Tilt Game (Team 48)

Code

```
#include <bits/stdc++.h>

#include <utility>
using namespace std;

#ifndef ONLINE_JUDGE
#include "Debug.cpp"
#define FileI(fileName) freopen(fileName, "r", stdin);
#define FileO(fileName) freopen(fileName, "w", stdout);
#else
#define FileI(fileName)
#define FileO(fileName)
#define debug(...) 1
#define Time(i, x...) x
#endif
#define un unsigned
#define ld long double
#define LL long long
#define allc(x) begin(x), end(x)
#define rallc(x) rbegin(x), rend(x)
#define all(x,i,j) begin(x) + i, begin(x) + j
#define Test int TC, _(0); cin >> TC; while(_++ < TC)

template<typename T>
using HashedBoard = vector<pair<T, T>>;

string output, expected;
const string Dir[] = {"left", "up", "right", "down"};

struct Graph
```

```
{
    enum Direction {NONE = -1, LEFT, UP, RIGHT, DOWN};
    enum Algorithm {S_LOG_S, N_SQUARED};
    struct Node
    {
        HashedBoard<un short> ballBoardState;
        Direction parentDir = NONE;
        int sourceDistance = 0;
        Node* Parent = nullptr;
        Node() = default;
        Node(const HashedBoard<un short>& ballBoardState, i
nt sourceDistance = 0, Node* Parent = nullptr, Direction pa
rentDir = NONE)
        {
            this->ballBoardState = ballBoardState;
            this->sourceDistance = sourceDistance;
            this->Parent = Parent;
            this->parentDir = parentDir;
        }
    };

    unsigned moveTimer = 0; // counter for how many times w
e call move method
    Node* initialState, *finalState = nullptr;
    vector<vector<char>> globalBoard; // grid without balls
    vector<vector<pair<un short, un short>>> obstaclesRowPo
s, obstaclesColPos; // used for constant queries about the
nearest obstacle in 4 main directions
    vector<pair<un short, un>> ballsRowPos; // dynamically
updating the nearest ball (above or below) me for constant
queries
    pair<un short, un short> target; // coordinates of the
target
    const short toBase = UCHAR_MAX + 1; // toBase = 256
    vector<vector<string>> baseChar; // constant queries fo
r ContainerHashing function
    Algorithm choosenAlgo = N_SQUARED; // algorithm to appl
y on vertical moves
```

```
    vector<vector<Direction>> adj = {{UP, DOWN}, {LEFT, RIG
HT}, {UP, DOWN}, {LEFT, RIGHT}}; // for every direction wha
t are the next directions

    Graph(const vector<vector<char>>& board, const pair<un
short, un short>& target) // O(max(slogs, n^2))
    {
        HashedBoard<un short> slidersPos;
        int n = board.size() - 2;

        baseChar = vector<vector<string>>(n + 2, vector<str
ing>(n + 2));
        for (int i = 0; i <= n + 1; i++)
            for (int j = 0; j <= n + 1; j++)
                ConvFromDec(i, baseChar[i][j]), ConvFromDec
(j, baseChar[i][j]);

        globalBoard = board;
        ballsRowPos = vector<pair<un short, un>>(n + 2, {-
1, moveTimer});
        obstaclesRowPos = obstaclesColPos = vector<vector<p
air<un short, un short>>>(n + 2, vector<pair<un short, un s
hort>>(n + 2));
        for (int i = 0; i <= n + 1; i++)
            for (int lj = 0, rj = n + 1, j = 0, jj = n + 1;
j <= n + 1; j++, jj--)
            {
                if (board[i][j] != '#')
                    obstaclesColPos[i][j].first = lj;
                else
                    lj = j;
                if (board[i][jj] != '#')
                    obstaclesColPos[i][jj].second = rj;
                else
                    rj = jj;
                if (board[i][j] == 'o')
                    slidersPos.emplace_back(i, j), globalBo
ard[i][j] = '.';
```

```
            }
        for (int j = 0; j <= n + 1; j++)
            for (int ui = 0, di = n + 1, i = 0, ii = n + 1;
i <= n + 1; i++, ii--)
            {
                if (board[i][j] != '#')
                    obstaclesRowPos[j][i].first = ui;
                else
                    ui = i;
                if (board[ii][j] != '#')
                    obstaclesRowPos[j][ii].second = di;
                else
                    di = ii;
            }

        initialState = new Node(slidersPos); // first state
have HashedBoard of balls sorted about rows then cols
        this->target = target;

        auto ci = chrono::high_resolution_clock::now();
        Move(slidersPos, UP);
        auto cf = chrono::high_resolution_clock::now();
        auto squareTime = chrono::duration_cast<chrono::mic
roseconds>(cf - ci).count();
        choosenAlgo = S_LOG_S;
        ci = chrono::high_resolution_clock::now();
        Move(slidersPos, UP);
        cf = chrono::high_resolution_clock::now();
        auto s_lg_s = chrono::duration_cast<chrono::microse
conds>(cf - ci).count();
        if (squareTime <= s_lg_s)
            choosenAlgo = N_SQUARED;
        debug(squareTime);
        debug(s_lg_s);
    }

    HashedBoard<un short> Move(const HashedBoard<un short>&
currentBoard, Direction dir) // O(min(s*log(s), n^2)), Ω(s)
```

```
    {
        moveTimer++;
        un s = currentBoard.size(); // number of sliders
        switch (dir)
        {
            case RIGHT:
            {
                auto nextBoard = currentBoard;
                un short mnj;
                int it = s - 1;
                if (s)
                    nextBoard[it].second = obstaclesColPos
[nextBoard[it].first][nextBoard[it].second].second - 1;
                for (it--; ~it; it--)
