, the

/// value of the path 8 → 4 → 5 → 9 is 4.

///

/// A path moves some number of times from one visited cell to any neighbouring

/// unvisited cell in one of the 4 cardinal directions (north, east, west,

/// south).

///

///

///

/// Example 1:

/// Input: [[5,4,5],[1,2,6],[7,4,6]]

/// Output: 4

/// Explanation:

/// The path with the maximum score is highlighted in yellow.

///

/// Example 2:

/// Input: [[2,2,1,2,2,2],[1,2,2,2,1,2]]

/// Output: 2

///

/// Example 3:

///

/// Input: [[3,4,6,3,4],[0,2,1,1,7],[8,8,3,2,7],[3,2,4,9,8],[4,1,2,0,0],

/// [4,6,5,4,3]]

/// Output: 3

///

/// Note:

/// 1. 1 <= R, C <= 100

/// 2. 0 <= A[i][j] <= 10^9

/// </summary>

int LeetCodeGraph::maximumMinimumPath(vector<vector<int>>& A)

{

priority\_queue<vector<int>> pq;

pq.push({ A[0][0], 0, 0 });

vector<vector<int>> directions = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };

int R = A.size();

int C = A[0].size();

vector<vector<int>> result(R, vector<int>(C, -1));

while (!pq.empty())

{

vector<int> pos = pq.top();

pq.pop();

if (pos[1] == R - 1 && pos[2] == C - 1)

{

return pos[0];

}

for (size\_t i = 0; i < directions.size(); i++)

{

int r = pos[1];

int c = pos[2];

r += directions[i][0];

c += directions[i][1];

if (r < 0 || r >= R || c < 0 || c >= C)

{

continue;

}

int v = min(A[r][c], pos[0]);

if (v > result[r][c])

{

result[r][c] = v;

pq.push({ v, r, c });

}

}

}

return -1;

}

## 1514. Path with Maximum Probability

Medium

You are given an undirected weighted graph of n nodes (0-indexed), represented by an edge list where edges[i] = [a, b] is an undirected edge connecting the nodes a and b with a probability of success of traversing that edge succProb[i].

Given two nodes start and end, find the path with the maximum probability of success to go from start to end and return its success probability.

If there is no path from start to end, **return 0**. Your answer will be accepted if it differs from the correct answer by at most **1e-5**.

**Example 1:**

**A picture containing text, clock, watch

Description automatically generated**

**Input:** n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.2], start = 0, end = 2

**Output:** 0.25000

**Explanation:** There are two paths from start to end, one having a probability of success = 0.2 and the other has 0.5 \* 0.5 = 0.25.

**Example 2:**

**Diagram

Description automatically generated**

**Input:** n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.3], start = 0, end = 2

**Output:** 0.30000

**Example 3:**

**Diagram

Description automatically generated**

**Input:** n = 3, edges = [[0,1]], succProb = [0.5], start = 0, end = 2

**Output:** 0.00000

**Explanation:** There is no path between 0 and 2.

**Constraints:**

* 2 <= n <= 10^4
* 0 <= start, end < n
* start != end
* 0 <= a, b < n
* a != b
* 0 <= succProb.length == edges.length <= 2\*10^4
* 0 <= succProb[i] <= 1
* There is at most one edge between every two nodes.

### Analysis:

Follow the maximum probability as next stop, use Dijkstra algorithm.

/// <summary>

/// Leet code #1514. Path with Maximum Probability

///

/// Medium

///

/// You are given an undirected weighted graph of n nodes (0-indexed),

/// represented by an edge list where edges[i] = [a, b] is an undirected

/// edge connecting the nodes a and b with a probability of success of

/// traversing that edge succProb[i].

///

/// Given two nodes start and end, find the path with the maximum

/// probability of success to go from start to end and return its success

/// probability.

///

/// If there is no path from start to end, return 0. Your answer will

/// be accepted if it differs from the correct answer by at most 1e-5.

///

///

/// Example 1:

/// Input: n = 3, edges = [[0,1],[1,2],[0,2]],

/// succProb = [0.5,0.5,0.2], start = 0, end = 2

/// Output: 0.25000

/// Explanation: There are two paths from start to end, one having a

/// probability of success = 0.2 and the other has 0.5 \* 0.5 = 0.25.

///

/// Example 2:

/// Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.3],

/// start = 0, end = 2

/// Output: 0.30000

///

/// Example 3:

/// Input: n = 3, edges = [[0,1]], succProb = [0.5], start = 0, end = 2

/// Output: 0.00000

/// Explanation: There is no path between 0 and 2.

///

/// Constraints:

/// 1. 2 <= n <= 10^4

/// 2. 0 <= start, end < n

/// 3. start != end

/// 4. 0 <= a, b < n

/// 5. a != b

/// 6. 0 <= succProb.length == edges.length <= 2\*10^4

/// 7. 0 <= succProb[i] <= 1

/// 8. There is at most one edge between every two nodes.

