LeetCode Training Day 23 BFS

BFS is a typical graph search pattern which is discussed in chapter 7. Here we just spend more time to look at some typical pattern.

BFS pattern is that we put our starting point into queue, set the visited flag for the starting point then we run the queue until empty. For every round we take a node from queue and search its neighbors, inserted unvisited node to queue and mark them visited. When the queue is empty, our search complete.

Please remember that for each round, it means our steps increase by 1, so if we care about how many steps we reach to the destination, we should get the size of the current queue and loop that each node, on each round we will insert nodes in the next round in the queue.

The search in graph is same as what we discussed in chapter 7, if we do it in grid, we run 4 directions; if we do connected graph, we should convert edges to neighbors.

## 752. Open the Lock

Medium

You have a lock in front of you with 4 circular wheels. Each wheel has 10 slots: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'. The wheels can rotate freely and wrap around: for example we can turn '9' to be '0', or '0' to be '9'. Each move consists of turning one wheel one slot.

The lock initially starts at '0000', a string representing the state of the 4 wheels.

You are given a list of deadends dead ends, meaning if the lock displays any of these codes, the wheels of the lock will stop turning and you will be unable to open it.

Given a target representing the value of the wheels that will unlock the lock, return the minimum total number of turns required to open the lock, or -1 if it is impossible.

**Example 1:**

**Input:** deadends = ["0201","0101","0102","1212","2002"], target = "0202"

**Output:** 6

**Explanation:**

A sequence of valid moves would be "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202".

Note that a sequence like "0000" -> "0001" -> "0002" -> "0102" -> "0202" would be invalid,

because the wheels of the lock become stuck after the display becomes the dead end "0102".

**Example 2:**

**Input:** deadends = ["8888"], target = "0009"

**Output:** 1

**Explanation:**

We can turn the last wheel in reverse to move from "0000" -> "0009".

**Example 3:**

**Input:** deadends = ["8887","8889","8878","8898","8788","8988","7888","9888"], target = "8888"

**Output:** -1

**Explanation:**

We can't reach the target without getting stuck.

**Example 4:**

**Input:** deadends = ["0000"], target = "8888"

**Output:** -1

**Note:**

1. The length of deadends will be in the range [1, 500].
2. target will not be in the list deadends.
3. Every string in deadends and the string target will be a string of 4 digits from the 10,000 possibilities '0000' to '9999'.

### Analysis:

You may be tempted to use DFS, but in fact BFS is more suitable, because you are asked the minimum turns. From any digit with one move you will get the next state.

/// <summary>

/// Leet code #752. Open the Lock

///

/// You have a lock in front of you with 4 circular wheels. Each wheel has

/// 10 slots: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'. The wheels

/// can rotate freely and wrap around: for example we can turn '9' to be

/// '0', or '0' to be '9'. Each move consists of turning one wheel one

/// slot.

///

/// The lock initially starts at '0000', a string representing the state

/// of the 4 wheels.

///

/// You are given a list of deadends dead ends, meaning if the lock

/// displays any of these codes, the wheels of the lock will stop turning

/// and you will be unable to open it.

///

/// Given a target representing the value of the wheels that will unlock

/// the lock, return the minimum total number of turns required to open

/// the lock, or -1 if it is impossible.

///

/// Example 1:

/// Input: deadends = ["0201","0101","0102","1212","2002"], target = "0202"

/// Output: 6

/// Explanation:

/// A sequence of valid moves would be "0000" -> "1000" -> "1100" ->

/// "1200" -> "1201" -> "1202" -> "0202".

/// Note that a sequence like "0000" -> "0001" -> "0002" -> "0102" ->

/// "0202" would be invalid,

/// because the wheels of the lock become stuck after the display

/// becomes the dead end "0102".

/// Example 2:

/// Input: deadends = ["8888"], target = "0009"

/// Output: 1

/// Explanation:

/// We can turn the last wheel in reverse to move from "0000" -> "0009".