                {
                    auto& [x, y] = nextBoard[it];
                    mnj = obstaclesColPos[x][y].second;
                    if (nextBoard[it + 1].first == x)
                        mnj = min(mnj, nextBoard[it + 1].se
cond);
                    y = mnj - 1;
                }
                return nextBoard;
            }
            case LEFT:
            {
                auto nextBoard = currentBoard;
                un short mxj;
                int it = 0;
                if (s)
                    nextBoard[it].second = obstaclesColPos
[nextBoard[it].first][nextBoard[it].second].first + 1;
                for (it++; it < s; it++)
                {
                    auto& [x, y] = nextBoard[it];
                    mxj = obstaclesColPos[x][y].first;
                    if (nextBoard[it - 1].first == x)
                        mxj = max(mxj, nextBoard[it - 1].se
```

```
cond);
                    y = mxj + 1;
                }
                return nextBoard;
            }
            case UP:
            {
                switch (choosenAlgo)
                {
                    case S_LOG_S:
                    {
                        HashedBoard<un short> nextBoard;
                        un short mxi;
                        for (int it = 0; it < s; it++)
                        {
                            auto &[x, y] = currentBoard[i
t];
                            mxi = obstaclesRowPos[y][x].fir
st;
                            if (ballsRowPos[y].second < mov
eTimer)
                                ballsRowPos[y].second = mov
eTimer;
                            else
                                mxi = max(mxi, ballsRowPos
[y].first);
                            ballsRowPos[y].first = mxi + 1;
                            nextBoard.emplace_back(mxi + 1,
y);
                        }
                        stable_sort(allc(nextBoard));
                        return nextBoard;
                    }
                    case N_SQUARED:
                    {
                        un n = globalBoard.size() - 2;
                        HashedBoard<un short> nextBoard;
                        un short mxi;
```

```
                        for (int it = 0; it < s; it++)
                        {
                            auto &[x, y] = currentBoard[i
t];
                            mxi = obstaclesRowPos[y][x].fir
st;
                            if (ballsRowPos[y].second < mov
eTimer)
                                ballsRowPos[y].second = mov
eTimer;
                            else
                                mxi = max(mxi, ballsRowPos
[y].first);
                            ballsRowPos[y].first = mxi + 1;
                            globalBoard[mxi + 1][y] = 'o';
                        }
                        for (int i = 1; i <= n; i++)
                            for (int j = 1; j <= n; j++)
                                if (globalBoard[i][j] ==
'o')
                                    globalBoard[i][j] =
'.', nextBoard.emplace_back(i, j);
                        return nextBoard;
                    }
                }
            }
            case DOWN:
            {
                switch (choosenAlgo)
                {
                    case S_LOG_S:
                    {
                        HashedBoard<un short> nextBoard;
                        un short mni;
                        for (int it = s - 1; ~it; it--)
                        {
                            auto& [x, y] = currentBoard[i
t];
```

```
                            mni = obstaclesRowPos[y][x].sec
ond;
                            if (ballsRowPos[y].second < mov
eTimer)
                                ballsRowPos[y].second = mov
eTimer;
                            else
                                mni = min(mni, ballsRowPos
[y].first);
                            ballsRowPos[y].first = mni - 1;
                            nextBoard.emplace_back(mni - 1,
y);
                        }
                        stable_sort(allc(nextBoard));
                        return nextBoard;
                    }
                    case N_SQUARED:
                    {
                        un n = globalBoard.size() - 2;
                        HashedBoard<un short> nextBoard;
                        un short mni;
                        for (int it = s - 1; ~it; it--)
                        {
                            auto& [x, y] = currentBoard[i
t];
                            mni = obstaclesRowPos[y][x].sec
ond;
                            if (ballsRowPos[y].second < mov
eTimer)
                                ballsRowPos[y].second = mov
eTimer;
                            else
                                mni = min(mni, ballsRowPos
[y].first);
                            ballsRowPos[y].first = mni - 1;
                            globalBoard[mni - 1][y] = 'o';
                        }
                        for (int i = 1; i <= n; i++)
```

```
                              for (int j = 1; j <= n; j++)
                                  if (globalBoard[i][j] ==
'o')
                                      globalBoard[i][j] =
'.', nextBoard.emplace_back(i, j);
                          return nextBoard;
                      }
                  }