/// </summary>

double LeetCodeGraph::maxProbability(int n, vector<vector<int>>& edges, vector<double>& succProb, int start, int end)

{

vector<vector<pair<int, double>>> next(n);

for (size\_t i = 0; i < edges.size(); i++)

{

next[edges[i][0]].push\_back(make\_pair(edges[i][1], succProb[i]));

next[edges[i][1]].push\_back(make\_pair(edges[i][0], succProb[i]));

}

vector<double> probability(n);

priority\_queue<pair<double, int>> pq;

probability[start] = 1;

pq.push(make\_pair(1, start));

while (!pq.empty())

{

pair<double, int> pr = pq.top();

pq.pop();

if (pr.second == end) break;

if (pr.first < probability[pr.second]) continue;

for (size\_t i = 0; i < next[pr.second].size(); i++)

{

pair<int, double> node = next[pr.second][i];

if (pr.first \* node.second <= probability[node.first]) continue;

probability[node.first] = pr.first \* node.second;

pq.push(make\_pair(pr.first \* node.second, node.first));

}

}

return probability[end];

}

## 1631. Path With Minimum Effort

Medium

You are a hiker preparing for an upcoming hike. You are given heights, a 2D array of size rows x columns, where heights[row][col] represents the height of cell (row, col). You are situated in the top-left cell, (0, 0), and you hope to travel to the bottom-right cell, (rows-1, columns-1) (i.e., **0-indexed**). You can move **up**, **down**, **left**, or **right**, and you wish to find a route that requires the minimum **effort**.

A route's **effort** is the **maximum absolute difference**in heights between two consecutive cells of the route.

Return *the minimum****effort****required to travel from the top-left cell to the bottom-right cell.*

**Example 1:**

A picture containing calendar

Description automatically generated

**Input:** heights = [[1,2,2],[3,8,2],[5,3,5]]

**Output:** 2

**Explanation:** The route of [1,3,5,3,5] has a maximum absolute difference of 2 in consecutive cells.

This is better than the route of [1,2,2,2,5], where the maximum absolute difference is 3.

**Example 2:**

A picture containing calendar

Description automatically generated

**Input:** heights = [[1,2,3],[3,8,4],[5,3,5]]

**Output:** 1

**Explanation:** The route of [1,2,3,4,5] has a maximum absolute difference of 1 in consecutive cells, which is better than route [1,3,5,3,5].

**Example 3:**

Table

Description automatically generated

**Input:** heights = [[1,2,1,1,1],[1,2,1,2,1],[1,2,1,2,1],[1,2,1,2,1],[1,1,1,2,1]]

**Output:** 0

**Explanation:** This route does not require any effort.

**Constraints:**

* rows == heights.length
* columns == heights[i].length
* 1 <= rows, columns <= 100
* 1 <= heights[i][j] <= 106

### Analysis:

Follow the minimum effort as next stop, use Dijkstra algorithm.

/// <summary>

/// Leet code #1631. Path With Minimum Effort

///

/// Medium

///

/// You are a hiker preparing for an upcoming hike. You are given heights,

/// a 2D array of size rows x columns, where heights[row][col] represents

/// the height of cell (row, col). You are situated in the top-left cell,

/// (0, 0), and you hope to travel to the bottom-right cell,

/// (rows-1, columns-1) (i.e., 0-indexed). You can move up, down, left, or

/// right, and you wish to find a route that requires the minimum effort.

///

/// A route's effort is the maximum absolute difference in heights between

/// two consecutive cells of the route.

///

/// Return the minimum effort required to travel from the top-left cell to

/// the bottom-right cell.

///

/// Example 1:

/// Input: heights = [[1,2,2],[3,8,2],[5,3,5]]

/// Output: 2

/// Explanation: The route of [1,3,5,3,5] has a maximum absolute difference

/// of 2 in consecutive cells.

/// This is better than the route of [1,2,2,2,5], where the maximum absolute

/// difference is 3.

///

/// Example 2:

/// Input: heights = [[1,2,3],[3,8,4],[5,3,5]]

/// Output: 1

/// Explanation: The route of [1,2,3,4,5] has a maximum absolute difference

/// of 1 in consecutive cells, which is better than route [1,3,5,3,5].

///

/// Example 3:

/// Input: heights = [[1,2,1,1,1],[1,2,1,2,1],[1,2,1,2,1],[1,2,1,2,1],

/// [1,1,1,2,1]]

/// Output: 0

/// Explanation: This route does not require any effort.

///

/// Constraints:

/// 1. rows == heights.length

/// 2. columns == heights[i].length

/// 3. 1 <= rows, columns <= 100

/// 4. 1 <= heights[i][j] <= 10^6

/// </summary>

int LeetCodeGraph::minimumEffortPath(vector<vector<int>>& heights)

{

int n = heights.size();

int m = heights[0].size();

vector<vector<int>> effort(n, vector<int>(m, INT\_MAX));

priority\_queue<vector<int>> pq;

effort[0][0] = 0;

pq.push({ 0, 0, 0 });

vector<vector<int>> directions = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };

while (!pq.empty())

{

vector<int> node = pq.top();

pq.pop();

int r = node[1];

int c = node[2];

if (r == n - 1 && c == m - 1) break;

if (0 - node[0] > effort[r][c]) continue;

for (size\_t d = 0; d < directions.size(); d++)

{

int next\_r = r + directions[d][0];

int next\_c = c + directions[d][1];

if (next\_r < 0 || next\_r >= n || next\_c < 0 || next\_c >= m)

{

continue;

}

int gap = abs(heights[next\_r][next\_c] - heights[r][c]);

gap = max(gap, effort[r][c]);

if (gap < effort[next\_r][next\_c])

{

effort[next\_r][next\_c] = gap;

pq.push({ -gap, next\_r, next\_c});

}

}

}

return effort[n - 1][m - 1];

}

## 1976. Number of Ways to Arrive at Destination

Medium

You are in a city that consists of n intersections numbered from 0 to n - 1 with **bi-directional** roads between some intersections. The inputs are generated such that you can reach any intersection from any other intersection and that there is at most one road between any two intersections.

You are given an integer n and a 2D integer array roads where roads[i] = [ui, vi, timei] means that there is a road between intersections ui and vi that takes timei minutes to travel. You want to know in how many ways you can travel from intersection 0 to intersection n - 1 in the **shortest amount of time**.

