# Shortest Path

To resolve the shortest path problem, we use a priority queue to record all the reachable destination with the distance from the starting point. The priority queue should be as least first (we can use negative value in C++).

Everytime we fetch a node X from the priority queue with the shortest distance from start and visit all its neighbors, and calculate the new distance from start to this neighbor Y, which should be distance (start, X) + distance (X, Y). If Y is already visited, we only need to revisit Y when a shorter path to Y is discovered.

The whole process is a BFS process.

## 407. Trapping Rain Water II

Hard

Given an m x n matrix of positive integers representing the height of each unit cell in a 2D elevation map, compute the volume of water it is able to trap after raining.

**Note:**

Both *m* and *n* are less than 110. The height of each unit cell is greater than 0 and is less than 20,000.

**Example:**

Given the following 3x6 height map:

[

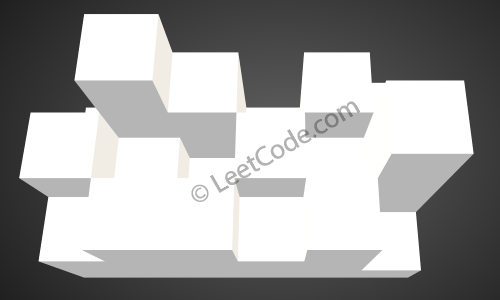
[1,4,3,1,3,2],

[3,2,1,3,2,4],

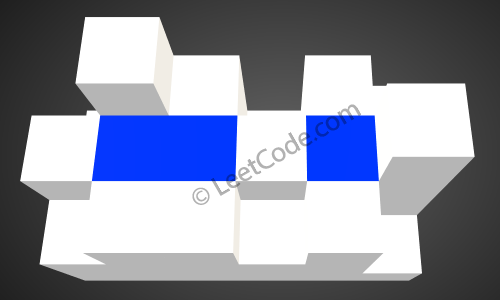
[2,3,3,2,3,1]

]

Return 4.



The above image represents the elevation map [[1,4,3,1,3,2],[3,2,1,3,2,4],[2,3,3,2,3,1]] before the rain.



After the rain, water is trapped between the blocks. The total volume of water trapped is 4.

### Analysis:

We first push all the edge nodes in the priority queue, and pick the lowest one, and check their neighbors, if we found any node which is lower than current one, we fill in water and raise the level as current. We run this process until end.

/// <summary>

/// Leet code #407. Trapping Rain Water II

///

/// Given an m x n matrix of positive integers representing the height of each unit cell

/// in a 2D elevation map, compute the volume of water it is able to trap after raining.

///

/// Note:

/// Both m and n are less than 110. The height of each unit cell is greater than 0 and is less than 20,000.

/// Example:

/// Given the following 3x6 height map:

/// [

/// [1,4,3,1,3,2],

/// [3,2,1,3,2,4],

/// [2,3,3,2,3,1]

/// ]

/// Return 4.

/// </summary>

int LeetCode::trapRainWater(vector<vector<int>>& heightMap)

{

int result = 0;

if (heightMap.empty() || heightMap[0].empty()) return 0;

priority\_queue<vector<int>> search;

vector<vector<int>> visited(heightMap.size(), vector<int>(heightMap[0].size()));

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

{

vector<int> pos = { -heightMap[i][0], (int)i, 0 };

search.push(pos);

visited[i][0] = 1;

int max\_col = heightMap[i].size() - 1;

if (max\_col > 0)

{

pos = { -heightMap[i][max\_col], (int)i, max\_col };

search.push(pos);

visited[i][max\_col] = 1;

}

}

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

{

vector<int> pos = { -heightMap[0][i], 0, (int)i };

search.push(pos);

visited[0][i] = 1;

int max\_row = heightMap.size() - 1;

if (max\_row > 0)

{

pos = { -heightMap[max\_row][i], max\_row, (int)i };

search.push(pos);

visited[max\_row][i] = 1;

}

}

while (!search.empty())

{

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

search.pop();

int floor\_level = -pos[0];

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

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

{

int row = pos[1] + directions[i][0];

int col = pos[2] + directions[i][1];

if ((row < 0) || (row >= (int)heightMap.size()) ||

(col < 0) || (col >= (int)heightMap[0].size()))

{

continue;

}

if (visited[row][col] == 1)

{

continue;

}

if (heightMap[row][col] < floor\_level)

{

result += floor\_level - heightMap[row][col];

search.push({ -floor\_level, row, col });

}

else

{

search.push({ -heightMap[row][col], row, col });

}

visited[row][col] = 1;

}

}

return result;

}

## 505. The Maze II

Medium

There is a **ball** in a maze with empty spaces and walls. The ball can go through empty spaces by rolling **up**, **down**, **left** or **right**, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the ball's **start position**, the **destination** and the **maze**, find the shortest distance for the ball to stop at the destination. The distance is defined by the number of **empty spaces** traveled by the ball from the start position (excluded) to the destination (included). If the ball cannot stop at the destination, return -1.

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The start and destination coordinates are represented by row and column indexes.