/// Example 3:

/// Input: deadends = ["8887","8889","8878","8898","8788","8988","7888",

/// "9888"], target = "8888"

/// Output: -1

/// Explanation:

/// We can't reach the target without getting stuck.

/// Example 4:

/// Input: deadends = ["0000"], target = "8888"

/// Output: -1

/// Note:

/// The length of deadends will be in the range [1, 500].

/// target will not be in the list deadends.

/// Every string in deadends and the string target will be a string of 4

/// digits from the 10,000 possibilities '0000' to '9999'.

/// </summary>

int LeetCode::openLock(vector<string>& deadends, string target)

{

queue<string> bfs;

unordered\_set<string> visited;

unordered\_set<string> deadset;

for (string str : deadends) deadset.insert(str);

int result = 0;

string lock = "0000";

if (deadset.count(lock) == 0)

{

visited.insert(lock);

bfs.push(lock);

}

while (!bfs.empty())

{

size\_t size = bfs.size();

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

{

string lock = bfs.front();

bfs.pop();

if (deadset.count(lock) > 0) continue;

if (lock == target) return result;

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

{

for (int k = -1; k <= 1; k += 2)

{

string next = lock;

next[j] = '0' + ((next[j] - '0') + k + 10) % 10;

if ((deadset.count(lock) == 0) && (visited.count(next) == 0))

{

visited.insert(next);

bfs.push(next);

}

}

}

}

result++;

}

return -1;

}

## 529. Minesweeper

Medium

Let's play the minesweeper game ([Wikipedia](https://en.wikipedia.org/wiki/Minesweeper_(video_game)), [online game](http://minesweeperonline.com/))!

You are given a 2D char matrix representing the game board. **'M'** represents an **unrevealed** mine, **'E'** represents an **unrevealed** empty square, **'B'** represents a **revealed** blank square that has no adjacent (above, below, left, right, and all 4 diagonals) mines, **digit** ('1' to '8') represents how many mines are adjacent to this **revealed** square, and finally **'X'** represents a **revealed** mine.

Now given the next click position (row and column indices) among all the **unrevealed** squares ('M' or 'E'), return the board after revealing this position according to the following rules:

1. If a mine ('M') is revealed, then the game is over - change it to **'X'**.
2. If an empty square ('E') with **no adjacent mines** is revealed, then change it to revealed blank ('B') and all of its adjacent **unrevealed** squares should be revealed recursively.
3. If an empty square ('E') with **at least one adjacent mine** is revealed, then change it to a digit ('1' to '8') representing the number of adjacent mines.
4. Return the board when no more squares will be revealed.

**Example 1:**

**Input:**

[['E', 'E', 'E', 'E', 'E'],

['E', 'E', 'M', 'E', 'E'],

['E', 'E', 'E', 'E', 'E'],

['E', 'E', 'E', 'E', 'E']]

Click : [3,0]

**Output:**

[['B', '1', 'E', '1', 'B'],

['B', '1', 'M', '1', 'B'],

['B', '1', '1', '1', 'B'],

['B', 'B', 'B', 'B', 'B']]

**Explanation:**

Graphical user interface, application, Teams

Description automatically generated

**Example 2:**

**Input:**

[['B', '1', 'E', '1', 'B'],

['B', '1', 'M', '1', 'B'],

['B', '1', '1', '1', 'B'],

['B', 'B', 'B', 'B', 'B']]

Click : [1,2]

**Output:**

[['B', '1', 'E', '1', 'B'],

['B', '1', 'X', '1', 'B'],

['B', '1', '1', '1', 'B'],

['B', 'B', 'B', 'B', 'B']]

**Explanation:**

Graphical user interface, application, Teams

Description automatically generated

**Note:**

1. The range of the input matrix's height and width is [1,50].
2. The click position will only be an unrevealed square ('M' or 'E'), which also means the input board contains at least one clickable square.
3. The input board won't be a stage when game is over (some mines have been revealed).
4. For simplicity, not mentioned rules should be ignored in this problem. For example, you **don't** need to reveal all the unrevealed mines when the game is over, consider any cases that you will win the game or flag any squares.

