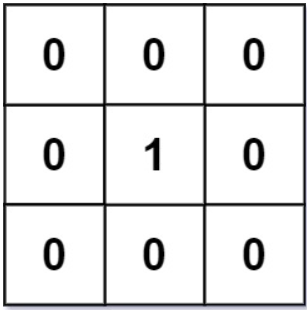
<https://leetcode.com/problems/01-matrix/>

Given an m x n binary matrix mat, return *the distance of the nearest* 0 *for each cell*.

The distance between two adjacent cells is 1.

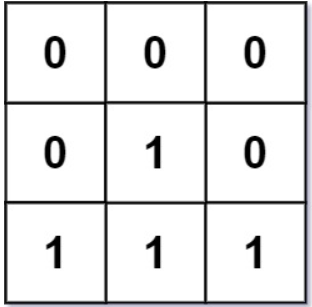
**Example 1:**



Input: mat = [[0,0,0],[0,1,0],[0,0,0]]

Output: [[0,0,0],[0,1,0],[0,0,0]]

**Example 2:**



Input: mat = [[0,0,0],[0,1,0],[1,1,1]]

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

**Constraints:**

* m == mat.length
* n == mat[i].length
* 1 <= m, n <= 104
* 1 <= m \* n <= 104
* mat[i][j] is either 0 or 1.
* There is at least one 0 in mat.

**Attempt 1: 2023-08-18**

**Solution 1: BFS + Topological pattern (30 min)**

class Solution {

public int[][] updateMatrix(int[][] mat) {

Queue<int[]> q = new LinkedList<int[]>();

int m = mat.length;

int n = mat[0].length;

for(int i = 0; i < m; i++) {

for(int j = 0; j < n; j++) {

if(mat[i][j] == 0) {

q.offer(new int[]{i, j});

} else {

// Change original 1 to -1 in case of distance for a cell might be set to 1

mat[i][j] = -1;

}

}

}

// Similar as Topological Sort pattern, start from all mat[i][j] = 0

// cells as "center"(s) and level by level gradually expand footprint

// on four directions of these "center"(s), if find a mat[i][j] = -1

// means we find a target cell originally not 0, update distance on

// this cell {i,j}, then consider the new cell {i,j} as new "center"

// which going to expand footprint on four directions again to find

// futher mat[i][j] = -1, and to implement this logic, we add new

// "center" into queue, after adding all new "center"(s) into queue,

// it consider as a new level, and correspondingly when start new level

// scanning around new "center"(s) we have to increase distance by 1

// to mark the potential mat[i][j] = -1 cell around new "center"(s)

int[] dx = new int[]{0,0,1,-1};

int[] dy = new int[]{1,-1,0,0};

int distance = 0;

while(!q.isEmpty()) {

distance++;

int size = q.size();

for(int i = 0; i < size; i++) {

int[] cur = q.poll();

for(int k = 0; k < 4; k++) {

int new\_x = cur[0] + dx[k];

int new\_y = cur[1] + dy[k];

if(new\_x >= 0 && new\_x < m && new\_y >= 0 && new\_y < n

&& mat[new\_x][new\_y] == -1) {

mat[new\_x][new\_y] = distance;

q.offer(new int[]{new\_x, new\_y});

}

}

}

}

return mat;

}

}

Time Complexity: O(M \* N), where M is number of rows, N is number of columns in the matrix.

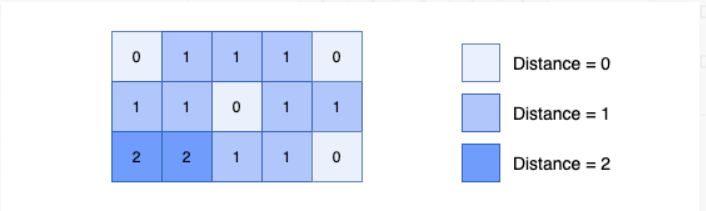
Space Complexity: O(M \* N), space for the queue.