**Example 1:**

**Input 1:** a maze represented by a 2D array

0 0 1 0 0

0 0 0 0 0

0 0 0 1 0

1 1 0 1 1

0 0 0 0 0

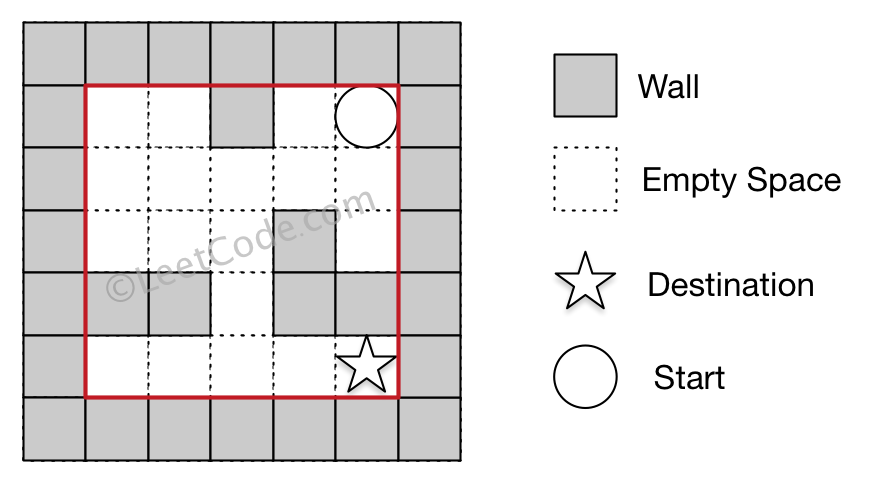
**Input 2:** start coordinate (rowStart, colStart) = (0, 4)

**Input 3:** destination coordinate (rowDest, colDest) = (4, 4)

**Output:** 12

**Explanation:** One shortest way is : left -> down -> left -> down -> right -> down -> right.

The total distance is 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12.



**Example 2:**

**Input 1:** a maze represented by a 2D array

0 0 1 0 0

0 0 0 0 0

0 0 0 1 0

1 1 0 1 1

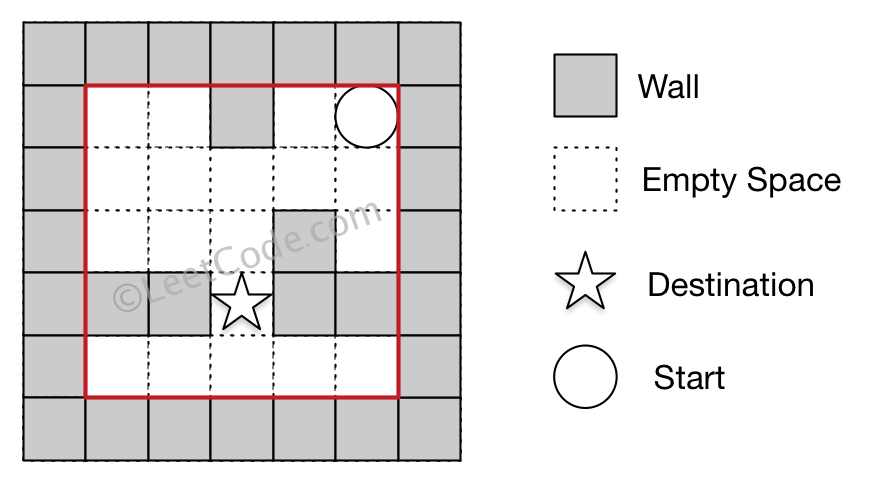
0 0 0 0 0

**Input 2:** start coordinate (rowStart, colStart) = (0, 4)

**Input 3:** destination coordinate (rowDest, colDest) = (3, 2)

**Output:** -1

**Explanation:** There is no way for the ball to stop at the destination.



**Note:**

1. There is only one ball and one destination in the maze.
2. Both the ball and the destination exist on an empty space, and they will not be at the same position initially.
3. The given maze does not contain border (like the red rectangle in the example pictures), but you could assume the border of the maze are all walls.
4. The maze contains at least 2 empty spaces, and both the width and height of the maze won't exceed 100.

### Analysis:

Find the next shortest stop from the starting point and then continue to explorer from there.

/// <summary>

/// Leet code #505. The Maze II

///

/// There is a ball in a maze with empty spaces and walls. The ball can

/// go through empty spaces by rolling up, down, left or right, but it

/// won't stop rolling until hitting a wall. When the ball stops, it could

/// choose the next direction.

///

/// Given the ball's start position, the destination and the maze, find the

/// shortest distance for the ball to stop at the destination. The distance

/// is defined by the number of empty spaces traveled by the ball from the

/// start position (excluded) to the destination (included). If the ball

/// cannot stop at the destination, return -1.

///

/// The maze is represented by a binary 2D array. 1 means the wall and 0 means

/// the empty space. You may assume that the borders of the maze are all walls.

/// The start and destination coordinates are represented by row and column

/// indexes.

