Starfleet

Interview

Staff 42 pedago@42.fr

Summary: This document is an interview question for the Starfleet Piscine.

Contents

Chapter I

General rules

- The interview should last between `45 minutes`.
- Both the interviewer and the interviewed student must be present.
- The interviewed student should write his code using a `whiteboard`, with the language of her/his choice.
- At the end of the interview, the interviewer evaluates the student based on the provided criteria.

Read carefully the interview question and solutions, and make sure you `understand` them before the interview. You can't share this document with other students, as they might be interviewed on the same question. Giving them the answer would prevent them from having to solve an unknown question during an interview.

I.1 During the interview

During the interview, we ask you to :

- Make sure the interviewed student `understands` the question.
- Give her/him any `clarification` on the subject that she/he might need.
- Let her/him come up with a solution before you guide her/him to the best solution given the constraints (time and space).
- Ask the student what is the `complexity` of her/his algorithm ? Can it be improved and how ?
- `Guide` her/him to the best solution without giving the answer. You may refer to the `hints` for that.
- You want to evaluate how the interviewed student thinks, so ask her/him to `explain everything` that she/he thinks or writes (there should be no silences).
- If you see a mistake in the code, wait untill the end and give her/him a chance to correct it by her/himself.
- Ask the student to show how the algorithm works on an `example`.
- Ask the student to explain how `limit cases` are handled.
- Bring out to the student any mistake she/he might have done.
- Give `feedback` on her/his performances after the interview.
- Be `fair` in your evaluation.

As always, stay mannerly, polite, respectful and constructive during the interview. If the interview is carried out smoothly, you will both benefit from it !

Chapter II

Mule Island

II.1 Interview question

Consider a map of islands (n x m squares where water is represented by `0s` and islands are formed by `1s`), with a mule placed on one square of an island.

The mule can only move to the 4 squares that are touching the square it is currently on.

If the mule moves `n times`, each time picking one of the 4 directions uniformly and at random (possibly a direction which makes the mule leave the map), `determine the probability that it sinks into the water`.

Once the mule has sunk into the water, it cannot go back to the island. Everything outside the map is considered water.

Example :

For the following map :

```
0 0 0 0                A mule at position row = 3 and col = 2
1 1 0 0                The probability that the mule sinks
1 1 1 0                into water after 1 move is 50%.
0 1 1 0 <-- row = 3
0 1 0 0                After 2 moves, it is 68.75%.
0 0 1 1
    ^
    |
col = 2
```

II.2 Acceptable answers (no constraint)

II.2.1 Recursion

A simple solution is to use recursion to count all the possible ways the mule can sink into water.

```
O(4^n) time , O(n) space
where n is the number of moves
```

Given:

```
int **map; // the map of islands
int nRow;  // the number of rows on the map
int nCol;  // the number of columns on the map
int row;   // the row where the mule is
int col;   // the column where the mule is
int n;     // the number of moves
```

code:

```
int hasNotSunk(int **map, int nRow, int nCol, int i, int j) {
        if (i < 0 || i >= nRow || j < 0 || j >= nCol)
                return (0);
        return (map[i][j]);
}

double probNotSunk(int **map, int nRow, int nCol, int i, int j, int n) {
        double prob = 0;

        if (hasNotSunk(map, nRow, nCol, i, j) == 0)
                return (0);
        if (n == 0)
                return (1);
        prob += probNotSunk(map, nRow, nCol, i - 1, j, n - 1);
        prob += probNotSunk(map, nRow, nCol, i + 1, j, n - 1);
        prob += probNotSunk(map, nRow, nCol, i, j - 1, n - 1);
        prob += probNotSunk(map, nRow, nCol, i, j + 1, n - 1);
        return (prob / (double)4);
}

double sinkMule(int **map, int nRow, int nCol, int row, int col, int n) {
        return ((double)1 - probNotSunk(map, nRow, nCol, row, col, n));
}
```

II.3 Follow up question

The previous algorithm has an exponential runtime. We should not have to recompute from scratch for each move.

Your algorithm must use `dynamic programming` to improve the runtime.