**Refer to**

<https://leetcode.com/problems/01-matrix/solutions/2583027/dfs-vs-bfs-which-is-better-when-to-use-them/>

**Solution 1: BFS on zero cells first**

* For convenience, let's call the cell with value 0 as **zero-cell**, the cell has with value 1 as **one-cell**, the distance of the nearest 0 of a cell as **distance**.
* Firstly, we can see that the distance of all **zero-cells** are 0.
* Same idea with Topology Sort, we process **zero-cells** first, then we use queue data structure to keep the order of processing cells, so that cells which have the smaller distance will be processed first. Then we expand the unprocessed neighbors of the current processing cell and push into our queue.
* Afterall, we can achieve the minimum distance of all cells in our matrix.



class Solution {

int[] DIR = new int[]{0, 1, 0, -1, 0};

public int[][] updateMatrix(int[][] mat) {

int m = mat.length, n = mat[0].length; // The distance of cells is up to (M+N)

Queue<int[]> q = new ArrayDeque<>();

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

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

if (mat[r][c] == 0) q.offer(new int[]{r, c});

else mat[r][c] = -1; // Marked as not processed yet!

while (!q.isEmpty()) {

int[] curr = q.poll();

int r = curr[0], c = curr[1];

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

int nr = r + DIR[i], nc = c + DIR[i+1];

if (nr < 0 || nr == m || nc < 0 || nc == n || mat[nr][nc] != -1) continue;

mat[nr][nc] = mat[r][c] + 1;

q.offer(new int[]{nr, nc});

}

}

return mat;

}

}

**Complexity**

* Time: O(M \* N), where M is number of rows, N is number of columns in the matrix.
* Space: O(M \* N), space for the queue.

**Solution 2: Traverse on two ways (30 min)**

class Solution {

public int[][] updateMatrix(int[][] mat) {

int m = mat.length;

int n = mat[0].length;

int max = m + n;

// Inplace update distance from top to bottom + from left to right on matrix

for(int i = 0; i < m; i++) {

for(int j = 0; j < n; j++) {

if(mat[i][j] == 0) {

continue;

} else {

// Handling first row and first column with default value 'max'

int top\_distance = max;

int left\_distance = max;

// Otherwise the top\_distance always coming from previous row, the

// left\_distance always coming from previous column

if(i > 0) {

top\_distance = mat[i - 1][j];

}

if(j > 0) {

left\_distance = mat[i][j - 1];

}

// Compare top\_distance and left\_distance to get the minimum value

// and also locally update matrix itself without create new 2D array

mat[i][j] = Math.min(top\_distance, left\_distance) + 1;

}

}

}

// Inplace update distance from bottom to top + from right to left on matrix

for(int i = m - 1; i >= 0; i--) {

for(int j = n - 1; j >= 0; j--) {

if(mat[i][j] == 0) {

continue;

} else {

int bottom\_distance = max;

int right\_distance = max;

if(i < m - 1) {

bottom\_distance = mat[i + 1][j];

}

if(j < n - 1) {

right\_distance = mat[i][j + 1];

}

mat[i][j] = Math.min(mat[i][j], Math.min(bottom\_distance, right\_distance) + 1);

}

}

}

return mat;

}

}

Time Complexity: O(M \* N), where M is number of rows, N is number of columns in the matrix.

Space Complexity: O(1)

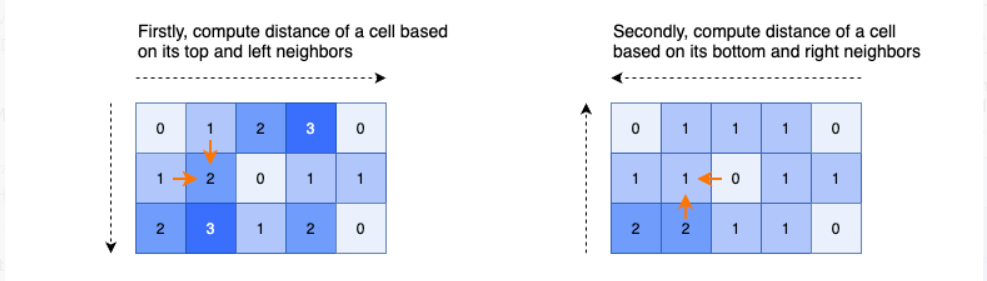
**Refer to**

<https://leetcode.com/problems/01-matrix/solutions/1369741/c-java-python-bfs-dp-solutions-with-picture-clean-concise-o-1-space/>

