CMIMC Programming 2022

Problem Packet

Help I’m stuck in an IKEA and my only contact with the outside world is this problem title

The hardest thing of all is to find a black cat in a dark room,
especially if there is no cat.
- Confucius

As lead developer of the delivery company Amazin, you have launched a new program where drones can buy furniture for people. As a prototype, you have deployed a robot in a large IKEA store, which conveniently has the layout of a 64x64 grid, where each cell in the grid is either a floor (can be traveled on), or a piece of furniture (cannot be traveled on). As an unfortunate consequence of the latest interior design trends, it turns out that all pieces of furniture at IKEA are 4-way symmetric and indistinguishable from one another.

At any point in time, your robot is facing one of four axial directions (up, down, left, right), and on each step, it may move forwards, right, left, or backwards by one cell. The robot will successfully complete a move if the destination cell is a floor, otherwise, it will remain in the same cell it started in. Regardless of whether or not the robot successfully moved, at the end of its move, the robot will be re-oriented in the direction you attempted to move in.

At all times, your robot can see the 3x3 cell of squares immediately surrounding its current location, oriented in the direction it is facing. As an example, consider the sequence of moves below (0 denotes a floor, 1 denotes furniture, the arrow denotes the direction your robot is facing):

```
Actual:                 Robot sees:
  0110                      101
  1>01                      1^0
  0011                      010
          Move forward
  0110                      011
  10>1                      1^1
  0011                      100
           Move right
  0110                      110
  10V1                      1^0
  0011                      011
```

Due to recent concerns over the growth rate of your company’s growth rate slowing down, you have been forced to take increasingly desperate measures, and have begun researching the dark arts. As a result, your robot is equipped with one **special ability**: at any

given moment, **if the robot can see at least one wall**, it may use one step to *apparate*, and teleport to a location, with a possibly different orientation, such that its new surroundings look identical to its old ones. Among all such destinations (including its original location), one will be chosen uniformly at random. An example of apparition is given below:

```
Actual:                 Robot sees:
  0100                      000
  1>00                      1^1
  0101                      010
  0010
            Apparate
  0100                      000
  1000                      1^1
  01^1                      010
  0010
```

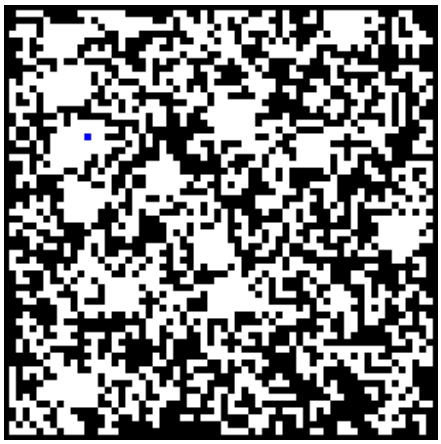
Initially, your robot is given a blueprint of the entire IKEA store, as well as a specified target destination in the store that you must navigate your robot to. However, as IKEA did not grant permission for this test, you had to conduct it in the middle of the night via lowering the robot in through the ceiling. Therefore, you do not know the exact starting position of your robot, and are only given what the robot sees on each step. Your task is to develop an algorithm that can navigate your robot to the target destination in as few steps as possible.

All teams' code will be evaluated once at the end of the contest against the same set of blueprints. These blueprints will be generated randomly using the provided generator. During the contest, feedback and replays will exist to allow teams to check that their code runs properly on our servers, but will otherwise be identical to running locally.

INPUT

The layout of the store will be a randomly generated 64x64 2D array of 0s (floors) and 1s (furniture), where roughly half of the squares will be 1s. The outer perimeter of the grid is guaranteed to be all 1s. There will be one specific cell marked as a 2, which denotes the target destination. It is guaranteed that it is not possible for the robot to see a 1 and a 2 at the same time (i.e., the target is not near any walls), and that it is possible to reach the target from any accessible point in the grid. The grid is NOT guaranteed to be connected otherwise.

Example layout (0 = white, 1 = black, 2 = blue):



Your robot's current view will be given as a 3x3 2D array, of 0s, 1s, and 2s, as described above. The middle entry of the first row denotes the square directly in front of the robot, in the direction it is facing.

Your code will first receive the layout as message a message of the form

```
{
      "layout": [[64x64 grid of 0, 1, or 2]]
}
```

Your robot is uniformly randomly chosen to start at any 0 cell not directly next to the finish. After that, your code will repeatedly receive messages of the form

```
{
      "view": [[3x3 grid as described above]]
}
```

and should respond with

```
{
      "move": one of "forward","backward","left","right","apparate"
}
```

where moving in a direction reorients your view such that the move you just made appears to be forward, and apparating behaves as described in the problem statement.

SCORING

Your score on a single test case is determined by

$$\frac{4096}{64 + \#\text{ of steps to target destination}}$$

If you do not reach the target destination within 4096 steps, your program will terminate with a score of 0. This score will be averaged across all test cases.