              }
              case NONE:
                  return currentBoard;
          }
          return currentBoard;
      }

      vector<vector<char>> Debug(HashedBoard<un short> c)
      {
          vector<vector<char>> tmp = globalBoard;
          for (auto& [x, y] : c)
              tmp[x][y] = 'o';
          return tmp;
      }

      void PrintAnswer(int Difficulty)
      {
          Time(Algorithm, ShortestPath_BFS();)

          if (!finalState)
              return cerr << (output += "Unsolvable\n"), void
();

          auto curr = finalState;
          deque<Direction> path;
          while (curr)
          {
              path.push_front(curr->parentDir);
              curr = curr->Parent;
          }
```

```
        int k = path.size();

        output += "Solvable\nMin number of moves: " + to_st
ring(k - 1) + "\nSequence of moves: ";
        for (int i = 1; i < k; i++)
            output += Dir[path[i]] + ", ";
        cerr << (output += '\n');


        if (!Difficulty)
            return;

        auto curBallBoardState = initialState->ballBoardSta
te;
        cerr << "Initial\n";
        debug(Debug(curBallBoardState));

        for (int i = 1; i < k; i++)
            cerr << Dir[path[i]] << "\n", debug(Debug(curBa
llBoardState = Move(curBallBoardState, path[i])));













    }

    inline void ConvFromDec(un short n, string& ret) // O(l
og256(n))
    {
```

```
        while (n)
            ret += (n % toBase), n /= toBase;
        ret += ' ';
    }

    string ContainerHashing(const HashedBoard<un short>& co
ntainer) // O(s*log256(n))
    {
        string ret;
        un s = container.size();
        for (int it = 0; it < s; it++)
            ret += baseChar[container[it].first][container
[it].second];
        ret.pop_back();
        return ret;
    }

    void ShortestPath_BFS() // O(2^min(k, d) * min(s log
(s), n^2)) : (k) is the depth of the answer if solvable,
(d) is the max depth of the graph,   (d) is O(log((n^2 - b)
C(s)))
    {
        queue<Node*> nxt;
        unordered_set<string> vis;
        vis.insert(ContainerHashing(initialState->ballBoard
State));

        if (binary_search(allc(initialState->ballBoardStat
e), target))
            return finalState = initialState, void();
        for (auto dir : {LEFT, UP, RIGHT, DOWN})
        {
            auto newBoardState = move(Move(initialState->ba
llBoardState, dir));
            if (vis.insert(ContainerHashing(newBoardStat
e)).second)
            {
                auto child = new Node(newBoardState, initia
```

```
lState->sourceDistance + 1, initialState, dir);
                if (binary_search(allc(newBoardState), targ
et))
                    return finalState = child, void();
                nxt.push(child);
            }
        }
        while (!nxt.empty())
        {
            auto cur = nxt.front(); nxt.pop();
            for (auto dir : adj[cur->parentDir])
            {
                auto newBoardState = move(Move(cur->ballBoa
rdState, dir));
                if (vis.insert(ContainerHashing(newBoardSta
te)).second)
                {
                    auto child = new Node(newBoardState, cu
r->sourceDistance + 1, cur, dir);
                    if (binary_search(allc(newBoardState),
target))
                        return finalState = child, void();
                    nxt.push(child);
                }
            }
        }
        return void();
    }
};

void Solve()
{
    short chooseDifficulty; cout << "Choose Test Difficulty
1 : sample , 2 : Hard .\n"; cin >> chooseDifficulty;
    string FileName, outFileName;
    if (--chooseDifficulty)
    {
        FileName += "Complete Tests/";
```

```
        cout << "\nChoose Case size: ";
        short sz; cin >> sz;
        if (sz == 1)
            FileName += "1 small/";
        else if (sz == 2)
            FileName += "2 medium/";
        else if (sz == 3)
            FileName += "3 large/";

        short inputCase; cout << "\nChoose Test Case File: 
";
        cin >> inputCase;
        FileName += "Case " + to_string(inputCase) + "/Cas
e" + to_string(inputCase);
    }
    else
    {
        short inputCase; cout << "\nChoose Test Case File: 
";
        cin >> inputCase;
        FileName = "Sample Tests/Case" + to_string(inputCas
e);
    }
    outFileName = FileName + "-output.txt";
    FileName += ".txt";
    FileI(FileName.c_str());

    int n; cin >> n;
    vector<vector<char>> grid(n + 2, vector<char>(n + 2, 
'#'));

    char cell;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
        {
            while (cin >> cell, cell != '#' && cell != '.' 
&& cell != 'o');
            grid[i][j] = cell;
```

```
        }


    pair<un short, un short> target;

    cin >> target.second, cin.ignore(), cin >> target.firs
t;
    target.first++;
    target.second++;

    Graph g(grid, target);
    g.PrintAnswer(chooseDifficulty - 1);


    fclose(stdin);
    FileI(outFileName.c_str())
    string tmp;
    while (getline(cin, tmp) && tmp != "Initial")
        expected += tmp + '\n';
    cerr << "\t\t\t\t\t" << (output == expected ? "ACCEPTE
D" : "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX [REJECTE
D] XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX");
}

signed main()
{
    Solve(), cout << '\n';
    return 0;
}
```

