Weekly contest 362

2849. Determine if a Cell Is Reachable at a Given Time
Medium

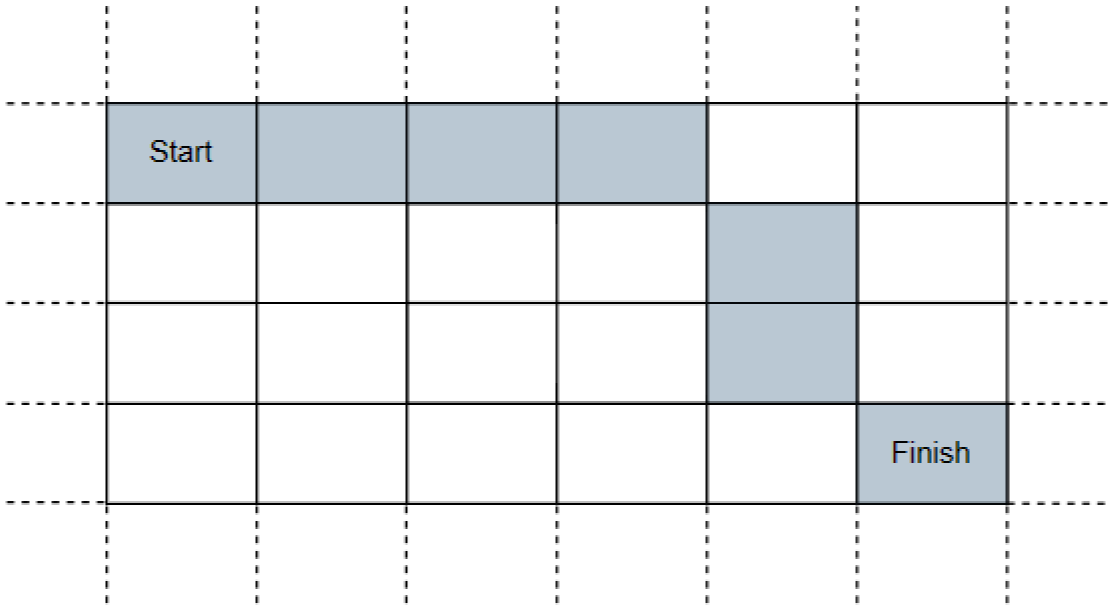
You are given four integers sx, sy, fx, fy, and a non-negative integer t.

In an infinite 2D grid, you start at the cell (sx, sy). Each second, you must move to any of its adjacent cells.

Return true *if you can reach cell* (fx, fy) *after exactly* t *seconds*, *or* false *otherwise*.

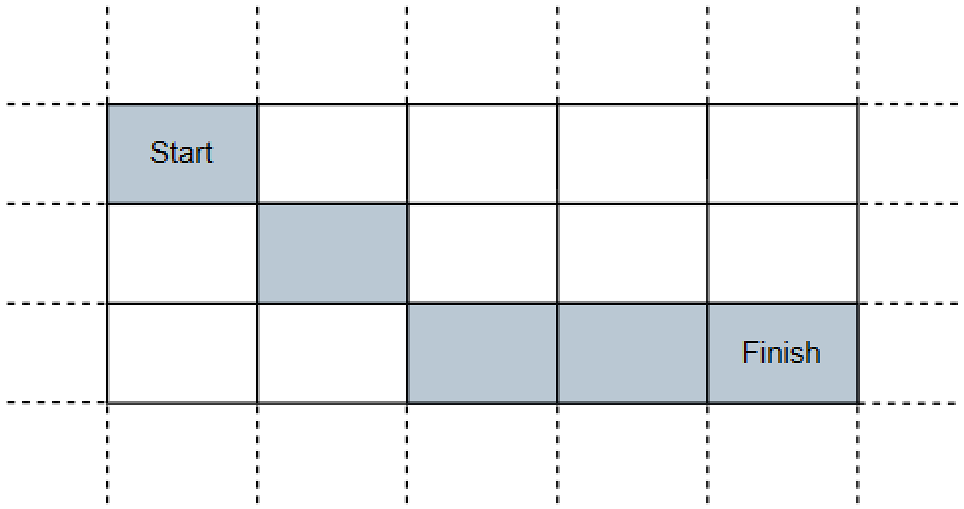
A cell's adjacent cells are the 8 cells around it that share at least one corner with it. You can visit the same cell several times.