Return *the****number of ways****you can arrive at your destination in the****shortest amount of time***. Since the answer may be large, return it **modulo** 109 + 7.

**Example 1:**

A screenshot of a computer screen

Description automatically generated with low confidence

**Input:** n = 7, roads = [[0,6,7],[0,1,2],[1,2,3],[1,3,3],[6,3,3],[3,5,1],[6,5,1],[2,5,1],[0,4,5],[4,6,2]]

**Output:** 4

**Explanation:** The shortest amount of time it takes to go from intersection 0 to intersection 6 is 7 minutes.

The four ways to get there in 7 minutes are:

- 0 ➝ 6

- 0 ➝ 4 ➝ 6

- 0 ➝ 1 ➝ 2 ➝ 5 ➝ 6

- 0 ➝ 1 ➝ 3 ➝ 5 ➝ 6

**Example 2:**

**Input:** n = 2, roads = [[1,0,10]]

**Output:** 1

**Explanation:** There is only one way to go from intersection 0 to intersection 1, and it takes 10 minutes.

**Constraints:**

* 1 <= n <= 200
* n - 1 <= roads.length <= n \* (n - 1) / 2
* roads[i].length == 3
* 0 <= ui, vi <= n - 1
* 1 <= timei <= 109
* ui!= vi
* There is at most one road connecting any two intersections.
* You can reach any intersection from any other intersection.

### Analysis:

Because you want shortest path, use Dijkstra

/// <summary>

/// Leet Code 1976. Number of Ways to Arrive at Destination

///

/// Medium

///

/// You are in a city that consists of n intersections numbered from 0 to

/// n - 1 with bi-directional roads between some intersections. The inputs

/// are generated such that you can reach any intersection from any other

/// intersection and that there is at most one road between any two

/// intersections.

/// You are given an integer n and a 2D integer array roads where roads[i]

/// = [ui, vi, timei] means that there is a road between intersections ui

/// and vi that takes timei minutes to travel. You want to know in how

/// many ways you can travel from intersection 0 to intersection n - 1 in

/// the shortest amount of time.

/// Return the number of ways you can arrive at your destination in the

/// shortest amount of time. Since the answer may be large, return it

/// modulo 10^9 + 7.

///

/// Example 1:

/// Input: n = 7, roads = [[0,6,7],[0,1,2],[1,2,3],[1,3,3],[6,3,3],

/// [3,5,1],[6,5,1],[2,5,1],[0,4,5],[4,6,2]]

/// Output: 4

/// Explanation: The shortest amount of time it takes to go from

/// intersection 0 to intersection 6 is 7 minutes.

/// The four ways to get there in 7 minutes are:

/// - 0 -> 6

/// - 0 -> 4 -> 6

/// - 0 -> 1 -> 2 -> 5 -> 6

/// - 0 -> 1 -> 3 -> 5 -> 6

///

/// Example 2:

/// Input: n = 2, roads = [[1,0,10]]

/// Output: 1

/// Explanation: There is only one way to go from intersection 0 to

/// intersection 1, and it takes 10 minutes.

///

/// Constraints:

/// 1. 1 <= n <= 200

/// 2. n - 1 <= roads.length <= n \* (n - 1) / 2

/// 3. roads[i].length == 3

/// 4. 0 <= ui, vi <= n - 1

/// 5. 1 <= timei <= 10^9

/// 6. ui != vi

/// 7. There is at most one road connecting any two intersections.

/// 8. You can reach any intersection from any other intersection.

/// </summary>

int LeetCodeGraph::countPaths(int n, vector<vector<int>>& roads)

{

vector<pair<long long, int>> dist(n, make\_pair(LLONG\_MAX, 0));

vector<vector<pair<int, int>>> neighbors(n);

for (size\_t i = 0; i < roads.size(); i++)

{

neighbors[roads[i][0]].push\_back(make\_pair(roads[i][1], roads[i][2]));

neighbors[roads[i][1]].push\_back(make\_pair(roads[i][0], roads[i][2]));

}

priority\_queue<pair<long long, int>> pq;

pq.push(make\_pair(0, 0));

dist[0] = make\_pair(0, 1);

int M = 1000000007;

while (!pq.empty())

{

pair<long long, int> pos = pq.top();

pq.pop();

int node = pos.second;

for (size\_t i = 0; i < neighbors[node].size(); i++)

{

int next = neighbors[node][i].first;

long long d = (long long)(0 - pos.first) + (long long)neighbors[node][i].second;

if (dist[next].first > d)

{

dist[next].first = d;

dist[next].second = dist[node].second;

pq.push(make\_pair(0 - d, next));

}

else if (dist[next].first == d)

{

dist[next].second = (dist[next].second + dist[node].second) % M;

}

}

}

return dist[n - 1].second;

}

# Advance Problems

## 778. Swim in Rising Water

Hard

You are given an n x n integer matrix grid where each value grid[i][j] represents the elevation at that point (i, j).

The rain starts to fall. At time t, the depth of the water everywhere is t. You can swim from a square to another 4-directionally adjacent square if and only if the elevation of both squares individually are at most t. You can swim infinite distances in zero time. Of course, you must stay within the boundaries of the grid during your swim.

Return *the least time until you can reach the bottom right square*(n - 1, n - 1)*if you start at the top left square*(0, 0).

**Example 1:**

A picture containing text, clock

Description automatically generated

**Input:** grid = [[0,2],[1,3]]

**Output:** 3

Explanation:

At time 0, you are in grid location (0, 0).

You cannot go anywhere else because 4-directionally adjacent neighbors have a higher elevation than t = 0.