///

/// Example 1

/// Input 1: a maze represented by a 2D array

///

/// 0 0 1 0 0

/// 0 0 0 0 0

/// 0 0 0 1 0

/// 1 1 0 1 1

/// 0 0 0 0 0

/// Input 2: start coordinate (rowStart, colStart) = (0, 4)

/// Input 3: destination coordinate (rowDest, colDest) = (4, 4)

///

/// Output: 12

/// Explanation: One shortest way is :

/// left -> down -> left -> down -> right -> down -> right.

/// The total distance is 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12.

///

/// Example 2

/// Input 1: a maze represented by a 2D array

///

/// 0 0 1 0 0

/// 0 0 0 0 0

/// 0 0 0 1 0

/// 1 1 0 1 1

/// 0 0 0 0 0

/// Input 2: start coordinate (rowStart, colStart) = (0, 4)

/// Input 3: destination coordinate (rowDest, colDest) = (3, 2)

/// Output: -1

/// Explanation: There is no way for the ball to stop at the destination.

///

/// Note:

/// 1.There is only one ball and one destination in the maze.

/// 2.Both the ball and the destination exist on an empty space,

/// and they will not be at the same position initially.

/// 3.The given maze does not contain border (like the red rectangle in the

/// example pictures), but you could assume the border of the maze are all

/// walls.

/// 4.The maze contains at least 2 empty spaces, and both the width and height

/// of the maze won't exceed 100.

/// </summary>

int LeetCode::shortestDistance(vector<vector<int>>& maze, vector<int>& start, vector<int>& destination)

{

if (maze.empty() || maze[0].empty()) return -1;

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

visited[start[0]][start[1]] = 0;

priority\_queue<pair<int, vector<int>>> process\_queue;

process\_queue.push(make\_pair(0, start));

while (!process\_queue.empty())

{

start = process\_queue.top().second;

process\_queue.pop();

if (start == destination) break;

shortestDistance(maze, visited, start, process\_queue);

}

int shortest\_distance = visited[destination[0]][destination[1]];

if (shortest\_distance == INT\_MAX) return -1;

else return shortest\_distance;

}

/// <summary>

/// Leet code #505. The Maze II

/// </summary>

void LeetCode::shortestDistance(vector<vector<int>>& maze, vector<vector<int>>& visited,

vector<int>& start, priority\_queue<pair<int, vector<int>>> &process\_queue)

{

vector<vector<int>> next\_list;

int distance = visited[start[0]][start[1]];

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

for (size\_t i = 0; i < 4; i++)

{

int step = 0;

vector<int> pos = start;

while (true)

{

pos[0] += directions[i][0];

pos[1] += directions[i][1];

if ((pos[0] >= 0) && (pos[0] < (int)maze.size()) &&

(pos[1] >= 0) && (pos[1] < (int)maze[0].size()) &&

(maze[pos[0]][pos[1]] == 0))

{

step++;

}

else

{

pos[0] -= directions[i][0];

pos[1] -= directions[i][1];

break;

}

}

if (pos == start) continue;

if (visited[pos[0]][pos[1]] > distance + step)

{

visited[pos[0]][pos[1]] = distance + step;

process\_queue.push(make\_pair(-visited[pos[0]][pos[1]], pos));

}

}

}

## 778. Swim in Rising Water

Hard

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].

### Analysis:

From the starting point, the time we visit the neighbor is the max(time(current), height(neighbor))

/// <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 LeetCode::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];

}

}

## 882. Reachable Nodes In Subdivided Graph

Hard

Starting with an **undirected** graph (the "original graph") with nodes from 0 to N-1, subdivisions are made to some of the edges.

The graph is given as follows: edges[k] is a list of integer pairs (i, j, n) such that (i, j) is an edge of the original graph,

and n is the total number of **new** nodes on that edge.

Then, the edge (i, j) is deleted from the original graph, n new nodes (x\_1, x\_2, ..., x\_n) are added to the original graph,

and n+1 new edges (i, x\_1), (x\_1, x\_2), (x\_2, x\_3), ..., (x\_{n-1}, x\_n), (x\_n, j) are added to the original graph.

Now, you start at node 0 from the original graph, and in each move, you travel along one edge.

Return how many nodes you can reach in at most M moves.

