COL 106 : Data Structures and Algorithms
Semester II, 2022-23, IIT Delhi

Assignment - 5 (due on 8th April, 11:00 PM)

Important Guidelines:

- You must ensure that your submitted code runs in the JDK 11.0.17 environment.
- You are not allowed to share your code with anyone. Cheating of any form will lead to strict disciplinary action. Typical penalty would include Fail grade in the course.
- The assignment must be done individually, and no group submissions are allowed.
- All starter code files must be included in the submission, which must be uploaded on the gradescope without compressing the files.
- The names of files and method signatures must not be changed in the starter code.
- You should write the code in the portion of the starter code labelled **"To be filled in by the student"**.
- Do not declare global variables as your code will run on multiple testcases and hence might give incorrect results when you submit the code on autograder.
- While submitting the code zip the folders Q1 and Q2 into a single file rather than zipping the folder containing Q1 and Q2.

1 Cheapest Cell Tower using Quadtrees

Quadtrees are an efficient data-structure used to represent multi-dimensional spatial data. Quadtrees come in many flavours, we shall be looking at one of them, namely Point Quadtrees, in this assignment to represent points on a 2-dimensional cartesian plane.

A Point Quadtree contains a point in the cartesian plane at every node. Further, a point (x_0, y_0) in it has 4 children, each representing a quadrant obtained by lines $x = x_0$ and $y = y_0$. (See Figure below). Note that, the structure of the Quadtree is dependent on the order of insertion. In the example below, the order of insertion is (5,0), (6,2), (2,2) and (3,0)

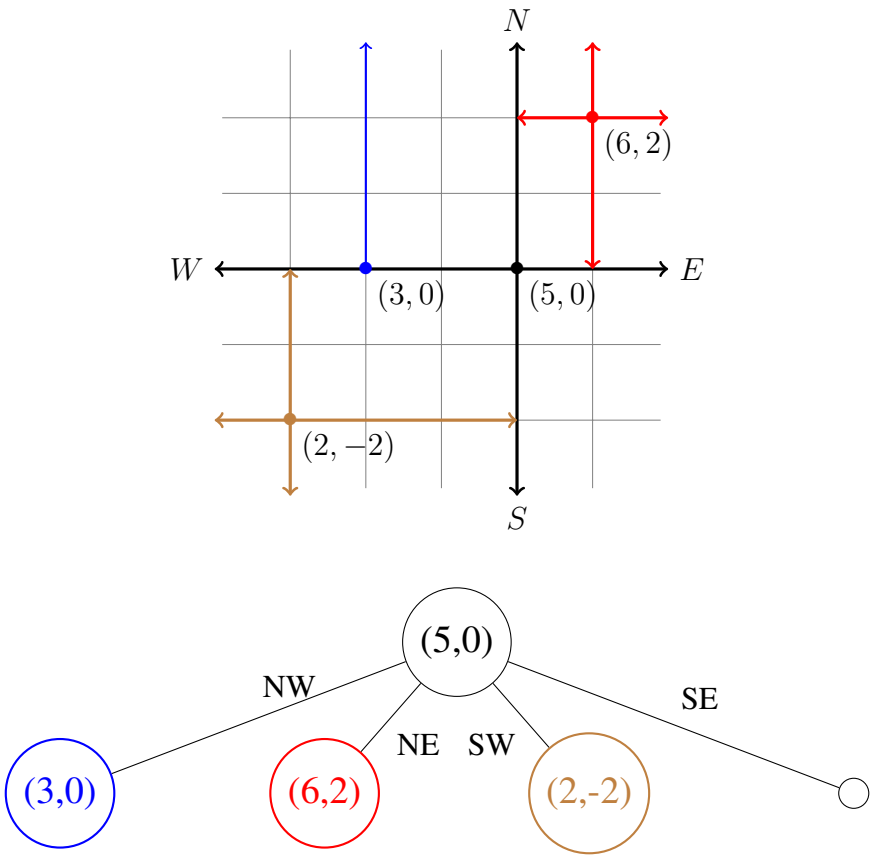

Figure 1: An example Quadtree representing the points $(5, 0),\ (6, 2),\ (2, -2),\ (3, 0)$.