You cannot reach point (1, 1) until time 3.

When the depth of water is 3, we can swim anywhere inside the grid.

**Example 2:**

Text

Description automatically generated with low confidence

**Input:** grid = [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,7,6]]

**Output:** 16

**Explanation:** The final route is shown.

We need to wait until time 16 so that (0, 0) and (4, 4) are connected.

**Constraints:**

* n == grid.length
* n == grid[i].length
* 1 <= n <= 50
* 0 <= grid[i][j] < n2
* Each value grid[i][j] is **unique**.

### Analysis:

We swim to the position based on height and use Dijkstra to find out what position is available next. Remember we never swim from a higher position to a lower position, we need to fill the lower position if we are already in a higher position.

/// <summary>

/// Leet code #778. Swim in Rising Water

///

/// On an N x N grid, each square grid[i][j] represents the elevation at

/// that point (i,j).

///

/// Now rain starts to fall. At time t, the depth of the water everywhere

/// is t. You can swim from a square to another 4-directionally adjacent

/// square if and only if the elevation of both squares individually are

/// at most t. You can swim infinite distance in zero time. Of course,

/// you must stay within the boundaries of the grid during your swim.

///

/// You start at the top left square (0, 0). What is the least time

/// until you can reach the bottom right square (N-1, N-1)?

///

/// Example 1:

///

/// Input: [[0,2],[1,3]]

/// Output: 3

/// Explanation:

/// At time 0, you are in grid location (0, 0).

/// You cannot go anywhere else because 4-directionally adjacent neighbors

/// have a higher elevation than t = 0.

///

/// You cannot reach point (1, 1) until time 3.

/// When the depth of water is 3, we can swim anywhere inside the grid.

/// Example 2:

///

/// Input: [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],

/// [10,9,8,7,6]]

/// Output: 16

/// Explanation:

/// 0 1 2 3 4

/// 24 23 22 21 5

/// 12 13 14 15 16

/// 11 17 18 19 20

/// 10 9 8 7 6

///

/// The final route is marked in bold.

/// We need to wait until time 16 so that (0, 0) and (4, 4) are connected.

/// Note:

///

/// 1. 2 <= N <= 50.

/// 2. grid[i][j] is a permutation of [0, ..., N\*N - 1].

/// </summary>

int LeetCodeGraph::swimInWater(vector<vector<int>>& grid)

{

vector<vector<int>> direction = { { -1, 0 },{ 1, 0 },{ 0, -1 },{ 0, 1 } };

vector<vector<int>> time(grid.size(), vector<int>(grid[0].size(), INT\_MAX));

priority\_queue<vector<int>> search;

// vector[0] = time, vector[1] = row, vector[2] = col

vector<int> start = { -grid[0][0], 0, 0 };

time[0][0] = grid[0][0];

search.push(start);

while (!search.empty())

{

vector<int> current = search.top();

current[0] = 0 - current[0];

search.pop();

// we already visit this node

if (current[0] > time[current[1]][current[2]]) continue;

for (size\_t i = 0; i < direction.size(); i++)

{

vector<int> next = current;

next[1] += direction[i][0];

next[2] += direction[i][1];

// out of boundary

if ((next[1] < 0) || (next[1] == grid.size()) ||

(next[2] < 0) || (next[2] == grid[0].size()))

{

continue;

}

next[0] = max(current[0], grid[next[1]][next[2]]);

if (next[0] < time[next[1]][next[2]])

{

time[next[1]][next[2]] = next[0];

next[0] = 0 - next[0];

search.push(next);

}

}

}

if (time[grid.size() - 1][grid[0].size() - 1] == INT\_MAX)

{

return -1;

}

else

{

return time[grid.size() - 1][grid[0].size() - 1];

}

}

## 1168. Optimize Water Distribution in a Village

Hard

There are n houses in a village. We want to supply water for all the houses by building wells and laying pipes.

For each house i, we can either build a well inside it directly with cost wells[i - 1] (note the -1 due to **0-indexing**), or pipe in water from another well to it. The costs to lay pipes between houses are given by the array pipes, where each pipes[j] = [house1j, house2j, costj] represents the cost to connect house1j and house2j together using a pipe. Connections are bidirectional.

Return *the minimum total cost to supply water to all houses*.

**Example 1:**

Diagram

Description automatically generated

**Input:** n = 3, wells = [1,2,2], pipes = [[1,2,1],[2,3,1]]

**Output:** 3

**Explanation:**

The image shows the costs of connecting houses using pipes.

The best strategy is to build a well in the first house with cost 1 and connect the other houses to it with cost 2 so the total cost is 3.

**Example 2:**

**Input:** n = 2, wells = [1,1], pipes = [[1,2,1]]

**Output:** 2

**Constraints:**

* 2 <= n <= 104
* wells.length == n
* 0 <= wells[i] <= 105
* 1 <= pipes.length <= 104
* pipes[j].length == 3
* 1 <= house1j, house2j <= n
* 0 <= costj <= 105
* house1j != house2j

### Analysis:

Start from lowest cost of well, and use Dijkstra to find the next cheapest village, do not visit the same village twice.

/// <summary>

/// Leet code #1168. Optimize Water Distribution in a Village

///

/// There are n houses in a village. We want to supply water for all

/// the houses by building wells and laying pipes.

/// For each house i, we can either build a well inside it directly with

/// cost wells[i], or pipe in water from another well to it. The costs to

/// lay pipes between houses are given by the array pipes, where each

/// pipes[i] = [house1, house2, cost] represents the cost to connect house1

/// and house2 together using a pipe. Connections are bidirectional.

/// Find the minimum total cost to supply water to all houses.