LIMITS

Time - 5 seconds (each)
Memory - 256 MB

Murder on the Panda Express

Therefore a wise general strives to feed off the enemy.
Each pound of food taken from the enemy
is equivalent to twenty pounds you provide by yourself.
- Sun Tzu

As lead developer of the delivery company Amazin, you have launched a new program where drones can buy food for people. The business model is simple: your customer orders food, your drones fly out to the restaurant, then fly back with the food. Unfortunately, competition in this space is fierce, and your competitors are desperate and overfunded with venture capitalist fuel, so they have deployed surface-to-air missiles (SAMs) around major metropolitan areas to shoot down Amazin's drones and reduce your efficiency. You are tasked with getting drones from their origin to their destination without getting destroyed — or at least getting as far as you can.

Even more unfortunately, even though Amazin's UAVs can recover a map of what airspace is and isn't safe, Dashdoor has deployed radio jammers to stop you from being able to send this information to your drones, and you'll have to make do with just communicating using 64 bits.

Luckily, your corporate espionage team has managed to extract Dashdoor's code to place SAMs — descriptions are provided below and source code is in the handout.

The airspace is a 256 x 256 grid, where you start on the bottom and must get as far as you can to the top. Any tile in the grid is either safe or unsafe, where going through an unsafe tile results in your drone being shot down and ending your run. You may choose which square along the bottom row to start in. Your drone may move up, down, left, or right (NOT diagonally) through the grid, and your score for a given grid airspace is the furthest up your drone reached while it was still flying.

Before your drone takes off, it manages to receive a single list of 64 bits from your radio tower, where your radio tower is given the full map. Your drone and radio tower code are run separately and independently of one another, so the 64 bits are the only way to communicate between them.

GENERATOR DESCRIPTIONS

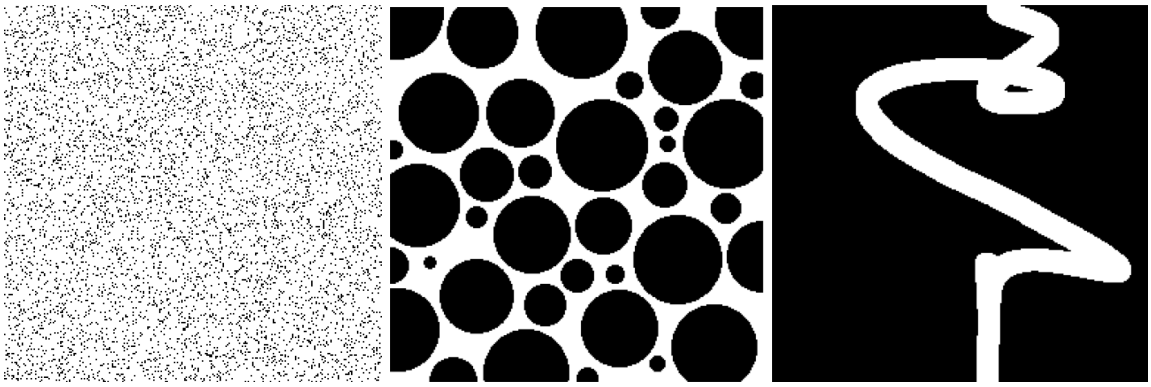
Each generator comes with a specifiable `parameter`, that will change the difficulty of the task.

Task 1 - Random: Each cell has a `parameter`% chance of being unsafe, where `parameter` varies between 6 and 15.

Task 2 - Circles: There are multiple “turrets”, each of which has a shooting radius between 4 and 32, inclusive. Any cell within the shooting radius of these turrets is unsafe, forming a circle of unsafe cells. It is guaranteed that none of these circles overlap, and that there is a space of width at least `parameter` between any two circles, where `parameter` varies from 2 to 10.

Task 3 - Path: Every cell is unsafe, except for one specific winding path, from bottom to top, that is `parameter` cells wide at all points, where `parameter` varies between 10 and 24. This path will be mostly smooth, but otherwise can curve in any direction, including looping back in on itself.

Example Tests:



INPUT

All teams’ code will be evaluated once at the end of the contest against the same set of airspaces. These airspaces will be generated using the provided generators, 100 for each of the generators with random parameters. During the contest, feedback and replays will exist to allow teams to check that their code runs properly on our servers, but will otherwise be identical to running locally.

Your code will be run twice. Each run you recieve

```
{
      “role”: “tower” or “drone”,
      “generator”: “random” or “circles” or “path”,
      “parameter”: the difficulty parameter (as described above),
      (if you are tower) “airspace”: [[2d array of 0s and 1s]],
      (if you are drone) “bits”: [array of 64 0s or 1s],
}
```

NOTE: we are using i-j coordinates rather than x-y, so the first index goes down and the second goes to the right. This means the bottom of the grid is the last row of the array.