### Analysis:

This is a simulation problem, you do BFS search from the neighbors from the click point, the search will end by the cell with mines nearby.

/// <summary>

/// Leet code #529. Minesweeper

///

/// Let's play the minesweeper game (Wikipedia, online game)!

/// You are given a 2D char matrix representing the game board. 'M'

/// represents an unrevealed mine, 'E' represents an unrevealed empty square,

/// 'B' represents a revealed blank square that has no adjacent (above, below,

/// left, right, and all 4 diagonals) mines, digit ('1' to '8') represents how

/// many mines are adjacent to this revealed square, and finally 'X' represents

/// a revealed mine.

///

/// Now given the next click position (row and column indices) among all the

/// unrevealed squares ('M' or 'E'), return the board after revealing this

/// position according to the following rules:

///

/// If a mine ('M') is revealed, then the game is over - change it to 'X'.

/// If an empty square ('E') with no adjacent mines is revealed, then change it

/// to revealed blank ('B') and all of its adjacent unrevealed squares should be

/// revealed recursively.

/// If an empty square ('E') with at least one adjacent mine is revealed, then

/// change it to a digit ('1' to '8') representing the number of adjacent mines.

/// Return the board when no more squares will be revealed.

/// Example 1:

/// Input:

///

/// [['E', 'E', 'E', 'E', 'E'],

/// ['E', 'E', 'M', 'E', 'E'],

/// ['E', 'E', 'E', 'E', 'E'],

/// ['E', 'E', 'E', 'E', 'E']]

/// Click : [3,0]

/// Output:

///

/// [['B', '1', 'E', '1', 'B'],

/// ['B', '1', 'M', '1', 'B'],

/// ['B', '1', '1', '1', 'B'],

/// ['B', 'B', 'B', 'B', 'B']]

///

/// Explanation:

///

/// Example 2:

/// Input:

///

/// [['B', '1', 'E', '1', 'B'],

/// ['B', '1', 'M', '1', 'B'],

/// ['B', '1', '1', '1', 'B'],

/// ['B', 'B', 'B', 'B', 'B']]

/// Click : [1,2]

/// Output:

///

/// [['B', '1', 'E', '1', 'B'],

/// ['B', '1', 'X', '1', 'B'],

/// ['B', '1', '1', '1', 'B'],

/// ['B', 'B', 'B', 'B', 'B']]

///

/// Explanation:

/// Note:

/// The range of the input matrix's height and width is [1,50].

/// The click position will only be an unrevealed square ('M' or 'E'),

/// which also means the input board contains at least one clickable square.

/// The input board won't be a stage when game is over (some mines have been revealed).

/// For simplicity, not mentioned rules should be ignored in this problem. For example,

/// you don't need to reveal all the unrevealed mines when the game is over, consider

/// any cases that you will win the game or flag any squares.

/// </summary>

vector<vector<char>> LeetCodeArray::updateBoard(vector<vector<char>>& board, vector<int>& click)

{

vector<vector<int>> directions =

{

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

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

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

};

vector<int> curr = click;

if (board.empty() || board[0].empty() || curr.size() < 2) return board;

queue<vector<int>> process\_queue;

if (board[curr[0]][curr[1]] == 'M')

{

board[curr[0]][curr[1]] = 'X';

return board;

}

process\_queue.push(curr);

while (!process\_queue.empty())

{

curr = process\_queue.front();

process\_queue.pop();

int count = 0;

vector<vector<int>> result;

if (board[curr[0]][curr[1]] == 'E')

{

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

{

vector<int> next = curr;

next[0] = curr[0] + directions[d][0];

next[1] = curr[1] + directions[d][1];

if (next[0] < 0 || next[0] >= (int)board.size() ||

next[1] < 0 || next[1] >= (int)board[0].size())

{

continue;

}

if (board[next[0]][next[1]] == 'M') count++;

else result.push\_back(next);

}

if (count == 0)

{

board[curr[0]][curr[1]] = 'B';

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

{

process\_queue.push(result[i]);

}

}

else board[curr[0]][curr[1]] = '0' + count;

}

}

return board;

}

## 773. Sliding Puzzle

Hard

On a 2x3 board, there are 5 tiles represented by the integers 1 through 5, and an empty square represented by 0.