///

/// Example 1:

///

/// Input: n = 3, wells = [1,2,2], pipes = [[1,2,1],[2,3,1]]

/// Output: 3

/// Explanation:

/// The image shows the costs of connecting houses using pipes.

/// The best strategy is to build a well in the first house with cost 1 and

/// connect the other houses to it with cost 2 so the total cost is 3.

///

/// Constraints:

/// 1. 1 <= n <= 10000

/// 2. wells.length == n

/// 3. 0 <= wells[i] <= 10^5

/// 4. 1 <= pipes.length <= 10000

/// 5. 1 <= pipes[i][0], pipes[i][1] <= n

/// 6. 0 <= pipes[i][2] <= 10^5

/// 7. pipes[i][0] != pipes[i][1]

/// </summary>

int LeetCodeGraph::minCostToSupplyWater(int n, vector<int>& wells, vector<vector<int>>& pipes)

{

vector<vector<pair<int, int>>> house\_pipe(n);

priority\_queue<pair<int, int>> heap;

vector<int> cost(n, INT\_MAX);

for (size\_t i = 0; i < wells.size(); i++)

{

heap.push(make\_pair(-wells[i], i));

}

for (size\_t i = 0; i < pipes.size(); i++)

{

house\_pipe[pipes[i][0] - 1].push\_back(make\_pair(pipes[i][1] - 1, pipes[i][2]));

house\_pipe[pipes[i][1] - 1].push\_back(make\_pair(pipes[i][0] - 1, pipes[i][2]));

}

while (!heap.empty())

{

pair<int, int> well = heap.top();

heap.pop();

int village = well.second;

if (cost[village] == INT\_MAX)

{

cost[village] = -well.first;

for (size\_t i = 0; i < house\_pipe[village].size(); i++)

{

pair<int, int> pipe = house\_pipe[village][i];

if (pipe.second < wells[pipe.first] &&

cost[pipe.first] == INT\_MAX)

{

wells[pipe.first] = pipe.second;

heap.push(make\_pair(-pipe.second, pipe.first));

}

}

}

}

int result = 0;

for (size\_t i = 0; i < cost.size(); i++)

{

result += cost[i];

}

return result;

}

## 1368. Minimum Cost to Make at Least One Valid Path in a Grid

Hard

Given an m x n grid. Each cell of the grid has a sign pointing to the next cell you should visit if you are currently in this cell. The sign of grid[i][j] can be:

* 1 which means go to the cell to the right. (i.e go from grid[i][j] to grid[i][j + 1])
* 2 which means go to the cell to the left. (i.e go from grid[i][j] to grid[i][j - 1])
* 3 which means go to the lower cell. (i.e go from grid[i][j] to grid[i + 1][j])
* 4 which means go to the upper cell. (i.e go from grid[i][j] to grid[i - 1][j])

Notice that there could be some signs on the cells of the grid that point outside the grid.

You will initially start at the upper left cell (0, 0). A valid path in the grid is a path that starts from the upper left cell (0, 0) and ends at the bottom-right cell (m - 1, n - 1) following the signs on the grid. The valid path does not have to be the shortest.

You can modify the sign on a cell with cost = 1. You can modify the sign on a cell **one time only**.

Return *the minimum cost to make the grid have at least one valid path*.

**Example 1:**

Shape

Description automatically generated

**Input:** grid = [[1,1,1,1],[2,2,2,2],[1,1,1,1],[2,2,2,2]]

**Output:** 3

**Explanation:** You will start at point (0, 0).

The path to (3, 3) is as follows. (0, 0) --> (0, 1) --> (0, 2) --> (0, 3) change the arrow to down with cost = 1 --> (1, 3) --> (1, 2) --> (1, 1) --> (1, 0) change the arrow to down with cost = 1 --> (2, 0) --> (2, 1) --> (2, 2) --> (2, 3) change the arrow to down with cost = 1 --> (3, 3)

The total cost = 3.

**Example 2:**

Shape

Description automatically generated

**Input:** grid = [[1,1,3],[3,2,2],[1,1,4]]

**Output:** 0

**Explanation:** You can follow the path from (0, 0) to (2, 2).

**Example 3:**

Shape

Description automatically generated with medium confidence

**Input:** grid = [[1,2],[4,3]]

**Output:** 1

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 100
* 1 <= grid[i][j] <= 4

### Analysis:

Follow the valid path and if there are multiple available next stop, use Dijkstra to find the cheapest one.

/// <summary>

/// Leet code #1368. Minimum Cost to Make at Least One Valid Path in a Grid

///

/// Hard

///

/// Given a m x n grid. Each cell of the grid has a sign pointing to the

/// next cell you should visit if you are currently in this cell. The sign

/// of grid[i][j] can be:

/// 1 which means go to the cell to the right. (i.e go from grid[i][j]

/// to grid[i][j + 1])

/// 2 which means go to the cell to the left. (i.e go from grid[i][j] to

/// grid[i][j - 1])

/// 3 which means go to the lower cell. (i.e go from grid[i][j] to

/// grid[i + 1][j])

/// 4 which means go to the upper cell. (i.e go from grid[i][j] to

/// grid[i - 1][j])

/// Notice that there could be some invalid signs on the cells of the

/// grid which points outside the grid.

///

/// You will initially start at the upper left cell (0,0). A valid path

/// in the grid is a path which starts from the upper left cell (0,0) and

/// ends at the bottom-right cell (m - 1, n - 1) following the signs on

/// the grid. The valid path doesn't have to be the shortest.

///

/// You can modify the sign on a cell with cost = 1. You can modify the

/// sign on a cell one time only.