If you are the tower, you should respond with

```
{
      "bits": [array of 64 0s or 1s],
}
```

If you are the drone, you should respond with

```
{
      "col": number between 0 and 255,
      "moves": string consisting of ULDR,
}
```

SCORING

Your score on a single test case is determined by the maximum y coordinate reached at any point during your path up until right before it crosses an unsafe square, i.e. a number between 1 and 256, inclusive. If you begin on an unsafe square, you will get a score of 0. This score will be averaged across all test cases.

NOTE: Only the first 2^16 = 65536 moves of your path will be evaluated.

LIMITS

Time - 10 seconds total

Memory - 256 MB

TNTwentyXX

Do not go where the path may lead,
go instead where there is no path and leave a trail.
- Ralph Waldo Emerson

The year is 20XX. Everyone plays Minecraft TNT Run at TAS levels of perfection. Because of this, the winner of a match depends solely on miniscule ping differences. The lag switch metagame has evolved to ridiculous levels due to it being the only remaining factor to decide matches. Humanity has reached its pinnacle.

It seems nothing can stop the great leader of 20XX and his army, the monks who live in great monasteries where they levitate while TASing TNT Run with one hand, and winning tournaments with the other. The tournament metagame has gotten to this point where everything is played out to theoretical perfection, so tournament goers play Rock Paper Scissors for 1ms ping difference, and that's the game.

As lead developer of the delivery company Amazin, you have launched a new program where drones can buy games for people. But, as 20XX killed the casual video game market, you must keep 20XX from coming. To do that, you have to get on their level.

TNT Run is a simple minigame where players run and jump around a 2D grid of tiles, but shortly after they step on a tile it disappears and nobody else can use that tile. In our simulation, games consist of 12 players, all initially in the middle of a 25 x 25 grid. Players move simultaneously, each turn selecting a tile that differs by at most 2 in its X coordinate and at most 2 in its Y coordinate to jump to --- note that multiple players may occupy the same tile. After the initial 6 grace turns, if a player was on a tile on a turn, the next turn that tile is gone and cannot be jumped to by any other player. If the tile they jumped to is still present, in the next turn they are now on that tile, otherwise they fell on that turn. Your score is the number of turns you lasted before falling.

Try as you will, but 20XX is coming. Or maybe, it's already here.

INPUT

On any turn where you are still alive, your program will be given the state of game in its entirety. That is

- A 2D (25 x 25) list of integers representing the current arena state, 1 for a present tile and 0 for empty
- For every player, whether they are alive or dead as well as their zero-indexed coordinates (i, j). The order of the players remains consistent in this list.
- Which index into the player list you are
- How many grace moves are remaining (if any). E.g. if this is one, then on your next move the ground currently underneath you does not disappear, but if this is zero it does.

as

```
{
      "arena": [[0,0,1,1,0,...],[0,1,0,1,1,0,...],...],
      "players": [{
            "alive": true,
            "i": 0,
            "j": 2
      }, {
            "alive": false,
            "i": 0,
            "j": 0
      }, ... ],
      "my_index": 0,
      "grace_moves_left": 0
}
```

and your program should respond with the coordinates of the tile you would like to jump to. This tile's **i** coordinate must be no more than 2 away from your current **i** coordinate and its **j** coordinate must be no more than 2 away from your current **j** coordinate.

```
{
      "i": 1,
      "j": 4
}
```

LIMITS

Time - 2 seconds total
Memory - 256 MB

Non-Fungible Greed Control

BitcoNNEEEEEEEEEEEEEEEEECT! WOOOoo...
bitconnEEEEEEEEEEEEEEEEEEEECT!
- Carlos Matos

As lead developer of the delivery company Amazin, you have launched a new program where drones can buy BitCoin for people. However, since the world's leading BitCoin exchanges have long since forgotten how to implement BlockChain™ Technology (there was a bug in the geeksforgeeks sample implementation), they've decided that it's okay if multiple people own the same coin, and its value will simply be the original value of the coin, divided by the number of people who simultaneously own it (and in fact, one person may even own the same coin multiple times). As a result, there are now only 10 singular coins in existence, which have (total) values of 1, 2, …, 10, respectively.

Each day, you may make (up to) 100 purchases, distributed across the 10 coins, and at the end of the day, you can cash them out for (actual) money. More specifically, if you make c_k purchases of coin k, and in total t_k purchases of coin k were made that day, then your payout at the end of the day will be

$$\sum_{k=1}^{10} k \cdot \frac{c_k}{t_k}$$

Your task is to make as much money as possible, over the course of 30 days. At the end of each day (and not any time before then), you will find out who people bought what types of coin the prior day, and you may use this (and all previous days) to inform what you decide to purchase on the following days.

INPUT

Every round that is played, your bot is given

```
{
     "yesterday": [] on first day, otherwise [[list of 20-length lists corresponding to how much of each coin every player bought the prior day]]
     "scores": [list of all players' current total money made]
     "my_index": Which index you are in the above two lists
}
```

and it should reply with

```
{
```

```
    "buys": [10-length list of how many of each coin you would like
to buy, which must be non-negative integers summing to at most 100]
}
```

LIMITS

Time - 3 seconds total
Memory - 100 MB