We will use Point Quadtrees to model the map of cell towers in a city. The class `CellTower` that is used to represent individual cell towers, has the following components.

- Instance variables

  1. `int x`: The x-coordinate of the cell tower.
  2. `int y`: The y-coordinate of the cell tower.
  3. `int cost`: A non-negative integer representing the cost of the services offered by the cell tower.

- Member functions

  `public double distance(int b, int c)`: Calculate the distance (L_2-norm) between the cell tower's location and the point `(b,c)`.

The class `PointQuadtree` has the following components.

- Instance variables

  `PointQuadtreeNode root`: The root of the `PointQuadTree`.

- Member functions

  1. `public boolean insert(CellTower a)`: Insert the cell tower `a` into the Quadtree. Return `true` if successful and `false` if a duplicate exists.

     Use the tie-breaking scheme represented by figure below. For example, in the tree in Figure 1 the point $(3, 0)$ lied at the boundary. Following the below scheme, it should be added to the NW quadrant.

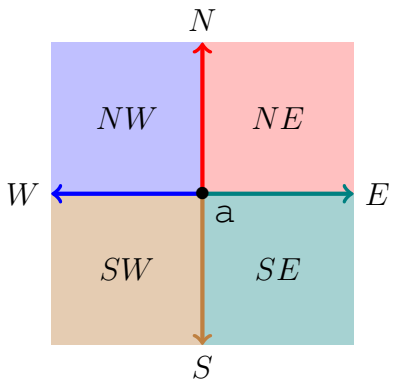

Figure 2: Tie breaking scheme for insertions in Quadtree.

  2. `public boolean cellTowerAt(int x, int y)`: Check if a cell tower exists at location `(x,y)`. Return `true` or `false` accordingly.
  3. `public CellTower chooseCellTower(int x, int y, int r)`: Return the cell tower that offers cheapest services, that is, has minimum cost, and lies within distance `r` from point `(x,y)`. Return `null` if no such cell tower exists.

`PointQuadtree` is built using class `PointQuadtreeNode` which contains a `CellTower` and a quadrant array.

- Instance variables

  1. `CellTower c`: The cell tower at this node.

2. `PointQuadtreeNode[] quadrants`: Array of children each representing a quadrant in the order NW, NE, SW, SE. You can also find an `enum` in the starter code you can make use of for this purpose.

Note that, like always, you are not allowed to change the signatures of existing methods or create new instance variables. You are, of course, free to create any private helper methods to help your code.

```
Test case:

PointQuadtree obj = new PointQuadtree();

CellTower c1 = new CellTower(0,0,5);
CellTower c2 = new CellTower(-2,0,4);
CellTower c3 = new CellTower(2,3,10);
CellTower c4 = new CellTower(-4,6,9);

obj.insert(c1);
obj.insert(c2);
obj.insert(c3);

obj.cellTowerAt(-2,0);          // returns true
obj.cellTowerAt(2,4);           // returns false
obj.chooseCellTower(0, 6, 5);   // returns c3

obj.insert(c4);
obj.chooseCellTower(0, 6, 5);   // returns c4

// The current tree is shown in Figure 3.

CellTower c5 = new CellTower(-3,7,5);
CellTower c6 = new CellTower(-3,3,4);
CellTower c7 = new CellTower(-6,7,2);
CellTower c8 = new CellTower(-5,4,9);

obj.insert(c5);
obj.insert(c6);
obj.insert(c7);
obj.insert(c8);
obj.insert(c3);                 // should fail

obj.chooseCellTower(-2, 6, 2);  // returns c5

// The current tree is shown in Figure 4.
```