II.3.1 Hints

- Why not `cache` the results from previous moves and use them later?
- Maybe with an hash map.

II.4 Best solutions

II.4.1 Bottom-up dynamic programming

The solution is to use a `hash map`, with the same size as the map, in order to store the probability that the mule doest not sink for every position on the map.

It is bottom-up dynamic programming since we start with 0 moves and walk our way to n moves.

Let's see with an example :

```
Given this 3 x 3 map :
    0   0   0
    1   1   0
    1   1   1

Hash map for 0 move :
(i.e., probability that, for each initial position of the mule,
it stays on the island after 0 move)
 0.00  0.00  0.00
 1.00  1.00  0.00         for i = 1 and j = 1,
 1.00  1.00  1.00         prob = 1.00

Hash map for 1 move :
 0.00  0.00  0.00
 0.50  0.50  0.00         for i = 1 and j = 1,
 0.50  0.75  0.25         prob = (0.00 + 0.00 + 1.00 + 1.00) / 4

Hash map for 2 moves :
 0.00  0.00  0.00
 0.25  0.31  0.00         for i = 1 and j = 1,
 0.31  0.31  0.19         prob = (0.00 + 0.00 + 0.75 + 0.50) / 4
```

`Note` : The probability that the mule sinks into the water is the (1 - prob).

```
O(n * nRow * nCol) time , O(nRow * nCol) space
where n in the number of moves
and nRow and nCol are the numbers of row and columns of the map
```

code for the `hash map` : (these functions do not need to be coded during the interview)

```
double **initHashMap(int nRow, int nCol) {
        double **map;

        map = (double **)malloc(sizeof(double *) * nRow);
        for (int i = 0; i < nRow; i++) {
                map[i] = (double *)malloc(sizeof(double) * nCol);
                memset(map[i], 0, sizeof(double) * nCol);
        }
        return (map);
}

void fillHashMap(double **map, int **map, int nRow, int nCol) {
        for (int i = 0; i < nRow; i++) {
                for (int j = 0; j < nCol; j++) {
                        map[i][j] = map[i][j];
                }
        }
}
```

code:

```
double getHashMap(double **map, int nRow, int nCol, int i, int j) {
        if (i < 0 || i >= nRow || j < 0 || j >= nCol)
                return (0);
        return (map[i][j]);
}

double getProbNextMove(double **map, int nRow, int nCol, int i, int j) {
        double p = 0;

        if (getHashMap(map, nRow, nCol, i, j) == 0)
                return (0);
        p += getHashMap(map, nRow, nCol, i - 1, j);
        p += getHashMap(map, nRow, nCol, i + 1, j);
        p += getHashMap(map, nRow, nCol, i, j - 1);
        p += getHashMap(map, nRow, nCol, i, j + 1);
        return (p / (double)4);
}

double probNotSunk(int **map, int nRow, int nCol, int row, int col, int n) {
        double **map1; // hashmap of probabilities for n moves
        double **map2; // hashmap of probabilities for n+1 moves
        double **tmp;

        if (row < 0 || row >= nRow || col < 0 || col >= nCol)
                return (-1);
        if (n == 0)
                return (map[row][col]);

        // initialize 2 hashmap of nRow rows and nCol columms
        map1 = initHashMap(nRow, nCol);
        map2 = initHashMap(nRow, nCol);

        // compute hashmap with probabilities for 0 move
        fillHashMap(map1, map, nRow, nCol);

        while (n) {

                // compute hashmap for n+ 1 moves based on hastable for n moves
                for (int i = 0; i < nRow; i++) {
                        for (int j = 0; j < nCol; j++) {
                                map2[i][j] = getProbNextMove(map1, nRow, nCol, i, j);
                        }
                }

                // swap hashmaps
                tmp = map1;
                map1 = map2;
                map2 = tmp;

                n--;
        }

        // return the probability that the mule does not sink into the water
        return (map1[row][col]);
}

double sinkMule(int **map, int nRow, int nCol, int row, int col, int n) {
        return ((double)1 - probNotSunk(map, nRow, nCol, row, col, n));
}
```