Example 1:



Input: sx = 2, sy = 4, fx = 7, fy = 7, t = 6
Output: true
Explanation: Starting at cell (2, 4), we can reach cell (7, 7) in exactly 6 seconds by going through the cells depicted in the picture above.

Example 2:



Input: sx = 3, sy = 1, fx = 7, fy = 3, t = 3
Output: false
Explanation: Starting at cell (3, 1), it takes at least 4 seconds to reach cell (7, 3) by going through the cells depicted in the picture above. Hence, we cannot reach cell (7, 3) at the third second.

Constraints:

- **$1 <= sx, sy, fx, fy <= 10_9$**
- **$0 <= t <= 10_9$**

CODE-

```
class Solution {

public:

    bool isReachableAtTime(int sx, int sy, int fx, int fy, int t) {

        int n = abs(fx - sx);

        int m = abs(fy - sy);

        int minTime = 0;

        if (n < m) {

            minTime = n + abs(n - m);
        } else {
            minTime = m + abs(n - m);
        }
        if (minTime == 0) return t != 1;
        return minTime <= t;
    }
};
```

This code defines a C++ function `isReachableAtTime` that takes five integer parameters: `sx` (starting x-coordinate), `sy` (starting y-coordinate), `fx` (ending x-coordinate), `fy` (ending y-coordinate), and `t` (time in seconds).

Here's a step-by-step explanation of the code:

Calculate the absolute difference between the x-coordinates (`fx - sx`) and store it in the variable `n`. This represents the horizontal distance between the starting and ending points.
Calculate the absolute difference between the y-coordinates (`fy - sy`) and store it in the variable `m`. This represents the vertical distance between the starting and ending points.
Initialize the variable `minTime` to 0. This variable will be used to calculate the minimum time required to reach the ending point.
Determine whether the horizontal distance (`n`) is less than the vertical distance (`m`). If `n` is less than `m`, it means that it's more efficient to first move vertically (`n` units) and then move horizontally (`m` units). In this case, `minTime` is calculated as follows:

- `minTime = n + abs(n - m)`: This computes the time required to move `n` units vertically and then move `abs(n - m)` units horizontally.

If the horizontal distance (`n`) is greater than or equal to the vertical distance (`m`), it means that it's more efficient to first move horizontally (`m` units) and then move vertically (`n` units). In this case, `minTime` is calculated as follows:

- `minTime = m + abs(n - m)`: This computes the time required to move `m` units horizontally and then move `abs(n - m)` units vertically.

Check if `minTime` is equal to 0. If `minTime` is 0, it means that the starting and ending points are the same (i.e., `sx == fx` and `sy == fy`). In this case, check if `t` is not equal to 1 (i.e., `t != 1`). If `t` is not 1, it's possible to stay at the same point for multiple seconds, so return `true`. Otherwise, return `false` because there's no need to move if the starting and ending points are the same.
Finally, return `true` if `minTime` is less than or equal to `t`. This checks whether it's possible to reach the ending point within `t` seconds based on the minimum time calculated in steps 4 or 5.

In summary, the code calculates the minimum time required to reach the ending point considering both horizontal and vertical movements and then checks if it's possible to reach the ending point within the given time `t`.