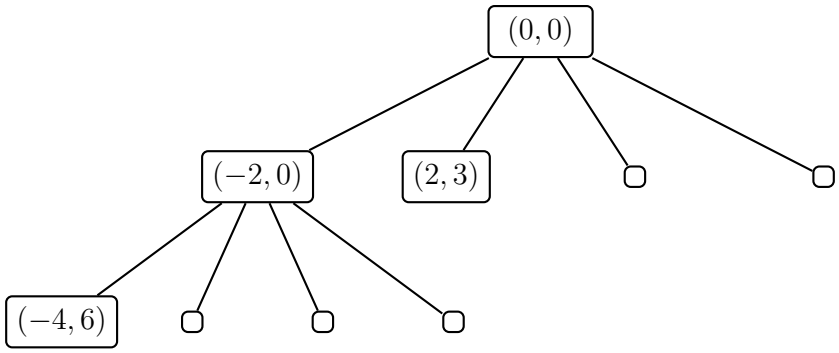

Figure 3: Tree after first four insertions.

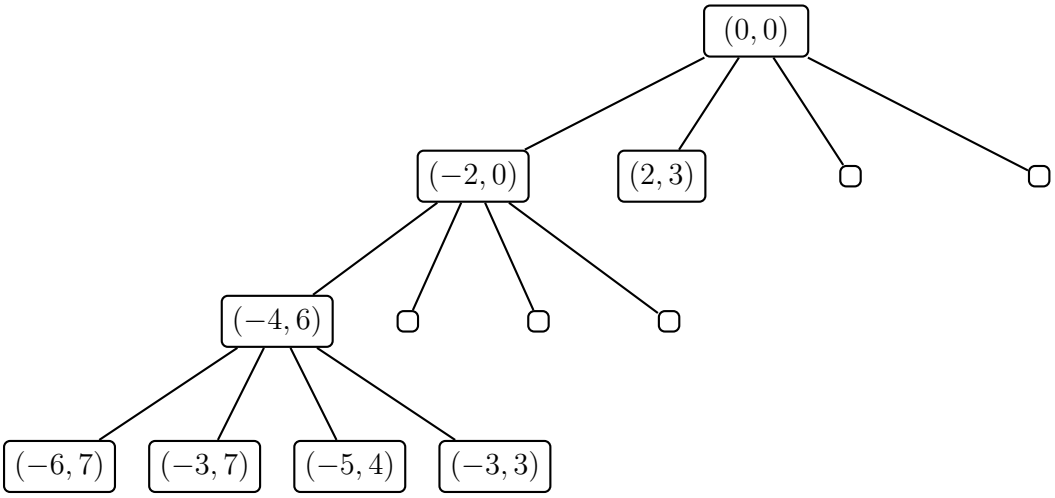

Figure 4: Tree after all the insertions.

2 Optimal Moves in Snake and Ladder Game

Snake and Ladders is a traditional board game that is played by two or more players on a square board with a grid of $n \times n$ cells. The objective of the game is to reach the final cell, which is numbered n^2, before the other players. Each player takes turns rolling a dice to move their game piece along the board. If a player lands on a ladder, they need to advance to a higher-numbered cell, while landing on a snake sends them back to a lower-numbered cell.

In this question, we consider a slightly different problem. Here, you are given a board which consists of 'N' cells and 'M' snakes and ladders. Your objective is to find out the optimal number of moves required to complete the game by a single player.

You should start at cell 1 of the board. In each move, starting from cell '$curr$', choose a destination vertex '$next$' with a label in the range $[curr + 1,\ \min(curr + 6,\ N)]$. If your '$next$' destination has a snake or ladder, you must move to the destination of that snake or ladder. If that destination also has a snake or ladder, you must move to it's destination (you must follow the concatenated sequence of snakes and ladders until you reach a cell where no snake or ladder begins). The game ends when you reach the cell labeled N.