///

/// Return the minimum cost to make the grid have at least one valid path.

///

/// Example 1:

///

/// Input: grid = [[1,1,1,1],[2,2,2,2],[1,1,1,1],[2,2,2,2]]

/// Output: 3

/// Explanation: You will start at point (0, 0).

/// The path to (3, 3) is as follows. (0, 0) --> (0, 1) --> (0, 2) -->

/// (0, 3) change the arrow to down with cost = 1 --> (1, 3) -->

/// (1, 2) --> (1, 1) --> (1, 0) change the arrow to down with cost =

/// 1 --> (2, 0) --> (2, 1) --> (2, 2) --> (2, 3) change the arrow to

/// down with cost = 1 --> (3, 3)

/// The total cost = 3.

///

/// Example 2:

/// Input: grid = [[1,1,3],[3,2,2],[1,1,4]]

/// Output: 0

/// Explanation: You can follow the path from (0, 0) to (2, 2).

///

/// Example 3:

/// Input: grid = [[1,2],[4,3]]

/// Output: 1

///

/// Example 4:

/// Input: grid = [[2,2,2],[2,2,2]]

/// Output: 3

///

/// Example 5:

/// Input: grid = [[4]]

/// Output: 0

///

/// Constraints:

/// 1. m == grid.length

/// 2. n == grid[i].length

/// 3. 1 <= m, n <= 100

/// </summary>

int LeetCodeGraph::minCost(vector<vector<int>>& grid)

{

int n = grid.size();

int m = grid[0].size();

vector<pair<int, int>> directions =

{

{0, 1}, {0, -1}, {1, 0}, {-1, 0}

};

vector<vector<int>> cost(n, vector<int>(m, INT\_MIN));

priority\_queue<vector<int>> pq;

pq.push({ 0, 0, 0 });

cost[0][0] = 0;

int result = -1;

while (!pq.empty())

{

vector<int> pos = pq.top();

pq.pop();

if (pos[1] == n - 1 && pos[2] == m - 1)

{

result = 0 - pos[0];

break;

}

vector<int> next(3);

for (int i = 0; i < 4; i++)

{

if (i == grid[pos[1]][pos[2]] - 1)

{

next[0] = pos[0];

}

else

{

next[0] = pos[0] - 1;

}

next[1] = pos[1] + directions[i].first;

next[2] = pos[2] + directions[i].second;

if (next[1] < 0 || next[1] >= n || next[2] < 0 || next[2] >= m)

{

continue;

}

if (next[0] <= cost[next[1]][next[2]]) continue;

cost[next[1]][next[2]] = next[0];

pq.push(next);

}

}

return result;

}

## 1928. Minimum Cost to Reach Destination in Time

Hard

There is a country of n cities numbered from 0 to n - 1 where **all the cities are connected** by bi-directional roads. The roads are represented as a 2D integer array edges where edges[i] = [xi, yi, timei] denotes a road between cities xi and yi that takes timei minutes to travel. There may be multiple roads of differing travel times connecting the same two cities, but no road connects a city to itself.

Each time you pass through a city, you must pay a passing fee. This is represented as a **0-indexed** integer array passingFees of length n where passingFees[j] is the amount of dollars you must pay when you pass through city j.

In the beginning, you are at city 0 and want to reach city n - 1 in maxTime**minutes or less**. The **cost** of your journey is the **summation of passing fees** for each city that you passed through at some moment of your journey (**including** the source and destination cities).

Given maxTime, edges, and passingFees, return *the****minimum cost****to complete your journey, or*-1*if you cannot complete it within*maxTime*minutes*.

**Example 1:**

Diagram

Description automatically generated

**Input:** maxTime = 30, edges = [[0,1,10],[1,2,10],[2,5,10],[0,3,1],[3,4,10],[4,5,15]], passingFees = [5,1,2,20,20,3]

**Output:** 11

**Explanation:** The path to take is 0 -> 1 -> 2 -> 5, which takes 30 minutes and has $11 worth of passing fees.

**Example 2:**

**Diagram

Description automatically generated**

**Input:** maxTime = 29, edges = [[0,1,10],[1,2,10],[2,5,10],[0,3,1],[3,4,10],[4,5,15]], passingFees = [5,1,2,20,20,3]

**Output:** 48

**Explanation:** The path to take is 0 -> 3 -> 4 -> 5, which takes 26 minutes and has $48 worth of passing fees.

You cannot take path 0 -> 1 -> 2 -> 5 since it would take too long.

**Example 3:**

**Input:** maxTime = 25, edges = [[0,1,10],[1,2,10],[2,5,10],[0,3,1],[3,4,10],[4,5,15]], passingFees = [5,1,2,20,20,3]

**Output:** -1

**Explanation:** There is no way to reach city 5 from city 0 within 25 minutes.

**Constraints:**

* 1 <= maxTime <= 1000
* n == passingFees.length
* 2 <= n <= 1000
* n - 1 <= edges.length <= 1000
* 0 <= xi, yi <= n - 1
* 1 <= timei <= 1000
* 1 <= passingFees[j] <= 1000
* The graph may contain multiple edges between two nodes.
* The graph does not contain self loops.

### Analysis:

Use Dijkstra to track cost in priority queue, but also track the time, when time exceed, discard it.

/// <summary>

/// Leet code 1928. Minimum Cost to Reach Destination in Time

///

/// Hard

///

/// There is a country of n cities numbered from 0 to n - 1 where all the

/// cities are connected by bi-directional roads. The roads are

/// represented as a 2D integer array edges where edges[i] = [xi, yi,

/// timei] denotes a road between cities xi and yi that takes timei

/// minutes to travel. There may be multiple roads of differing travel

/// times connecting the same two cities, but no road connects a city

/// to itself.

