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.

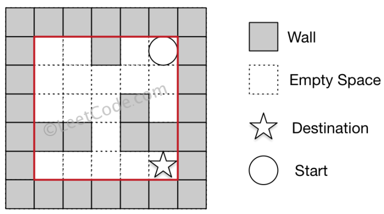
Example 1

```
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.
```



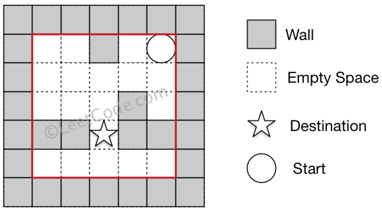
Example 2

```
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.

Solution 1

Solution of *The Maze*: https://discuss.leetcode.com/topic/77471/easy-understanding-java-bfs-solution
Solution of *The Maze III*: https://discuss.leetcode.com/topic/77474/similar-to-the-maze-ii-easy-understanding-java-bfs-solution

We need to use `PriorityQueue` instead of standard queue, and record the minimal length of each point.

```
public class Solution {
    class Point {
        int x,y,l;
        public Point(int _x, int _y, int _l) {x=_x;y=_y;l=_l;}
    }
    public int shortestDistance(int[][] maze, int[] start, int[] destination) {
        int m=maze.length, n=maze[0].length;
        int[][] length=new int[m][n]; // record length
        for (int i=0;i<m*n;i++) length[i/n][i%n]=Integer.MAX_VALUE;
        int[][] dir=new int[][] {{-1,0},{0,1},{1,0},{0,-1}};
        PriorityQueue<Point> list=new PriorityQueue<>((o1,o2)->o1.l-o2.l); // using priority queue
        list.offer(new Point(start[0], start[1], 0));
        while (!list.isEmpty()) {
            Point p=list.poll();
            if (length[p.x][p.y]<=p.l) continue; // if we have already found a route shorter
            length[p.x][p.y]=p.l;
            for (int i=0;i<4;i++) {
                int xx=p.x, yy=p.y, l=p.l;
                while (xx>=0 && xx<m && yy>=0 && yy<n && maze[xx][yy]==0) {
                    xx+=dir[i][0];
                    yy+=dir[i][1];
                    l++;
                }
                xx-=dir[i][0];
                yy-=dir[i][1];
                l--;
                list.offer(new Point(xx, yy, l));
            }
        }
        return length[destination[0]][destination[1]]==Integer.MAX_VALUE?-1:length[destination[0]][destination[1]];
    }
}
```

written by ckcz123 original link here

Solution 2

My BFS using queue. Use matrix to store the distance. If smaller distance is found, need to process the position[i,j] one more time.

```
class Solution {
public:
    int shortestDistance(vector<vector<int>>& maze, vector<int>& start, vector<int>& destination) {
 int m = maze.size();
 int n = maze[0].size();
 int minDist = INT_MAX;

 vector<vector<int>> dists(m, vector<int>(n, -1));
 queue<pair<int, int>> q;

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

 q.push({ start[0], start[1] });
 dists[start[0]][start[1]] = 0;

 while (!q.empty())
 {
  auto curr = q.front();
  q.pop();
  int x = curr.first;
  int y = curr.second;
  int dist = dists[x][y];

  for (int k = 0; k < 4; ++k)
  {
   int i = x;
   int j = y;
   int step = 0;
   int d_i = incr[k].first;
   int d_j = incr[k].second;
   int tempMin = INT_MAX;

   while (i + d_i < m && i + d_i >= 0 && j + d_j >= 0 && j + d_j <n && maze[i + d_i][j + d_j] == 0)
   {
    ++step;
    i += incr[k].first;
    j += incr[k].second;

   }
   if (dists[i][j] == -1) // visited first time
   {
    dists[i][j] = dist + step;
    q.push({ i,j });
   }
   else
   {
    if (dists[i][j] > dist + step) // not the first time, but generate smaller dist, process one more time
    {
```

```
      dists[i][j] = dist + step;
      q.push({ i,j });
     }
    }
   }
  }

  return dists[destination[0]][destination[1]];
     }
};
```

written by xiapeggy original link here

Solution 3

The idea is similar to the Dijkstra's shortest Path Algorithm.

```
public class Solution {
    private int[][] maze;
    private int[][] minSteps;//memorize the minimus steps to reach each position
    private int[][] dirs={{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    public int shortestDistance(int[][] maze, int[] start, int[] destination) {
        this.maze=maze;
        this.minSteps=new int[maze.length][maze[0].length];
        /*Initiallize minSteps matrix*/
        for(int i=0; i<maze.length; i++){
            for(int j=0; j<maze[0].length; j++){
                minSteps[i][j]=Integer.MAX_VALUE;
            }
        }
        /*Optimization: check if the destination is impossible to Reach*/
        boolean desL=canRoll(destination[0], destination[1], dirs[0]);
        boolean desR=canRoll(destination[0], destination[1], dirs[1]);
        boolean desD=canRoll(destination[0], destination[1], dirs[2]);
        boolean desU=canRoll(destination[0], destination[1], dirs[3]);
        if(desL && desR && desD && desU) return -1; //all neighbors are walls
        else if(!(desL||desR||desD||desU)) return -1; //all neighbors are empty spaces
        else if(desL && desR && !desU && !desD) return -1; //two opposite neigbhors are walls, and the other two are empty spaces
        else if(!desL && !desR && desU && desD) return -1;//two opposite neigbhors are walls, and the other two are empty spaces

        minSteps[start[0]][start[1]]=0;
        /*BFS; Optimization: use PriorityQueue based on the steps instead of Queue*/
        PriorityQueue<Position> pq=new PriorityQueue<>();
        pq.offer(new Position(start[0], start[1], 0));
        while(!pq.isEmpty()){
            Position pos=pq.poll();
            /*optimization: if the destination is at the head of the queue, we are done*/
            if(pos.r==destination[0] && pos.c==destination[1]) return pos.steps;
            for(int[] dir: dirs){
                int r=pos.r, c=pos.c, currSteps=0;
                while(canRoll(r, c, dir)){
                    r+=dir[0];
                    c+=dir[1];
                    currSteps++;
                }
                int totalSteps=pos.steps+currSteps;
                if(totalSteps<minSteps[r][c] && totalSteps<minSteps[destination[0]][destination[1]]){
                    minSteps[r][c]=totalSteps;
                    pq.offer(new Position(r, c, totalSteps));
                }
            }
        }
        /*May not be able to reach the destination*/
```

```java
        /*May not be able to reach the destination*/
        return minSteps[destination[0]][destination[1]]==Integer.MAX_VALUE? -1: m
inSteps[destination[0]][destination[1]];
    }

    private boolean canRoll(int r, int c, int[] dir){
        r+=dir[0];
        c+=dir[1];
        if(r<0 || c<0 || r>=maze.length || c>=maze[0].length || maze[r][c]==1) re
turn false;
        return true;
    }
}

class Position implements Comparable<Position>{
    public int r;
    public int c;
    public int steps;

    public Position(int r, int c, int s){
        this.r=r;
        this.c=c;
        this.steps=s;
    }
    @Override
    public int compareTo(Position other){
        return this.steps-other.steps;
    }
}
```

written by ydeng original link here