You are given a class `SnakesLadder` which has the following class variables:

- `int N`: Total number of cells on the board.
- `int M`: Total number of snakes and ladders on the board.
- `int snake[]`: An integer array of the length 'N' which will be used to store the endpoints of snakes on the board. For a snake starting at index 'i' and ending at index 'j', $snakes[i] = j$. If no snake starts at cell 'i', $snake[i] = -1$.
- `int ladder[]`: An integer array of the length 'N' which will be used to store the endpoints of ladders on the board. For a ladder starting at index 'i' and ending at index 'j', $ladders[i] = j$. If no ladder starts at cell 'i', $ladder[i] = -1$.

Your task is implement the following four class functions:

- `public SnakeLadder()`: Initializes a board game with 'N' cells and 'M' snakes and ladders.
- `public int OptimalMoves()`: Returns the optimal number of moves required to win the snake and ladder game.
- `public int Query(x, y)`: Returns $+1$ if adding the snake or ladder from x to y will improve the optimal solution, else returns -1.
- `public int[] FindBestNewSnake()`: Finds the best snake, if it exists, whose addition to the board will improve the optimal number of moves by highest possible value.

We next explain these functions in detail.

2.1 Constructor SnakeLadder()

The board is represented by $(M + 2)$ lines, where M is the total number of snakes and ladders in the board. The input will be given in a text file named 'input.txt'. The first line of this file consists of a single integer which is the number of cells on the board. The second line consists of the value of M (total number of snakes and ladders in the board).

Each subsequent line consists of two integers x and y, where x and y respectively denotes the source and destination of a snake or a ladder. If $x < y$, then it is a ladder otherwise it is a snake. For each $x \in [1, N]$, there will be at most one (x, y) entry, i.e. at most one snake/ladder would start from position x.