Running Time For Hard Tests

test:1,1 → 45.5s

test:1,2 → 32.4s

test:1,3 → 9.6s

test:2,1 → 10.8s

test:2,2 → 0.012s

test:2,3 → 29s

test3,1 → 0.018s

test3,2 → 4.5s

test:3,3 → 0.102s

Analysis

`ConvFromDec(n)`

given:

1) log(base, number) is the way to represent the log function

2) toBase = UCHAR_MAX + 1 = 256

this function takes the number and keeps dividing this number by toBase

→ Body of the loop consists of arithmetic operations so it takes Constant steps → **Θ(1)**

→ the loop it self takes exactly log(toBase, n) steps

→ this is **Θ(log(n))**

`ContainerHashing(container)`

given:

1) n = container size

this function takes the container and keeps looping on its size

→ Body of the loop consists of arithmetic operations so it takes Constant steps → **Θ(1)**

→ the loop it self takes exactly n steps

→ this is **Θ(n)**

`Debug(hashedBoard)`

given:

1) n = hashed board size

this function takes the hashed board and keeps looping on its size

→ Body of the loop consists of assignment operation so it takes Constant steps → **Θ(1)**

→ the loop it self takes exactly n steps

→ this is **Θ(n)**

`Move(board, direction)`

code changed not comp

given:

1) s = number of sliders in the given grid

2) n*n is the size of the grid

3) this function has 2 modes that can work based on deciding what is minimum (s*log(s) and n*n) or s in the other case

let's assume that we will move 'UP'