/// Each time you pass through a city, you must pay a passing fee.

/// This is represented as a 0-indexed integer array passingFees of

/// length n where passingFees[j] is the amount of dollars you must

/// pay when you pass through city j.

///

/// In the beginning, you are at city 0 and want to reach city n - 1

/// in maxTime minutes or less. The cost of your journey is the

/// summation of passing fees for each city that you passed through

/// at some moment of your journey (including the source and

/// destination cities).

///

/// Given maxTime, edges, and passingFees, return the minimum cost

/// to complete your journey, or -1 if you cannot complete it

/// within maxTime minutes.

///

/// Example 1:

/// Input: maxTime = 30, edges = [[0,1,10],[1,2,10],[2,5,10],[0,3,1],

/// [3,4,10],[4,5,15]], passingFees = [5,1,2,20,20,3]

/// Output: 11

/// Explanation: The path to take is 0 -> 1 -> 2 -> 5, which takes 30

/// minutes and has $11 worth of passing fees.

///

/// Example 2:

/// Input: maxTime = 29, edges = [[0,1,10],[1,2,10],[2,5,10],[0,3,1],

/// [3,4,10],[4,5,15]], passingFees = [5,1,2,20,20,3]

/// Output: 48

/// Explanation: The path to take is 0 -> 3 -> 4 -> 5, which takes

/// 26 minutes and has $48 worth of passing fees.

/// You cannot take path 0 -> 1 -> 2 -> 5 since it would take too long.

///

/// Example 3:

/// Input: maxTime = 25, edges = [[0,1,10],[1,2,10],[2,5,10],[0,3,1],

/// [3,4,10],[4,5,15]], passingFees = [5,1,2,20,20,3]

/// Output: -1

/// Explanation: There is no way to reach city 5 from city 0 within

/// 25 minutes.

///

/// Constraints:

/// 1. 1 <= maxTime <= 1000

/// 2. n == passingFees.length

/// 3. 2 <= n <= 1000

/// 4. n - 1 <= edges.length <= 1000

/// 5. 0 <= xi, yi <= n - 1

/// 6. 1 <= timei <= 1000

/// 7. 1 <= passingFees[j] <= 1000

/// 8. The graph may contain multiple edges between two nodes.

/// 9. The graph does not contain self loops.

/// </summary>

int LeetCodeGraph::minCost(int maxTime, vector<vector<int>>& edges, vector<int>& passingFees)

{

int n = passingFees.size();

vector <vector<pair<int, int>>> adjacent(n, vector<pair<int, int>>());

vector<int> city\_cost(n, { INT\_MAX });

vector<int> city\_time(n, { INT\_MAX });

for (size\_t i = 0; i < edges.size(); i++)

{

adjacent[edges[i][0]].push\_back({ edges[i][1], edges[i][2] });

adjacent[edges[i][1]].push\_back({ edges[i][0], edges[i][2] });

}

set<vector<int>> pq;

pq.insert({ passingFees[0], 0, 0 });

while (!pq.empty())

{

vector<int> destination = \*pq.begin();

pq.erase(pq.begin());

int cost = destination[0];

int time = destination[1];

int city = destination[2];

if (city == n - 1) return cost;

for (size\_t i = 0; i < adjacent[city].size(); i++)

{

int next\_city = adjacent[city][i].first;

int next\_cost = cost + passingFees[next\_city];

int next\_time = time + adjacent[city][i].second;

if (next\_time > maxTime) continue;

if (city\_cost[next\_city] <= next\_cost && city\_time[next\_city] <= next\_time)

{

continue;

}

city\_cost[next\_city] = min(city\_cost[next\_city], next\_cost);

city\_time[next\_city] = min(city\_time[next\_city], next\_time);

pq.insert({ next\_cost, next\_time, next\_city });

}

}

return -1;

}

## 2092. Find All People With Secret

Hard

You are given an integer n indicating there are n people numbered from 0 to n - 1. You are also given a **0-indexed** 2D integer array meetings where meetings[i] = [xi, yi, timei] indicates that person xi and person yi have a meeting at timei. A person may attend **multiple meetings** at the same time. Finally, you are given an integer firstPerson.

Person 0 has a **secret** and initially shares the secret with a person firstPerson at time 0. This secret is then shared every time a meeting takes place with a person that has the secret. More formally, for every meeting, if a person xi has the secret at timei, then they will share the secret with person yi, and vice versa.

The secrets are shared **instantaneously**. That is, a person may receive the secret and share it with people in other meetings within the same time frame.

Return *a list of all the people that have the secret after all the meetings have taken place.*You may return the answer in **any order**.

**Example 1:**

**Input:** n = 6, meetings = [[1,2,5],[2,3,8],[1,5,10]], firstPerson = 1

**Output:** [0,1,2,3,5]

**Explanation:**

At time 0, person 0 shares the secret with person 1.

At time 5, person 1 shares the secret with person 2.

At time 8, person 2 shares the secret with person 3.

At time 10, person 1 shares the secret with person 5.​​​​

Thus, people 0, 1, 2, 3, and 5 know the secret after all the meetings.

**Example 2:**

**Input:** n = 4, meetings = [[3,1,3],[1,2,2],[0,3,3]], firstPerson = 3

**Output:** [0,1,3]

**Explanation:**

At time 0, person 0 shares the secret with person 3.

At time 2, neither person 1 nor person 2 know the secret.

At time 3, person 3 shares the secret with person 0 and person 1.

Thus, people 0, 1, and 3 know the secret after all the meetings.