A move consists of choosing 0 and a 4-directionally adjacent number and swapping it.

The state of the board is *solved* if and only if the board is [[1,2,3],[4,5,0]].

Given a puzzle board, return the least number of moves required so that the state of the board is solved. If it is impossible for the state of the board to be solved, return -1.

**Examples:**

**Input:** board = [[1,2,3],[4,0,5]]

**Output:** 1

**Explanation:** Swap the 0 and the 5 in one move.

**Input:** board = [[1,2,3],[5,4,0]]

**Output:** -1

**Explanation:** No number of moves will make the board solved.

**Input:** board = [[4,1,2],[5,0,3]]

**Output:** 5

**Explanation:** 5 is the smallest number of moves that solves the board.

An example path:

After move 0: [[4,1,2],[5,0,3]]

After move 1: [[4,1,2],[0,5,3]]

After move 2: [[0,1,2],[4,5,3]]

After move 3: [[1,0,2],[4,5,3]]

After move 4: [[1,2,0],[4,5,3]]

After move 5: [[1,2,3],[4,5,0]]

**Input:** board = [[3,2,4],[1,5,0]]

**Output:** 14

**Note:**

* board will be a 2 x 3 array as described above.
* board[i][j] will be a permutation of [0, 1, 2, 3, 4, 5].

### Analysis:

We need to get the shortest step, so we do BFS, the difficult part here is how to remember the visited pattern, given only 5 slides and 6 place, it is not difficult.

/// <summary>

/// Leetcode #773. Sliding Puzzle

///

/// On a 2x3 board, there are 5 tiles represented by the integers 1

/// through 5, and an empty square represented by 0.

///

/// A move consists of choosing 0 and a 4-directionally adjacent number

/// and swapping it.

///

/// The state of the board is solved if and only if the board is

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

///

/// Given a puzzle board, return the least number of moves required so

/// that the state of the board is solved. If it is impossible for the

/// state of the board to be solved, return -1.

///

/// Examples:

///

/// Input: board = [[1,2,3],[4,0,5]]

/// Output: 1

/// Explanation: Swap the 0 and the 5 in one move.

///

/// Input: board = [[1,2,3],[5,4,0]]

/// Output: -1

/// Explanation: No number of moves will make the board solved.

///

/// Input: board = [[4,1,2],[5,0,3]]

/// Output: 5

/// Explanation: 5 is the smallest number of moves that solves the board.

/// An example path:

/// After move 0: [[4,1,2],[5,0,3]]

/// After move 1: [[4,1,2],[0,5,3]]

/// After move 2: [[0,1,2],[4,5,3]]

/// After move 3: [[1,0,2],[4,5,3]]

/// After move 4: [[1,2,0],[4,5,3]]

/// After move 5: [[1,2,3],[4,5,0]]

///

/// Input: board = [[3,2,4],[1,5,0]]

/// Output: 14

/// Note:

/// board will be a 2 x 3 array as described above.

/// board[i][j] will be a permutation of [0, 1, 2, 3, 4, 5].