→ if s*log(s) > n*n → this code take 2*s + n*n so it takes **Θ(s + n*n) = Θ(n^2)**

```
case N_SQUARED:
{
        un n = globalBoard.size() - 2; /// Θ(1)
        HashedBoard<un short> nextBoard; /// Θ(1)
        un short mxi; /// Θ(1)
        for (int it = 0; it < s; it++) /// Θ(s)
     {
             auto &[x, y] = currentBoard[it]; /// Θ(1)
              mxi = obstaclesRowPos[y][x].first; /// Θ(1)
              if (ballsRowPos[y].second < moveTimer) /// Θ
(1)
                  ballsRowPos[y].second = moveTimer; /// Θ
(1)
```

```
                else /// O(1)
                     mxi = max(mxi, ballsRowPos[y].first);
/// O(1)
                ballsRowPos[y].first = mxi + 1; /// O(1)
                globalBoard[mxi + 1][y] = 'o'; /// O(1)
       }
       for (int i = 1; i <= n; i++) /// O(n)
             for (int j = 1; j <= n; j++) /// O(n)
                 if (globalBoard[i][j] == 'o') /// O(1)
                       globalBoard[i][j] = '.', nextBoard.
emplace_back(i, j);/// O(1)
     return nextBoard; /// O(s)
}
```

→ if s*log(s) < n*n → this code takes s + s*log(s) so it takes **Θ(s + s*log(s)) = Θ(s*log(s))**

```
case S_LOG_S:
{
      HashedBoard<un short> nextBoard; /// O(1)
      un short mxi; /// O(1)
      for (int it = 0; it < s; it++) /// O(s)
      {
              auto &[x, y] = currentBoard[it]; /// O(1)
              mxi = obstaclesRowPos[y][x].first; /// O(1)
              if (ballsRowPos[y].second < moveTimer) /// O
(1)
                     ballsRowPos[y].second = moveTimer; ///
O(1)
              else /// O(1)
                     mxi = max(mxi, ballsRowPos[y].first);
/// O(1)
              ballsRowPos[y].first = mxi + 1; /// O(1)
              nextBoard.emplace_back(mxi + 1, y); /// O(1)
       }
       stable_sort(allc(nextBoard)); /// O(slog(s))
```

```
        return nextBoard;/// O(s)
}
```

NOTE: the ‘DOWN’ runs in the same time complexity with little to no difference in constant times so we can consider that → ‘DOWN’ = ‘UP’ = **Θ(min(s*log(s), n*n))**

let’s assume that we will move ‘RIGHT’

→ we have 1 choice here and it is **Θ(s)**

```
auto nextBoard = currentBoard; /// O(s)
un short mnj; /// O(1)
int it = s - 1; /// O(1)
if (s) /// O(1)
nextBoard[it].second=obstaclesColPos[nextBoard[it].first][n
extBoard[it].second].secon-1;
for (it--; ~it; it--)/// O(s)
{
       auto& [x, y] = nextBoard[it]; /// O(1)
       mnj = obstaclesColPos[x][y].second;/// O(1)
       if (nextBoard[it + 1].first == x)/// O(1)
             mnj = min(mnj, nextBoard[it + 1].second);/// O
(1)
       y = mnj - 1;/// O(1)
 }
 return nextBoard;/// O(s)
```

NOTE: the ‘LEFT’ runs in the same time complexity with little to no difference in constant times so we can consider that → ‘RIGHT’ = ‘LEFT’ = **Θ(s)**

In conclusion, this function depends on the direction of the move if it is right or left it runs in **Θ(s)** but if it is up or down it runs in **Θ(min(s*log(s), n*n))**

→ so the over all function complexity is Ω(s) and O(min(s*log(s), n*n))

`shortestPath_BFS()`

given:

1) s = number of sliders in the given grid

2) n*n is the size of the grid

3) k = optimal path size (number of steps to reach the goal if it exists)

4) d = max depth of tree (where k ≤ d)

we can divide this function to three main blocks so let's analyze them

1- first block complexity:

Θ(s*log(s))

```
queue<Node*> nxt; /// O(1)
unordered_set<string> vis; /// O(1)
vis.insert(ContainerHashing(initialState->ballBoardState));
/// O(1)

if (binary_search(allc(initialState->ballBoardState), targe
t)) /// O(slog(s))
       return finalState = initialState, void(); /// O(1)
```