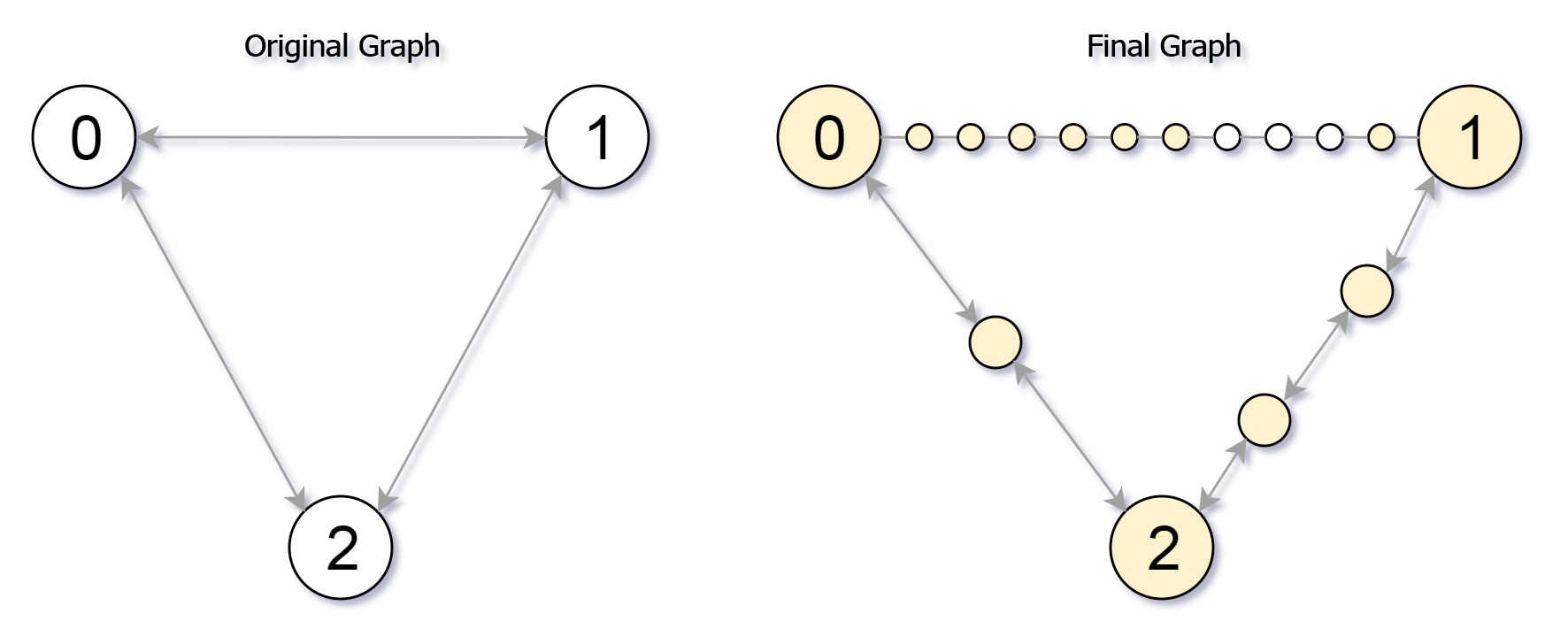
**Example 1:**

**Input:** edges = [[0,1,10],[0,2,1],[1,2,2]], M = 6, N = 3

**Output:** 13

**Explanation:**

The nodes that are reachable in the final graph after M = 6 moves are indicated below.



**Example 2:**

**Input:** edges = [[0,1,4],[1,2,6],[0,2,8],[1,3,1]], M = 10, N = 4

**Output:** 23

**Note:**

1. 0 <= edges.length <= 10000
2. 0 <= edges[i][0] < edges[i][1] < N
3. There does not exist any i != j for which edges[i][0] == edges[j][0] and edges[i][1] == edges[j][1].
4. The original graph has no parallel edges.
5. 0 <= edges[i][2] <= 10000
6. 0 <= M <= 10^9
7. 1 <= N <= 3000
8. A reachable node is a node that can be travelled to using at most M moves starting from node 0.

### Analysis:

Seach from the next node with modest unused steps.

/// <summary>

/// Leet code #882. Reachable Nodes In Subdivided Graph

///

/// Starting with an undirected graph (the "original graph") with nodes from

/// 0 to N-1, subdivisions are made to some of the edges.

///

/// The graph is given as follows: edges[k] is a list of integer pairs

/// (i, j, n) such that (i, j) is an edge of the original graph,

///

/// and n is the total number of new nodes on that edge.

///

/// Then, the edge (i, j) is deleted from the original graph, n new nodes

/// (x\_1, x\_2, ..., x\_n) are added to the original graph,

///

/// and n+1 new edges (i, x\_1), (x\_1, x\_2), (x\_2, x\_3), ..., (x\_{n-1}, x\_n),

/// (x\_n, j) are added to the original graph.

///

/// Now, you start at node 0 from the original graph, and in each move, you

/// travel along one edge.

///

/// Return how many nodes you can reach in at most M moves.

///

/// Example 1:

///

/// Input: edges = [[0,1,10],[0,2,1],[1,2,2]], M = 6, N = 3

/// Output: 13

/// Explanation:

/// The nodes that are reachable in the final graph after M = 6 moves are

/// indicated below.

///

/// Example 2:

///

/// Input: edges = [[0,1,4],[1,2,6],[0,2,8],[1,3,1]], M = 10, N = 4

/// Output: 23

///

/// Note:

///

/// 1. 0 <= edges.length <= 10000

/// 2. 0 <= edges[i][0] < edges[i][1] < N

/// 3. There does not exist any i != j for which edges[i][0] == edges[j][0] and

/// edges[i][1] == edges[j][1].

/// 4. The original graph has no parallel edges.

/// 5. 0 <= edges[i][2] <= 10000

/// 6. 0 <= M <= 10^9

/// 7. 1 <= N <= 3000

/// 8. reachable node is a node that can be travelled to using at most M moves

/// starting from node 0.