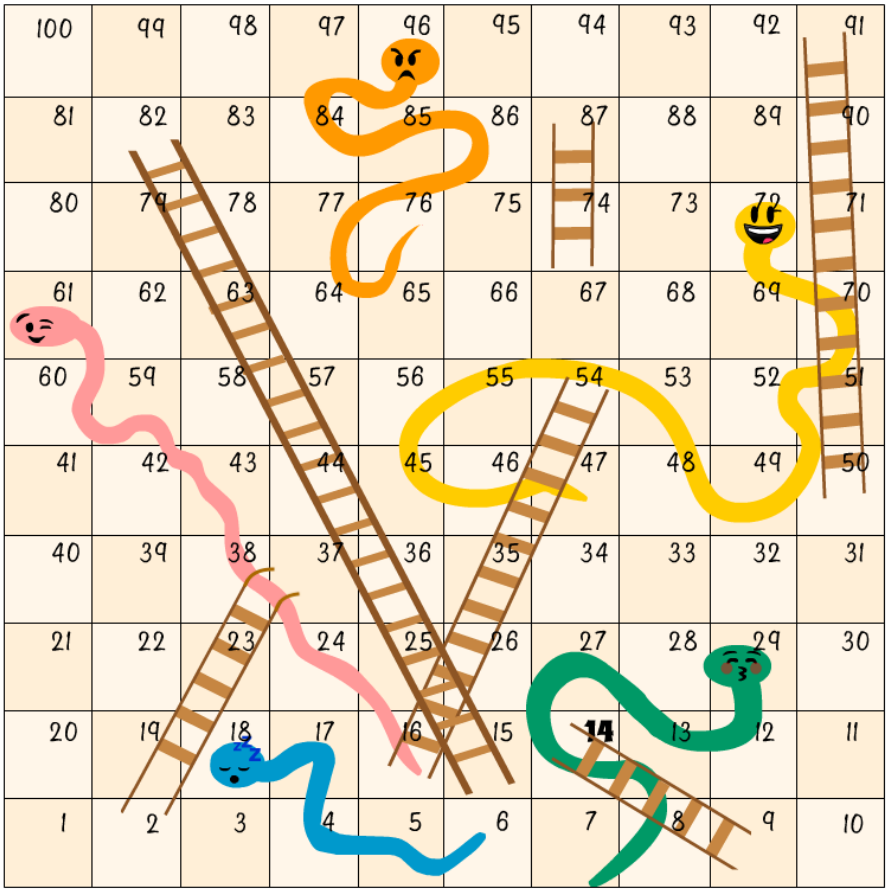

Figure 5: An example of Snake and Ladder Game. Note that on landing on Snake at cell 61 one reaches cell 54 since there is a ladder at cell 16 which must be used to climb to cell 54.

```
Test case: For the snake and ladder board given in above figure,
   the constructor can be initialized as follows:

Input file:
    100          // Number of cells on the board
    11           // Total number of snakes and ladders in the board
    2 37         // Ladder
    16 54        // Ladder
    9 14         // Ladder
    74 87        // Ladder
    15 82        // Ladder
    50 91        // Ladder
    96 76        // Snake
    72 47        // Snake
    29 7         // Snake
    18 6         // Snake
    61 16        // Snake
```

2.2 OptimalMoves()

In this part, you will implement the function OptimalMoves. This function should return the optimal number of moves required to reach the destination cell 'N'. The starting position of the player is outside the board, which means that if the first dice throw results in number 5, the position of player after the first dice throw will be 5.

Your code should return a single integer which is the minimum number of moves required to complete the game or -1 (in case no solution exists).

```
Test case: On calling the function OptimalMoves() for example in
   Figure 5, your code should return the following :

Output : 6

/* The optimal sequence of dice throws will be 2,6,6,1,3,6.

   On getting first two the player reaches cell 2. Since there is
   a ladder on 2, the player advances to cell 37.

   On getting six, six, and one in next three throws, the player
   advances to cell 43, 49, and then 50. At cell 50, the player
   takes the ladder to reach cell 91.

   Next on getting three the player reaches cell 94. On getting
   six in the last dice roll, player reaches the destination. */
```

2.3 Query(x, y)

In this part, you will implement the function Query(x, y). Here, x and y denotes endpoints of a snake or a ladder. If $x < y$, it is a ladder otherwise it is a snake. Query(x, y) should return $+1$ if adding the given snake or ladder improves the optimal solution, otherwise it should return -1.

Note: Do not call the previous function in black-box manner as otherwise it will result in time limit exceeded error. You may do some preprocessing in the constructor itself so that the queries here can be answered efficiently, ideally in constant time.

```
Test case: For the example in Figure 5:

Query(54, 50) should return +1.
// Adding a snake from 54 to 50 will reduce the optimal value
   from 6 to 5.

Query(54, 95) should return +1.
// Adding a ladder from 54 to 95 will reduce the optimal value
   from 6 to 5.

Query(54, 10) should return -1.
// Adding a snake from 54 to 10 will not reduce the optimal value.
```

2.4 FindBestNewSnake()

In this part, you are supposed to find a new snake whose addition to the board will **improve the optimal solution by the largest amount**, if it is possible. Such a snake may exist if there are two overlapping ladders, i.e (x_1, y_1) and (x_2, y_2) with $x_1 < x_2 < y_1 < y_2$.

If such a snake exists, return the array $[x, y]$, where x and y ($x > y$) denotes the endpoints of the snake. Otherwise return an empty array.

Remark: One should not use an $O(N^2)$ time approach for any of the functions in the Snake and Ladder Game as it would result in time limit exceeded error.

```
Test case: On calling the function FindBestNewSnake() for example
   in Figure 5, your code should return the following

Output :
(37,15)

/* The new optimal sequence of dice throws will be 2,6,6,6.

   On getting first two the player reaches cell 2, and takes
   ladder to reach 37. The new snake at 37 will take player to
   cell 15, and from the ladder at 15 the player will reach cell
   82.

   The next 3 occurrences of six in dice throws will take player
   to positions 88, 94, and finally 100. */
```