2- second block complexity:

because of observations in the third block we can conclude that calling the MOVE() will result in exact complexity of **Θ(s + min(s*logs, n^2))**

```
for (auto dir : {LEFT, UP, RIGHT, DOWN}) /// O(4*MOVE())
{
  auto newBoardState = move(Move(initialState->ballBoardSta
te, dir)); /// O(MOVE())
  if (vis.insert(ContainerHashing(newBoardState)).second)
/// O(s)
  {
```

```
      auto child = new Node(newBoardState, initialState->s
ourceDistance + 1, initialState, dir); /// O(1)

   if (binary_search(allc(newBoardState), target)) /// O(sl
og(s))
          return finalState = child, void(); /// O(1)
  nxt.push(child); /// O(1)
  }
}
```

3- third block is where we have the main logic of the BFS

first let's consider some facts that will be useful for the analysis:

- max nodes that the BFS will touch is 4 * (2^(d+1) - 1) such that d is the depth of the tree
- (2^(k) - 1) can become (2^(d+1) - 1) if we build the whole tree
- sometimes we will break of this loop if the answer was found so this will be less than (2^(k) - 1)

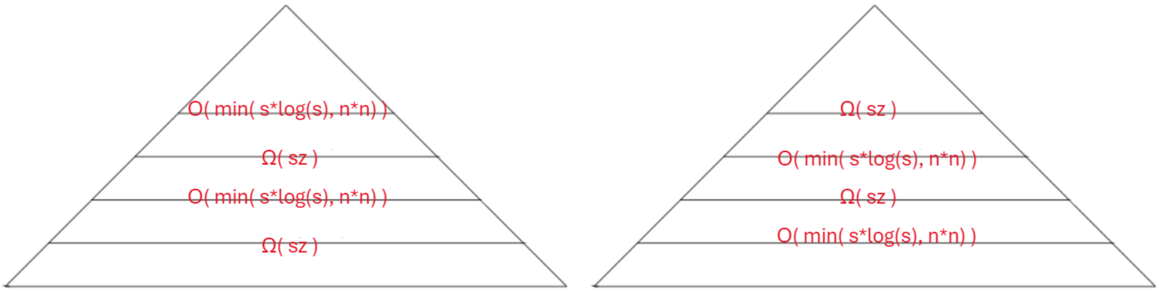
an interesting observation:

Move() can be called 4 times at the beginning however as we go we can call it 2 times as we go.. and another thing is that if we move 'LEFT' or 'RIGHT' we can move 'UP' or 'DOWN' after that and vice versa.. when we consider the 2 facts we conclude that 2 calls from the initial 4 calls will make Move() run in Ω(s) then O(min(s*log(s), n*n)) and the other calls will run O(min(s*log(s), n*n)) then Ω(s).. so this means half of the calls for MOVE() will be Ω(s) and the other half will be

O(min(s*log(s), n*n)).. **so although MOVE() doesn't have an exact complexity we know that we will call it from BFS in O(2* (2^(d+1) - 1) * min(s*log(s), n*n)).**

We can't conclude that this block have an exact complexity because as stated earlier sometimes we won't enter this loop at all so this is Ω(1)

these pyramids represents the levels of the tree




```
while (!nxt.empty())
{
/// the touches of nodes is analyzed above
    auto cur = nxt.front(); nxt.pop();
    ////adj.size() = 2 except one time where it equals 4
    for (auto dir : adj[cur->parentDir])
    {
        auto newBoardState = move(Move(cur->ballBoardState,
dir)); ///O(Move())
        if (vis.insert(ContainerHashing(newBoardState)).sec
ond)
        {
            auto child = new Node(newBoardState, cur->sourc
eDistance + 1, cur, dir);
            if (binary_search(allc(newBoardState), target))
/// O(slogs)
                return finalState = child, void();  /// O
(1)
            nxt.push(child);  /// O(1)
        }
    }
}
return void();
```

so let's combine the three blocks of code together:

Ω(s + min(slogs, n^2) + 1)

→ this becomes Ω(min(slogs, n^2))