/// </summary>

int LeetCode::reachableNodes(vector<vector<int>>& edges, int M, int N)

{

int result = 0;

unordered\_map<int, unordered\_map<int, int>> edge\_nodes;

vector<int> visited(N);

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

{

edge\_nodes[edges[i][0]][edges[i][1]] = edges[i][2];

edge\_nodes[edges[i][1]][edges[i][0]] = edges[i][2];

}

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

search.push(make\_pair(M, 0));

while (!search.empty())

{

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

search.pop();

int node = pos.second;

int step = pos.first;

if (visited[node] == 1) continue;

result++;

visited[node] = 1;

for (auto itr : edge\_nodes[node])

{

int target\_node = itr.first;

int nodes\_in\_middle = itr.second;

if (nodes\_in\_middle + 1 <= step)

{

result += nodes\_in\_middle;

if (visited[target\_node] == 0)

{

search.push(make\_pair(step - nodes\_in\_middle - 1, target\_node));

edge\_nodes[node][target\_node] = 0;

edge\_nodes[target\_node][node] = 0;

}

}

else

{

result += step;

edge\_nodes[node][target\_node] -= step;

edge\_nodes[target\_node][node] -= step;

}

}

}

return result;

}

## 1102. Path With Maximum Minimum Value

Medium

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

### Analysis:

Starting from the left corner, we record the max score to each node, we choose the path with maximum score to the a node which not visited, or revisit due to a higher score is discovered. And exploer all the neighbors, and store score (the minimum value on the path) to the new node. If we found there is a higher score to the node we already visited, we will revisit it with the new score.

/// <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 LeetCode::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;

}

## 1168. Optimize Water Distribution in a Village

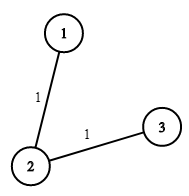
Hard

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], 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 <= n <= 10000
* wells.length == n
* 0 <= wells[i] <= 10^5
* 1 <= pipes.length <= 10000
* 1 <= pipes[i][0], pipes[i][1] <= n
* 0 <= pipes[i][2] <= 10^5
* pipes[i][0] != pipes[i][1]

### Analysis:

Because the pipe can only be built on wells, so if we already visit a village, and later we found a pipe to this viliage with even lesst cost, we should not change mind, otherwise we will end up with only pipes without wells.

/// <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 LeetCode::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;

}

## 174. Dungeon Game

Hard

The demons had captured the princess (**P**) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid. Our valiant knight (**K**) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (*negative* integers) upon entering these rooms; other rooms are either empty (*0's*) or contain magic orbs that increase the knight's health (*positive* integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

**Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.**

For example, given the dungeon below, the initial health of the knight must be at least **7** if he follows the optimal path RIGHT-> RIGHT -> DOWN -> DOWN.

|  |  |  |
| --- | --- | --- |
| -2 (K) | -3 | 3 |
| -5 | -10 | 1 |
| 10 | 30 | -5 (P) |

**Note:**

* The knight's health has no upper bound.
* Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

### Analysis:

This problem can be solved as a DP problem which we traverse from the destination place, (P) , by upper and left, track the minimum health needed in each cell. But this problem can also be solved as shortest path, starting from the entrance keep track on the maximum health remaining (which can be negative vale) on each new destination. Shortest path should be faster than DP in theory.

/// <summary>

/// Leet code #174. Dungeon Game

/// The demons had captured the princess (P) and imprisoned her in the

/// bottom-right corner of a dungeon. The dungeon consists

/// of M x N rooms laid out in a 2D grid. Our valiant knight (K) was

/// initially positioned in the top-left room and must fight

/// his way through the dungeon to rescue the princess.

///

/// The knight has an initial health point represented by a positive

/// integer. If at any point his health point drops to 0

/// or below, he dies immediately.

/// Some of the rooms are guarded by demons, so the knight loses health

/// (negative integers) upon entering these rooms;

/// other rooms are either empty (0's) or contain magic orbs that

/// increase the knight's health (positive integers).

/// In order to reach the princess as quickly as possible, the

/// knight decides to move only

/// rightward or downward in each step.

/// Write a function to determine the knight's minimum initial health

/// so that he is able to rescue the princess.

/// For example, given the dungeon below, the initial health of the

/// knight must be at least 7 if he follows the optimal path

/// RIGHT-> RIGHT -> DOWN -> DOWN.

/// -2 (K) -3 3

/// -5 -10 1

/// 10 30 -5 (P)

/// Notes:

/// The knight's health has no upper bound.

/// Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

/// </summary>

int LeetCode::calculateMinimumHP(vector<vector<int>>& dungeon)

{

if ((dungeon.size() == 0) || dungeon[0].size() == 0) return 1;

vector<vector<int>> min\_hp(dungeon.size());

for (int i = dungeon.size() - 1; i >= 0; i--)

{

min\_hp[i] = vector<int>(dungeon[i].size());

for (int j = dungeon[i].size() - 1; j >= 0; j--)

{

if ((i == dungeon.size() - 1) && (j == dungeon[i].size() - 1))

{

min\_hp[i][j] = max(1, 0 - dungeon[i][j] + 1);

}

else if (i == dungeon.size() - 1)

{

min\_hp[i][j] = max(1, min\_hp[i][j + 1] - dungeon[i][j]);

}

else if (j == dungeon[i].size() - 1)

{

min\_hp[i][j] = max(1, min\_hp[i + 1][j] - dungeon[i][j]);

}

else

{

min\_hp[i][j] = min(min\_hp[i + 1][j], min\_hp[i][j + 1]) - dungeon[i][j];

min\_hp[i][j] = max(1, min\_hp[i][j]);

}

}

}

return min\_hp[0][0];

}

/// <summary>

/// Leet code #174. Dungeon Game

/// The demons had captured the princess (P) and imprisoned her in the

/// bottom-right corner of a dungeon. The dungeon consists

/// of M x N rooms laid out in a 2D grid. Our valiant knight (K) was

/// initially positioned in the top-left room and must fight

/// his way through the dungeon to rescue the princess.

///

/// The knight has an initial health point represented by a positive

/// integer. If at any point his health point drops to 0

/// or below, he dies immediately.

/// Some of the rooms are guarded by demons, so the knight loses health

/// (negative integers) upon entering these rooms;

/// other rooms are either empty (0's) or contain magic orbs that

/// increase the knight's health (positive integers).

/// In order to reach the princess as quickly as possible, the

/// knight decides to move only

/// rightward or downward in each step.

/// Write a function to determine the knight's minimum initial health

/// so that he is able to rescue the princess.

/// For example, given the dungeon below, the initial health of the

/// knight must be at least 7 if he follows the optimal path

/// RIGHT-> RIGHT -> DOWN -> DOWN.

/// -2 (K) -3 3

/// -5 -10 1

/// 10 30 -5 (P)

/// Notes:

/// The knight's health has no upper bound.

/// Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

/// </summary>

int LeetCode::calculateMinimumHP(vector<vector<int>>& dungeon)

{

vector<vector<int>> dp(dungeon.size(), vector<int>(dungeon[0].size(), INT\_MIN));

dp[0][0] = dungeon[0][0];

vector<int> pos(3);

pos = { dungeon[0][0], 0, 0 };

priority\_queue<vector<int>> search;

search.push(pos);

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

int result = 1;

while (!search.empty())

{

pos = search.top();

search.pop();

result = max(result, 1 - pos[0]);

if (pos[1] == dungeon.size() - 1 && pos[2] == dungeon[0].size() - 1)

{

break;

}

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

{

vector<int> next = pos;

next[1] = pos[1] + directions[d][0];

next[2] = pos[2] + directions[d][1];

if (next[1] >= (int)dungeon.size()) continue;

if (next[2] >= (int)dungeon[0].size()) continue;

next[0] = pos[0] + dungeon[next[1]][next[2]];

if (next[0] > dp[next[1]][next[2]])

{

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

search.push(next);

}

}

}

return result;

}

## 847. Shortest Path Visiting All Nodes

Hard

An undirected, connected graph of N nodes (labeled 0, 1, 2, ..., N-1) is given as graph.

graph.length = N, and j != i is in the list graph[i] exactly once, if and only if nodes i and j are connected.

Return the length of the shortest path that visits every node. You may start and stop at any node, you may revisit nodes multiple times, and you may reuse edges.

**Example 1:**

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

**Output:** 4

**Explanation**: One possible path is [1,0,2,0,3]

**Example 2:**

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

**Output:** 4

**Explanation**: One possible path is [0,1,4,2,3]

**Note:**

1. 1 <= graph.length <= 12
2. 0 <= graph[i].length < graph.length

### Analysis:

This is a typical salesman problem. It may be too difficult for 45 minutes interview.

A user can start from any node and travel to next node. When user end up any node, given the nodes he already visited, we call it a state, for any state, we will only remember the shortest path.

Please refer to the following detailed explanation.

<https://simple.wikipedia.org/wiki/Travelling_salesman_problem>

The travelling salesman is to say for a list of given cites, the distances between the cities, the sales man need to travel all the cities at least once, what is the shortest way.

This an NP hard problem, so the time complexity is exponential. We should make our code pattern clean, short and easy to understand.

Please think how you handle the following issues in this problem.

1. How to search the path?

This can be a Breadth First Search, but it will not search once, you need to start from every city. But generally speaking, it is similar.

1. How to mark the current state?

During the search you need to represent what is the current state. For travelling salesman problem, the current state is which city the salesman currently reached (last visited), plus all the cities the salesman has already visited.

1. How to mark the cities already travelled?

We can use a bitmap, (a long integer will cover for 64), each bit represents the city is already visited or not.

1. How to optimize the algorithm by cutting branches?

For any state above, you only need to remember the shortest distance.

/// <summary>

/// Leet code #847. Shortest Path Visiting All Nodes

///

/// An undirected, connected graph of N nodes (labeled 0, 1, 2, ..., N-1)

/// is given as graph.

///

/// graph.length = N, and j != i is in the list graph[i] exactly once, if

/// and only if nodes i and j are connected.