/// </summary>

int LeetCode::slidingPuzzle(vector<vector<int>>& board)

{

set<string> visited;

queue<string> search\_queue;

string board\_str;

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

{

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

{

board\_str.push\_back('0' + board[i][j]);

}

}

search\_queue.push(board\_str);

visited.insert(board\_str);

int result = 0;

while (!search\_queue.empty())

{

size\_t size = search\_queue.size();

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

{

string curr\_board = search\_queue.front();

search\_queue.pop();

if (curr\_board == "123450")

{

return result;

}

int pos = curr\_board.find('0');

vector<int> direction = { -1, 1, -3, 3 };

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

{

int next = pos + direction[i];

if ((next < 0) || (next >= (int)curr\_board.size()))

{

continue;

}

// two edge cases

if (pos == 3 && direction[i] == -1) continue;

if (pos == 2 && direction[i] == 1) continue;

string next\_board = curr\_board;

swap(next\_board[pos], next\_board[next]);

if (visited.count(next\_board) == 0)

{

search\_queue.push(next\_board);

visited.insert(next\_board);

}

}

}

result++;

}

return -1;

}

## 815. Bus Routes

Hard

We have a list of bus routes. Each routes[i] is a bus route that the i-th bus repeats forever. For example if routes[0] = [1, 5, 7], this means that the first bus (0-th indexed) travels in the sequence 1->5->7->1->5->7->1->... forever.

We start at bus stop S (initially not on a bus), and we want to go to bus stop T. Travelling by buses only, what is the least number of buses we must take to reach our destination? Return -1 if it is not possible.

**Example:**

**Input:**

routes = [[1, 2, 7], [3, 6, 7]]

S = 1

T = 6

**Output:** 2

**Explanation:**

The best strategy is take the first bus to the bus stop 7, then take the second bus to the bus stop 6.

**Note:**

* 1 <= routes.length <= 500.
* 1 <= routes[i].length <= 500.
* 0 <= routes[i][j] < 10 ^ 6.

### Analysis:

Because we need to get the shortest step, we start the first bus and do BFS search, when we reach the destination, it is by default the shortest stops. During the search, we remember the visited stops.

/// <summary>

/// Leet code #815. Bus Routes

///

/// We have a list of bus routes. Each routes[i] is a bus route that the

/// i-th bus repeats forever. For example if routes[0] = [1, 5, 7], this

/// means that the first bus (0-th indexed) travels in the sequence

/// 1->5->7->1->5->7->1->... forever.

///

/// We start at bus stop S (initially not on a bus), and we want to go to

/// bus stop T. Travelling by buses only, what is the least number of

/// buses we must take to reach our destination? Return -1 if it is not

/// possible.

///

/// Example:

/// Input:

/// routes = [[1, 2, 7], [3, 6, 7]]

/// S = 1

/// T = 6

/// Output: 2

/// Explanation:

/// The best strategy is take the first bus to the bus stop 7, then take

/// the second bus to the bus stop 6.

/// Note:

///

/// 1. 1 <= routes.length <= 500.

/// 2. 1 <= routes[i].length <= 500.

/// 3. 0 <= routes[i][j] < 10 ^ 6.

/// </summary>

int LeetCode::numBusesToDestination(vector<vector<int>>& routes, int S, int T)

{

unordered\_map<int, set<int>> stop\_map;

unordered\_set<int> visited;

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

{

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

{

stop\_map[routes[i][j]].insert(i);

}

}

queue<int> stop\_queue;

stop\_queue.push(S);

visited.insert(S);

int result = 0;

while (!stop\_queue.empty())

{

size\_t size = stop\_queue.size();

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

{

int stop = stop\_queue.front();

stop\_queue.pop();

if (stop == T) return result;

for (int bus : stop\_map[stop])

{

for (int next : routes[bus])

{

if (visited.count(next) > 0)

{

continue;

}

stop\_queue.push(next);

visited.insert(next);

stop\_map[next].erase(bus);

}

}

stop\_map[stop].clear();

}

result++;

}

return -1;

}

## 1765. Map of Highest Peak

Medium

You are given an integer matrix isWater of size m x n that represents a map of **land** and **water** cells.

* If isWater[i][j] == 0, cell (i, j) is a **land** cell.
* If isWater[i][j] == 1, cell (i, j) is a **water** cell.