OR

```
bool isReachableAtTime(int sx, int sy, int fx, int fy, int t) {

   int xdiff = abs(sx - fx), ydiff = abs(sy - fy);

   if (xdiff == 0 && ydiff == 0 && t == 1) return false;

   return (min(xdiff, ydiff) + abs(xdiff - ydiff)) <= t;

}
```

Here's an explanation of the code:

Calculate the absolute differences between the starting x-coordinate `sx` and the ending x-coordinate `fx`, as well as the starting y-coordinate `sy` and the ending y-coordinate `fy`. Store these differences in the variables `xdiff` and `ydiff`.
Check if both `xdiff` and `ydiff` are zero (i.e., `xdiff == 0` and `ydiff == 0`) and `t` is equal to 1 (i.e., `t == 1`). This condition checks if the starting and ending points are the same, and `t` is exactly 1. If this condition is met, return `false` because it's not possible to move to the same point in one second.
Calculate the minimum of `xdiff` and `ydiff` using the `min` function, and add the absolute difference between `xdiff` and `ydiff` to it. This step calculates the minimum time required to reach the ending point by considering both horizontal and vertical movements.

Check if the result of step 3 is less than or equal to `t`. If the calculated minimum time is less than or equal to `t`, return `true`, indicating that it's possible to reach the ending point within the given time `t`. Otherwise, return `false`.

In summary, this code calculates the minimum time required to reach the ending point while considering both horizontal and vertical movements. It then checks if this minimum time is within the given time constraint `t` and returns the appropriate boolean value. If the starting and ending points are the same and `t` is exactly 1, it returns `false` because it's not possible to move in one second to the same point where you already are.

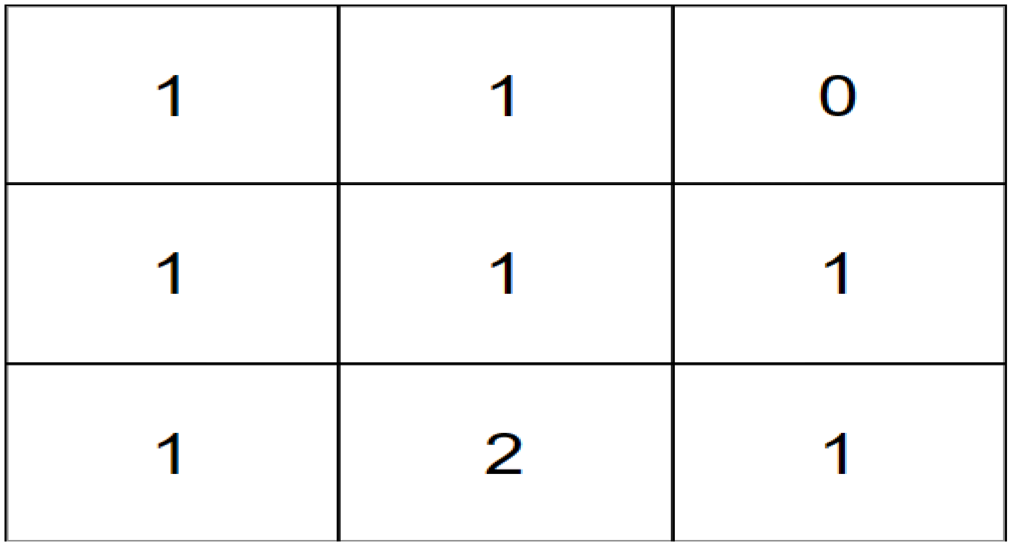
[2850. Minimum Moves to Spread Stones Over Grid]

You are given a **0-indexed** 2D integer matrix grid of size 3 * 3, representing the number of stones in each cell. The grid contains exactly 9 stones, and there can be **multiple** stones in a single cell.

In one move, you can move a single stone from its current cell to any other cell if the two cells share a side.

Return *the **minimum number of moves** required to place one stone in each cell*.

Example 1:



1	1	0
1	1	1
1	2	1

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

Output: 3

Explanation: One possible sequence of moves to place one stone in each cell is:

1- Move one stone from cell (2,1) to cell (2,2).