///

/// Return the length of the shortest path that visits every node. You may

/// start and stop at any node, you may revisit nodes multiple times, and

/// you may reuse edges.

///

/// Example 1:

///

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

/// Output: 4

/// Explanation: One possible path is [1,0,2,0,3]

///

/// Example 2:

///

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

/// Output: 4

/// Explanation: One possible path is [0,1,4,2,3]

///

///

/// Note:

/// 1. 1 <= graph.length <= 12

/// 2. 0 <= graph[i].length < graph.length

/// </summary>

int LeetCode::shortestPathLength(vector<vector<int>>& graph)

{

size\_t size = graph.size();

vector<vector<int>> dp(1 << size, vector<int>(size, -1));

queue<pair<int, int>> search;

// we start from each node

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

{

int cover = 1 << i;

dp[cover][i] = 0;

// make the current node as last node

search.push(make\_pair(cover, i));

}

while (!search.empty())

{

pair<int, int> route = search.front();

search.pop();

// we check from current last node what are next nodes to travel

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

{

int next = graph[route.second][i];

// Normally we need to check if we have already visited the

// next node, however the distance check will save us this

// effort, if the node is already visited, visit again will

// give us no chance to shorten the path

int cover = route.first | (1 << next);

if (dp[cover][next] == -1 ||

dp[cover][next] > dp[route.first][route.second] + 1)

{

dp[cover][next] = dp[route.first][route.second] + 1;

search.push(make\_pair(cover, next));

}

}

}

int full = (1 << size) - 1;

int result = dp[full][0];

// starting from every node, look for shortest path

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

{

result = min(result, dp[full][i]);

}

return result;

}

## 943. Find the Shortest Superstring

Hard

Given an array A of strings, find any smallest string that contains each string in A as a substring.

We may assume that no string in A is substring of another string in A.

**Example 1:**

**Input:** ["alex","loves","leetcode"]

**Output:** "alexlovesleetcode"

**Explanation:** All permutations of "alex","loves","leetcode" would also be accepted.

**Example 2:**

**Input:** ["catg","ctaagt","gcta","ttca","atgcatc"]

**Output:** "gctaagttcatgcatc"

**Note:**

1. 1 <= A.length <= 12
2. 1 <= A[i].length <= 20

### Analysis:

This is another salesman problem, the only difference is that you care which words are visited which are not, and you do not care which is the last word.

/// <summary>

/// Leet code #943. Find the Shortest Superstring

/// </summary>

string LeetCodeGraph::calculateOverlapString(string& string1, string& string2)

{

int prefix = min(string1.size(), string2.size());

string result = string1 + string2;

while (prefix > 0)

{

if (string1.substr(string1.size() - prefix) == string2.substr(0, prefix))

{

result = string1 + string2.substr(prefix);

break;

}

else if (string2.substr(string2.size() - prefix) == string1.substr(0, prefix))

{

result = string2 + string1.substr(prefix);

break;

}

prefix--;

}

return result;

}

/// <summary>

/// Leet code #943. Find the Shortest Superstring

///

/// Given an array A of strings, find any smallest string that contains each

/// string in A as a substring.

///

/// We may assume that no string in A is substring of another string in A.

///

/// Example 1:

/// Input: ["alex","loves","leetcode"]

/// Output: "alexlovesleetcode"

/// Explanation: All permutations of "alex","loves","leetcode" would also be

/// accepted.

///

/// Example 2:

/// Input: ["catg","ctaagt","gcta","ttca","atgcatc"]

/// Output: "gctaagttcatgcatc"

///

/// Note:

/// 1. 1 <= A.length <= 12

/// 2. 1 <= A[i].length <= 20

/// </summary>

string LeetCodeGraph::shortestSuperstring(vector<string>& A)

{

vector<string> dp(1 << A.size());

int bit\_full = (1 << A.size()) - 1;

queue<int> search;

search.push(0);

while (!search.empty())

{

int bit = search.front();

search.pop();

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

{

int bit\_set = 1 << i;

if ((bit & bit\_set) == 0)

{

bit\_set = bit | bit\_set;

if (dp[bit\_set].empty()) search.push(bit\_set);

string cand = calculateOverlapString(dp[bit], A[i]);

if (dp[bit\_set].empty() || dp[bit\_set].size() > cand.size())

{

dp[bit\_set] = cand;

}

}

}

}

return dp[bit\_full];

}

## 1066. Campus Bikes II

Medium

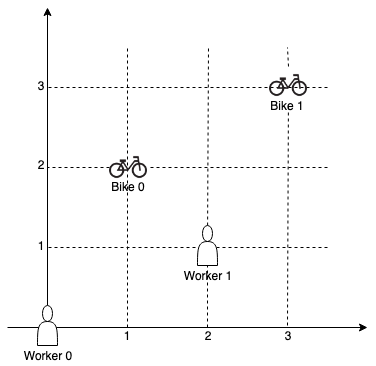
On a campus represented as a 2D grid, there are N workers and M bikes, with N <= M. Each worker and bike is a 2D coordinate on this grid.