You must assign each cell a height in a way that follows these rules:

* The height of each cell must be non-negative.
* If the cell is a **water** cell, its height must be 0.
* Any two adjacent cells must have an absolute height difference of **at most** 1. A cell is adjacent to another cell if the former is directly north, east, south, or west of the latter (i.e., their sides are touching).

Find an assignment of heights such that the maximum height in the matrix is **maximized**.

Return *an integer matrix*height*of size*m x n*where*height[i][j]*is cell*(i, j)*'s height. If there are multiple solutions, return****any****of them*.

**Example 1:**

**A picture containing text, clock

Description automatically generated**

**Input:** isWater = [[0,1],[0,0]]

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

**Explanation:** The image shows the assigned heights of each cell.

The blue cell is the water cell, and the green cells are the land cells.

**Example 2:**

**Table

Description automatically generated with medium confidence**

**Input:** isWater = [[0,0,1],[1,0,0],[0,0,0]]

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

**Explanation:** A height of 2 is the maximum possible height of any assignment.

Any height assignment that has a maximum height of 2 while still meeting the rules will also be accepted.

**Constraints:**

* m == isWater.length
* n == isWater[i].length
* 1 <= m, n <= 1000
* isWater[i][j] is 0 or 1.
* There is at least **one** water cell.

### Analysis:

This is a google problem. We start from water and use BFS to find its neighbor land then get more neighbors. If the land heights not marked, we will mark it as neighbor height + 1. Eventually we will mark all nodes. Because in this problem we set the original height as an impossible value INT\_MAX (you can do -1 as well), then we do not need the visited matrix to remember which positions are visited, i.e. any position without specific height are unvisited.

/// <summary>

/// Leet code 1765. Map of Highest Peak

///

/// Medium

///

/// You are given an integer matrix isWater of size m x n that represents

/// a map of land and water cells.

///

/// If isWater[i][j] == 0, cell (i, j) is a land cell.

/// If isWater[i][j] == 1, cell (i, j) is a water cell.

/// You must assign each cell a height in a way that follows these rules:

///

/// The height of each cell must be non-negative.

/// If the cell is a water cell, its height must be 0.

/// Any two adjacent cells must have an absolute height difference of at

/// most 1. A cell is adjacent to another cell if the former is directly

/// north, east, south, or west of the latter (i.e., their sides are

/// touching).

/// Find an assignment of heights such that the maximum height in the

/// matrix is maximized.

///

/// Return an integer matrix height of size m x n where height[i][j] is

/// cell (i, j)'s height. If there are multiple solutions, return any of

/// them.

///

/// Example 1:

/// Input: isWater = [[0,1],[0,0]]

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

/// Explanation: The image shows the assigned heights of each cell.

/// The blue cell is the water cell, and the green cells are the land

/// cells.

///

/// Example 2:

/// Input: isWater = [[0,0,1],[1,0,0],[0,0,0]]

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

/// Explanation: A height of 2 is the maximum possible height of any

/// assignment.

/// Any height assignment that has a maximum height of 2 while still meeting

/// the rules will also be accepted.

///

/// Constraints:

/// 1. m == isWater.length

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

/// 3. 1 <= m, n <= 1000

/// 4. isWater[i][j] is 0 or 1.

/// 5. There is at least one water cell.

/// </summary>

vector<vector<int>> LeetCodeGraph::highestPeak(vector<vector<int>>& isWater)

{

int n = isWater.size();

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

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

queue<pair<int, int>> queue;

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

{

for (int j = 0; j < m; j++)

{

if (isWater[i][j] == 1)

{

result[i][j] = 0;

queue.push(make\_pair(i, j));

}

}

}

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

while (!queue.empty())

{

pair<int, int> pos = queue.front();

queue.pop();

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

{

pair<int, int> next = pos;

next.first += directions[i][0];

next.second += directions[i][1];

if (next.first < 0 || next.first >= n || next.second < 0 || next.second >= m)

{

continue;

}

if (result[next.first][next.second] != INT\_MAX) continue;

result[next.first][next.second] = result[pos.first][pos.second] + 1;

queue.push(next);

}

}

return result;

}

## 994. Rotting Oranges

Medium

You are given an m x n grid where each cell can have one of three values:

* 0 representing an empty cell,
* 1 representing a fresh orange, or
* 2 representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return *the minimum number of minutes that must elapse until no cell has a fresh orange*. If *this is impossible, return* -1.