O(s + k + min(slogs, n^2) + k + (2^(d+1) - 1) * min(s*log(s), n*n))

→ this becomes O(k + (2^(d+1) - 1) * min(s*log(s), n*n))

`Graph()`

given:

1) n*n is the size of the grid

let's break this constructor function into 2 main blocks

1- first block: we have n*n nested loop that calls ConvFromDec()

each call runs in logarithmic time so this runs in ∑[log(i)] such that (i is from 0 to n+1)

so this block runs in O(n*n*(log(n)*2)) = O(n*n*log(n))

```
vector<pair<un short, un short>> slidersPos; /// O(1)
int n = board.size() - 2; /// O(1)
globalBoard = board; /// O(n*n)
ballsRowPos = vector<pair<un short, unsigned>>(n + 2, {-1,
moveTimer}); /// O(n)
baseChar = vector<vector<string>>(n + 2, vector<string>(n +
2)); /// O(n*n)

/// O(n*n*(log(i) + log(j)))
for (int i = 0; i <= n + 1; i++)
    for (int j = 0; j <= n + 1; j++)
        ConvFromDec(i, baseChar[i][j]), ConvFromDec(j, base
Char[i][j]);
```

2-second block: this blocks performs n*n moves then it calls MOVE()

so it becomes Ω(n*n + s + k) or O(n*n + min(s*log(s), n*n) + k)

```
/// O(n*n)
obstaclesRowPos = obstaclesColPos = vector<vector<pair<un s
hort, un short>>>(n + 2, vector<pair<un short, un short>>(n
+ 2));

/// O(n*n)
for (int i = 0; i <= n + 1; i++)
    for (int lj = 0, rj = n + 1, j = 0, jj = n + 1; j <= n
+ 1; j++, jj--)
    {
        if (board[i][j] != '#')/// O(1)
            obstaclesColPos[i][j].first = lj; /// O(1)
        else
            lj = j;/// O(1)
        if (board[i][jj] != '#')/// O(1)
            obstaclesColPos[i][jj].second = rj; /// O(1)
        else
            rj = jj;/// O(1)
        if (board[i][j] == 'o') /// O(1)
            slidersPos.emplace_back(i, j), globalBoard[i]
[j] = '.'; /// O(1)
    }

/// O(n*n)
for (int j = 0; j <= n + 1; j++)
    for (int ui = 0, di = n + 1, i = 0, ii = n + 1; i <= n
+ 1; i++, ii--)
    {
        if (board[i][j] != '#') /// O(1)
            obstaclesRowPos[j][i].first = ui; /// O(1)
        else
            ui = i;/// O(1)
        if (board[ii][j] != '#')/// O(1)
            obstaclesRowPos[j][ii].second = di; /// O(1)
        else
            di = ii; /// O(1)
    }
```

```
initialState = new Node(slidersPos); /// O(1)
this->target = target; /// O(1)

auto ci = chrono::high_resolution_clock::now();
Move(slidersPos,UP); ///time of move
auto cf = chrono::high_resolution_clock::now();
auto squareTime = chrono::duration_cast<chrono::microsecond
s>(cf - ci).count();
dimentionComplexity = 0;
ci = chrono::high_resolution_clock::now();
Move(slidersPos,UP); ///time of move
cf = chrono::high_resolution_clock::now();
auto s_lg_s = chrono::duration_cast<chrono::microseconds>(c
f - ci).count();
if (squareTime <= s_lg_s)
    dimentionComplexity = 1;
```

this function runs in O(n*n + n*n*log(n) + min(s*log(s), n*n) + k)

→ which can be simplified to O(n*n*log(n) + min(s*log(s), n*n) + k)

`PrintAnswer()`

given:

1) s = number of sliders in the given grid

2) n*n is the size of the grid

3) k = optimal path size (number of steps to reach the goal if it exists)

4) d = max depth of tree (where k ≤ d)

we have several loops that needs k steps and the body of these loops take constant times then we have a loop that takes k steps and calls MOVE().. however, we calculated the exact for the third block in the BFS() function we can do the same here and get the exact complexity as the 2 trees balances each others

so this is Θ(k/2 * (min(slogs, n*n) + s)) = Θ(k * min(s*log(s), n*n))