We assign one unique bike to each worker so that the sum of the Manhattan distances between each worker and their assigned bike is minimized.

The Manhattan distance between two points p1 and p2 is Manhattan(p1, p2) = |p1.x - p2.x| + |p1.y - p2.y|.

Return the minimum possible sum of Manhattan distances between each worker and their assigned bike.

**Example 1:**



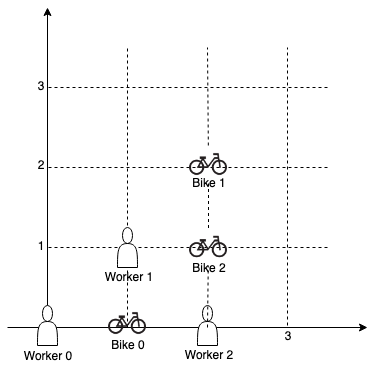
**Input:** workers = [[0,0],[2,1]], bikes = [[1,2],[3,3]]

**Output:** 6

**Explanation:**

We assign bike 0 to worker 0, bike 1 to worker 1. The Manhattan distance of both assignments is 3, so the output is 6.

**Example 2:**



**Input:** workers = [[0,0],[1,1],[2,0]], bikes = [[1,0],[2,2],[2,1]]

**Output:** 4

**Explanation:**

We first assign bike 0 to worker 0, then assign bike 1 to worker 1 or worker 2, bike 2 to worker 2 or worker 1. Both assignments lead to sum of the Manhattan distances as 4.

**Note:**

1. 0 <= workers[i][0], workers[i][1], bikes[i][0], bikes[i][1] < 1000
2. All worker and bike locations are distinct.
3. 1 <= workers.length <= bikes.length <= 10

### Analysis:

We can do simply backtracking, but using shortest path will optimize the algorithm, if we have select the same people, same bikes we should only choose the least total distance.

/// <summary>

/// Leet code #1066. Campus Bikes II

/// </summary>

int LeetCode::assignBikesII(vector<vector<pair<int, int>>> &worker\_distance, int worker,

vector<int> &visited, int sum, unordered\_map<int, int>& cache, int& result)

{

if (worker == worker\_distance.size())

{

result = min(sum, result);

return result;

}

for (size\_t i = 0; i < worker\_distance[worker].size(); i++)

{

int bike = worker\_distance[worker][i].second;

if (visited[bike] == 1) continue;

visited[bike] = 1;

sum += worker\_distance[worker][i].first;

int key = 0;

for (size\_t j = 0; j < visited.size(); j++)

{

key = (key << 1) | visited[j];

}

if (cache.count(key) == 0 || cache[key] > sum)

{

cache[key] = sum;

result = assignBikesII(worker\_distance, worker + 1, visited, sum, cache, result);

}

sum -= worker\_distance[worker][i].first;

visited[bike] = 0;

}

return result;

}

/// <summary>

/// Leet code #1066. Campus Bikes II

///

/// On a campus represented as a 2D grid, there are N workers and M bikes,

/// with N <= M. Each worker and bike is a 2D coordinate on this grid.

///

/// We assign one unique bike to each worker so that the sum of the

/// Manhattan distances between each worker and their assigned bike is

/// minimized.

///

/// The Manhattan distance between two points p1 and p2 is

/// Manhattan(p1, p2) = |p1.x - p2.x| + |p1.y - p2.y|.

///

/// Return the minimum possible sum of Manhattan distances between each

/// worker and their assigned bike.

///

/// Example 1:

/// Input: workers = [[0,0],[2,1]], bikes = [[1,2],[3,3]]

/// Output: 6

/// Explanation:

/// We assign bike 0 to worker 0, bike 1 to worker 1. The Manhattan

/// distance of both assignments is 3, so the output is 6.

///

/// Example 2:

/// Input: workers = [[0,0],[1,1],[2,0]], bikes = [[1,0],[2,2],[2,1]]

/// Output: 4

/// Explanation:

/// We first assign bike 0 to worker 0, then assign bike 1 to worker 1

/// or worker 2, bike 2 to worker 2 or worker 1. Both assignments lead

/// to sum of the Manhattan distances as 4.

///

/// Note:

/// 1. 0 <= workers[i][0], workers[i][1], bikes[i][0], bikes[i][1] < 1000

/// 2. All worker and bike locations are distinct.

/// 3. 1 <= workers.length <= bikes.length <= 10

/// </summary>

int LeetCode::assignBikesII(vector<vector<int>>& workers, vector<vector<int>>& bikes)

{

vector<vector<pair<int, int>>> worker\_distance(workers.size());

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

{

for (size\_t j = 0; j < bikes.size(); j++)

{

int distance = std::abs(workers[i][0] - bikes[j][0]) +

std::abs(workers[i][1] - bikes[j][1]);

worker\_distance[i].push\_back(make\_pair(distance, j));

}

sort(worker\_distance[i].begin(), worker\_distance[i].end());

}

vector<int> visited(bikes.size());

int sum = 0;

int result = INT\_MAX;

unordered\_map<int, int> cache;

assignBikesII(worker\_distance, 0, visited, sum, cache, result);

return result;

}