**Example 1:**

A picture containing scatter chart

Description automatically generated

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

**Output:** 4

**Example 2:**

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

**Output:** -1

**Explanation:** The orange in the bottom left corner (row 2, column 0) is never rotten, because rotting only happens 4-directionally.

**Example 3:**

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

**Output:** 0

**Explanation:** Since there are already no fresh oranges at minute 0, the answer is just 0.

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 10
* grid[i][j] is 0, 1, or 2.

### Analysis:

Starting from rotten oranges, walk 4 directions to get more oranges to rotten. At very beginning push all the rotten oranges to queue and count all fresh oranges.

/// <summary>

/// Leet code #994. Rotting Oranges

///

/// In a given grid, each cell can have one of three values:

///

/// the value 0 representing an empty cell;

/// the value 1 representing a fresh orange;

/// the value 2 representing a rotten orange.

/// Every minute, any fresh orange that is adjacent (4-directionally) to a

/// rotten orange becomes rotten.

///

/// Return the minimum number of minutes that must elapse until no cell has

/// a fresh orange. If this is impossible, return -1 instead.

///

/// Example 1:

///

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

/// Output: 4

///

/// Example 2:

///

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

/// Output: -1

/// Explanation: The orange in the bottom left corner (row 2, column 0)

/// is never rotten, because rotting only happens 4-directionally.

///

/// Example 3:

///

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

/// Output: 0

/// Explanation: Since there are already no fresh oranges at minute 0, the

/// answer is just 0.

///

///

/// Note:

///

/// 1. 1 <= grid.length <= 10

/// 2. 1 <= grid[0].length <= 10

/// 3. grid[i][j] is only 0, 1, or 2.

/// </summary>

int LeetCodeBFS::orangesRotting(vector<vector<int>>& grid)

{

queue<pair<int, int>> search;

int count = 0;

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

{

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

{

if (grid[i][j] == 2)

{

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

}

else if (grid[i][j] == 1)

{

count++;

}

}

}

int result = 0;

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

while (!search.empty())

{

size\_t size = search.size();

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

{

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

search.pop();

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

{

pair<int, int> next\_pos = pos;

next\_pos.first += directions[d][0];

next\_pos.second += directions[d][1];

if (next\_pos.first < 0 || next\_pos.first >= (int)grid.size() ||

next\_pos.second < 0 || next\_pos.second >= (int)grid[0].size())

{

continue;

}

if (grid[next\_pos.first][next\_pos.second] == 1)

{

count--;

search.push(make\_pair(next\_pos.first, next\_pos.second));

grid[next\_pos.first][next\_pos.second] = 2;

}

}

}

if (!search.empty()) result++;

}

if (count != 0) result = -1;

return result;

}

## 286. Walls and Gates

Medium

You are given an m x n grid rooms initialized with these three possible values.

* -1 A wall or an obstacle.
* 0 A gate.
* INF Infinity means an empty room. We use the value 231 - 1 = 2147483647 to represent INF as you may assume that the distance to a gate is less than 2147483647.

Fill each empty room with the distance to *its nearest gate*. If it is impossible to reach a gate, it should be filled with INF.

**Example 1:**

A picture containing text, crossword puzzle

Description automatically generated

**Input:** rooms = [[2147483647,-1,0,2147483647],[2147483647,2147483647,2147483647,-1],[2147483647,-1,2147483647,-1],[0,-1,2147483647,2147483647]]

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

**Example 2:**

**Input:** rooms = [[-1]]

**Output:** [[-1]]

**Constraints:**

* m == rooms.length
* n == rooms[i].length
* 1 <= m, n <= 250
* rooms[i][j] is -1, 0, or 231 - 1.