**✔️ Solution 2: Dynamic Programming**

* For convenience, let's call the cell with value 0 as **zero-cell**, the cell has with value 1 as **one-cell**, the distance of the nearest 0 of a cell as **distance**.
* Firstly, we can see that the distance of all **zero-cells** are 0, so we skip **zero-cells**, we process **one-cells** only.
* In DP, we can only use previous values if they're already computed.
* In this problem, a cell has at most 4 neighbors that are left, top, right, bottom. If we use dynamic programming to compute the distance of the current cell based on 4 neighbors simultaneously, it's impossible because we are not sure if distance of neighboring cells is already computed or not.
* That's why, we need to compute the distance one by one:

1. Firstly, for a cell, we restrict it to only 2 directions which are left and top. Then we iterate cells from **top to bottom**, and from **left to right**, we calculate the distance of a cell **based on its left and top neighbors**.
2. Secondly, for a cell, we restrict it only have 2 directions which are right and bottom. Then we iterate cells from **bottom to top**, and from **right to left**, we update the distance of a cell **based on its right and bottom neighbors**.



class Solution { // 5 ms, faster than 99.66%

public int[][] updateMatrix(int[][] mat) {

int m = mat.length, n = mat[0].length, INF = m + n; // The distance of cells is up to (M+N)

for (int r = 0; r < m; r++) {

for (int c = 0; c < n; c++) {

if (mat[r][c] == 0) continue;

int top = INF, left = INF;

if (r - 1 >= 0) top = mat[r - 1][c];

if (c - 1 >= 0) left = mat[r][c - 1];

mat[r][c] = Math.min(top, left) + 1;

}

}

for (int r = m - 1; r >= 0; r--) {

for (int c = n - 1; c >= 0; c--) {

if (mat[r][c] == 0) continue;

int bottom = INF, right = INF;

if (r + 1 < m) bottom = mat[r + 1][c];

if (c + 1 < n) right = mat[r][c + 1];

mat[r][c] = Math.min(mat[r][c], Math.min(bottom, right) + 1);

}

}

return mat;

}

}

**Complexity**

* Time: O(M \* N), where M is number of rows, N is number of columns in the matrix.
* Space: O(1)

**DFS vs BFS | Which is better | when to use them**

**Refer to**

<https://leetcode.com/problems/01-matrix/solutions/2583027/dfs-vs-bfs-which-is-better-when-to-use-them/>

Initially this problem seems to be very similar to all the backtracking problems that you would've solved earlier like(rat in a maze, N Queen problem) . Which made me think the same and i ended up getting the TLE for the DFS solution and then i tried applying DP but that is a mess too. Here are some of the points that i learnt from this question:

* The previous DFS + Backtracking problems we've always had the definite start and endpoint so it doesn't really matter to end the recursive all . But where as here the starting points and the ending points are scattered which may consume a lot of time if we apply a DFS solution.
* DFS will come in to action , if you have a choice and you decide to pick that up and try to build a solution using that choice and finally find the best of all the choices you made
* BFS works differently, where it finds the shorted node and the next shortest node to these visited nodes and so on... and eventually you reach a solution which will give you the shortest path.

**Verdict:**

**Whenever you see find shortest, minimum, nearest , quickest way Throw BFS blindly you got to break your head only to find , which will be my starting point(0 here) and which will be my desired ending point(1 here)**