**Example 3:**

**Input:** n = 5, meetings = [[3,4,2],[1,2,1],[2,3,1]], firstPerson = 1

**Output:** [0,1,2,3,4]

**Explanation:**

At time 0, person 0 shares the secret with person 1.

At time 1, person 1 shares the secret with person 2, and person 2 shares the secret with person 3.

Note that person 2 can share the secret at the same time as receiving it.

At time 2, person 3 shares the secret with person 4.

Thus, people 0, 1, 2, 3, and 4 know the secret after all the meetings.

**Constraints:**

* 2 <= n <= 105
* 1 <= meetings.length <= 105
* meetings[i].length == 3
* 0 <= xi, yi<= n - 1
* xi != yi
* 1 <= timei <= 105
* 1 <= firstPerson <= n - 1

### Analysis:

Use Dijkstra to track earliest time for each person get the secret.

/// <summary>

/// Leet Code 2092. Find All People With Secret

///

/// Hard

///

/// You are given an integer n indicating there are n people numbered

/// from 0 to n - 1. You are also given a 0-indexed 2D integer array

/// meetings where meetings[i] = [xi, yi, timei] indicates that person

/// xi and person yi have a meeting at timei. A person may attend

/// multiple meetings at the same time. Finally, you are given an integer

/// firstPerson.

///

/// Person 0 has a secret and initially shares the secret with a person

/// firstPerson at time 0. This secret is then shared every time a meeting

/// takes place with a person that has the secret. More formally, for

/// every meeting, if a person xi has the secret at timei, then they will

/// share the secret with person yi, and vice versa.

///

/// The secrets are shared instantaneously. That is, a person may receive

/// the secret and share it with people in other meetings within the same

/// time frame.

///

/// Return a list of all the people that have the secret after all the

/// meetings have taken place. You may return the answer in any order.

///

/// Example 1:

/// Input: n = 6, meetings = [[1,2,5],[2,3,8],[1,5,10]], firstPerson = 1

/// Output: [0,1,2,3,5]

/// Explanation:

/// At time 0, person 0 shares the secret with person 1.

/// At time 5, person 1 shares the secret with person 2.

/// At time 8, person 2 shares the secret with person 3.

/// At time 10, person 1 shares the secret with person 5.

/// Thus, people 0, 1, 2, 3, and 5 know the secret after all the meetings.

///

/// Example 2:

///

/// Input: n = 4, meetings = [[3,1,3],[1,2,2],[0,3,3]], firstPerson = 3

/// Output: [0,1,3]

/// Explanation:

/// At time 0, person 0 shares the secret with person 3.

/// At time 2, neither person 1 nor person 2 know the secret.

/// At time 3, person 3 shares the secret with person 0 and person 1.

/// Thus, people 0, 1, and 3 know the secret after all the meetings.

///

/// Example 3:

/// Input: n = 5, meetings = [[3,4,2],[1,2,1],[2,3,1]], firstPerson = 1

/// Output: [0,1,2,3,4]

/// Explanation:

/// At time 0, person 0 shares the secret with person 1.

/// At time 1, person 1 shares the secret with person 2, and person 2

/// shares the secret with person 3.

/// Note that person 2 can share the secret at the same time as receiving

/// it.

/// At time 2, person 3 shares the secret with person 4.

/// Thus, people 0, 1, 2, 3, and 4 know the secret after all the meetings.

///

/// Example 4:

/// Input: n = 6, meetings = [[0,2,1],[1,3,1],[4,5,1]], firstPerson = 1

/// Output: [0,1,2,3]

/// Explanation:

/// At time 0, person 0 shares the secret with person 1.

/// At time 1, person 0 shares the secret with person 2, and person 1

/// shares the secret with person 3.

/// Thus, people 0, 1, 2, and 3 know the secret after all the meetings.

/// Constraints:

/// 1. 2 <= n <= 10^5

/// 2. 1 <= meetings.length <= 10^5

/// 3. meetings[i].length == 3

/// 4. 0 <= xi, yi <= n - 1

/// 5. xi != yi

/// 6. 1 <= timei <= 10^5

/// 7. 1 <= firstPerson <= n - 1

/// </summary>

vector<int> LeetCodeGraph::findAllPeople(int n, vector<vector<int>>& meetings, int firstPerson)

{

vector<int> meeting\_times(n, INT\_MAX);

priority\_queue<pair<int, int>> pq;

vector<vector<pair<int, int>>> neighbors(n);

for (size\_t i = 0; i < meetings.size(); i++)

{

neighbors[meetings[i][0]].push\_back(make\_pair(meetings[i][1], meetings[i][2]));

neighbors[meetings[i][1]].push\_back(make\_pair(meetings[i][0], meetings[i][2]));

}

pq.push(make\_pair(0, 0));

pq.push(make\_pair(0, firstPerson));

meeting\_times[0] = 0;

meeting\_times[firstPerson] = 0;

vector<int>result;

while (!pq.empty())

{

pair<int, int> pos = pq.top();

pq.pop();

int time = 0 - pos.first;

int person = pos.second;

if (time > meeting\_times[person])

{

continue;

}

result.push\_back(person);

for (size\_t i = 0; i < neighbors[person].size(); i++)

{

int next\_person = neighbors[person][i].first;

int next\_meeting = neighbors[person][i].second;

// this meeting is too early

if (next\_meeting < time)

{

continue;

}

// we already have a meeting

if (meeting\_times[next\_person] <= next\_meeting)

{

continue;

}

if (meeting\_times[next\_person] = next\_meeting)

{

pq.push(make\_pair(0 - next\_meeting, next\_person));

}

}

}

return result;

}