### Analysis:

Starting from gate and mark every empty room distance.

/// <summary>

/// Leet Code 286. Walls and Gates

///

/// Medium

///

/// You are given an m x n grid rooms initialized with these three

/// possible values.

///

/// -1 A wall or an obstacle.

/// 0 A gate.

/// INF Infinity means an empty room. We use the value

/// 2^31 - 1 = 2147483647 to represent INF as you may assume that the

/// distance to a gate is less than 2147483647.

/// Fill each empty room with the distance to its nearest gate. If it

/// is impossible to reach a gate, it should be filled with INF.

///

/// Example 1:

/// Input: rooms = [[2147483647,-1,0,2147483647],

/// [2147483647,2147483647,2147483647,-1], [2147483647,-1,2147483647,-1],

/// [0,-1,2147483647,2147483647]]

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

///

/// Example 2:

/// Input: rooms = [[-1]]

/// Output: [[-1]]

///

/// Constraints:

/// 1. m == rooms.length

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

/// 3. 1 <= m, n <= 250

/// 4. rooms[i][j] is -1, 0, or 2^31 - 1.

/// </summary>

void LeetCodeGraph::wallsAndGates(vector<vector<int>>& rooms)

{

int n = rooms.size();

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

queue<vector<int>> process\_queue;

for (int i = 0; i < (int)rooms.size(); i++)

{

for (int j = 0; j < (int)rooms[i].size(); j++)

{

if (rooms[i][j] == 0)

{

process\_queue.push({ i, j });

}

}

}

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

while (!process\_queue.empty())

{

vector<int> pos = process\_queue.front();

process\_queue.pop();

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

{

vector<int> next = pos;

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

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

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

{

continue;

}

if (rooms[next[0]][next[1]] == -1 || rooms[next[0]][next[1]] != INT\_MAX)

{

continue;

}

rooms[next[0]][next[1]] = rooms[pos[0]][pos[1]] + 1;

process\_queue.push(next);

}

}

}

### Analysis:

For this problem you can also do DFS, just make sure check the gate position.

/// <summary>

/// Leet code #286. Walls and Gates

/// </summary>

void LeetCodeGraph::wallsAndGatesDFS(vector<vector<int>>& rooms, int row, int col, int distance)

{

if (row < 0 || col < 0 || row >= (int)rooms.size() || col >= (int)rooms[0].size())

{

return;

}

if (rooms[row][col] <= distance && distance != 0)

{

return;

}

rooms[row][col] = distance;

wallsAndGatesDFS(rooms, row - 1, col, distance + 1);

wallsAndGatesDFS(rooms, row, col - 1, distance + 1);

wallsAndGatesDFS(rooms, row + 1, col, distance + 1);

wallsAndGatesDFS(rooms, row, col + 1, distance + 1);

}

/// <summary>

/// Leet code #286. Walls and Gates

///

/// You are given a m x n 2D grid initialized with these three possible values.

/// 1.-1 - A wall or an obstacle.

/// 2.0 - A gate.

/// 3.INF - Infinity means an empty room. We use the value 231 - 1 = 2147483647 to represent

/// INF as you may assume that the distance to a gate is less than 2147483647.

/// Fill each empty room with the distance to its nearest gate. If it is impossible to reach a gate, it should be filled with INF.

/// For example, given the 2D grid:

/// INF -1 0 INF

/// INF INF INF -1

/// INF -1 INF -1

/// 0 -1 INF INF

/// After running your function, the 2D grid should be:

/// 3 -1 0 1

/// 2 2 1 -1

/// 1 -1 2 -1

/// 0 -1 3 4

/// </summary>

void LeetCodeGraph::wallsAndGatesDFS(vector<vector<int>>& rooms)

{

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

{

for (size\_t j = 0; j < rooms[0].size(); j++)

{

if (rooms[i][j] == 0)

{

wallsAndGatesDFS(rooms, i, j, 0);

}

}

}

}