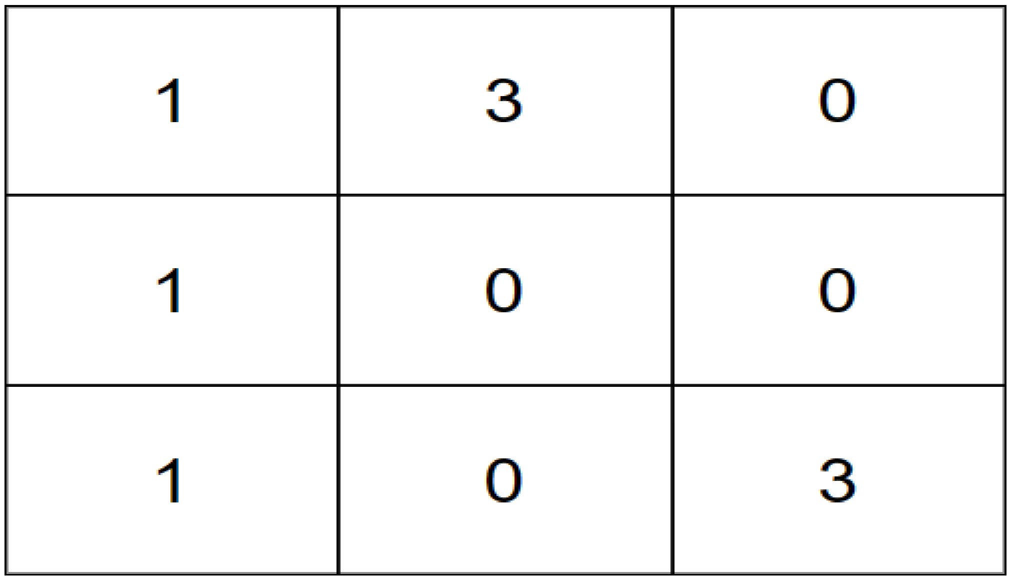
2- Move one stone from cell (2,2) to cell (1,2).

3- Move one stone from cell (1,2) to cell (0,2).

In total, it takes 3 moves to place one stone in each cell of the grid.

It can be shown that 3 is the minimum number of moves required to place one stone in each cell.

Example 2:



1	3	0
1	0	0
1	0	3

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

Output: 4

Explanation: One possible sequence of moves to place one stone in each cell is:

1- Move one stone from cell (0,1) to cell (0,2).

2- Move one stone from cell (0,1) to cell (1,1).

3- Move one stone from cell (2,2) to cell (1,2).

```
4- Move one stone from cell (2,2) to cell (2,1).

In total, it takes 4 moves to place one stone in each cell of the grid.

It can be shown that 4 is the minimum number of moves required to place one stone in each cell.
```

Constraints:

- grid.length == grid[i].length == 3
- 0 <= grid[i][j] <= 9
- Sum of grid is equal to 9.

Intuition

Since the constraints are very less we can apply recursion here.

Approach

Each time we encounter a cell with grid[i][j] == 0 then we can take 1 from any of the other 8 cells which have a value > 1

Complexity

- Time complexity:
The overall time complexity of the code is O(3^(n^2))

- Space complexity:
The space complexity is O(D), where D is the maximum depth of the recursion

Code (C++)

```
class Solution {

public:

    int minimumMoves(vector<vector<int>>& grid) {

        // Base Case
```

```
int t = 0;

for (int i = 0; i < 3; ++i)

    for (int j = 0; j < 3; ++j)

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

            t++;

if (t == 0)

    return 0;


int ans = INT_MAX;

for (int i = 0; i < 3; ++i)

{

    for (int j = 0; j < 3; ++j)

    {

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

        {

            for (int ni = 0; ni < 3; ++ni)

            {

                for (int nj = 0; nj < 3; ++nj)

                {

                    int d = abs(ni - i) + abs(nj - j);

                    if (grid[ni][nj] > 1)

                    {

                        grid[ni][nj]--;

                        grid[i][j]++;

                        ans = min(ans, d + minimumMoves(grid));

                        grid[ni][nj]++;

                        grid[i][j]--;

                    }
```

```
                        }
                    }
                }
            }
        }
        return ans;
    }
};
```

```
```