CSCI 1110: Assignment 1

Due: 11:59pm, Sunday, October 2, 2022

The purpose of this assignment is to provide you programming practice with loops and arrays in Java. For each problem you will be provided with sample inputs and corresponding outputs. All input is done via the console (unless specified otherwise) and all output is to be done to the console as well. You must submit your assignment via Codio and you can test it by clicking the Check It button.

Figure 1: Example of a game board https://toytheater.com/ wp-content/up-loads/snakes_and_ladders.gif (Retrieved on May 6, 2019)

Background

Snakes and Ladders is a children's board game that originated in India[1]. Players start at square 1 and take turns rolling moving on the board. On their turn, a player rolls a dice and moves the number of spaces rolled. If they end up at the foot of a ladder, they move up the ladder to the top. If they end up on the head of a snake, they slide down the snake to the tail. The first player to reach the last square on the board, wins.

For the first part of this assignment, your task will be to implement a game simulator that plays Snakes and Ladders. For the second part of this assignment, you will write a program that determines whether a game board is winnable. I.e., whether it is possible for a player to win.

A Snakes and Ladders game board can be best represented by an array of integers, where a 0 denotes a regular square on a board, a positive value denotes the foot of a ladder, and a negative value denotes the head of a snake. The actual nonnegative value (positive or negative) can either be the end location of the snake or ladder, or the number of squares that the player moves. Both representations work equally well.

Problem I: Childs Play

For this problem you are to write a program that simulates a Snakes and Ladders game between two or more opponents.

Write the body of the program called **`Problem1.java`** that reads in (i) a board configuration, (ii) the names of the players, and (iii) a sequence of dice rolls, and outputs the game-play.

Input

The input consists of three parts: board configuration, players, and dice rolls. The board configuration has the following format:

- An integer *B* denoting the size of the board. E.g., for the figure above, *B* would be 100.
- An integer *W*, denoting the total number of snakes and ladders in the board. E.g., for the figure above, *W* would be 18 (8 ladders and 10 snakes).
- *W* lines where each line is of the form "**Type Start End**" where

  Type is either `SNAKE` or `LADDER`

[1] "Chutes and Ladders - Snakes and Ladders", About.com.

Start is the position of the head of a snake or the foot of a ladder, respectively

End is the position of the tail of a snake or the top of a ladder, respectively

For example,

```
LADDER 28 84
SNAKE 87 24
```

denotes the light blue ladder and the green snake in the figure above.

The player information consists of

- An integer *P* denoting the number of players playing the game, where $P \geq 2$
- Followed by *P* lines, where each line contains a single word denoting the name of a player

The dice rolls consist of

- An integer *D* denoting the number of dice rolls
- Followed by *D* lines, where each line contains a single integer in the range 1 . . . 6

See the example below.

Processing

Your program should simulate a game of Snakes and Ladders using the board, player, and dice roll information provided in the input.

- The board size can range from 10 to 1000.
- Each square on the board will be one of the following: (i) empty, (ii) a foot of a ladder, (iii) a top of ladder, (iv) a head of snake, or (v) a tail of the snake.
- The number of players will range from 2 to 1000. All players start on square 1 of the board.
- The players play in order that they are listed in the input
- A player finishes the game if they move past the last square on the board.
- If a player finishes the game, they are removed from play.
- Play continues until there are no more dice rolls or no more players.

Example

Input	Output
10 4 SNAKE 9 1 LADDER 4 8 LADDER 2 5 SNAKE 7 3 3 Alice Bob Carol 9 3 5 2 6 3 4 5 1 5	Alice 1 8 Bob 1 6 Carol 1 3 Alice 8 * Bob 6 1 Carol 3 3 Bob 1 6 Carol 3 8 Bob 6 *

Output

For each player's turn, print out "**Name Current New**" where

Name is the player's name

Current is the player's current position (an integer)

New is the player's new position after they moved. This is an integer if the player has not completed the game or a * if they have.

For example, if the game is being played on the board in the image above, it is Alice's turn, she is on square 25, and she rolls a 3, then the output for her turn would be:

```
Alice 25 84
```

Problem 2: Winnable?

Suppose you wanted to create a program to generate Snakes and Ladders game boards. Your approach is to create a game board with N squares and then randomly put snakes and ladders on the board. However, before releasing the board you need to determine if the board is winnable. I.e., Is it possible for a player to win?

Write the body of the program called **`Problem2.java`** that reads in a board configuration in the same format as Problem 1 and determines whether or not there exists a sequence of dice rolls that would result in a win for a player.

Input

The same format as the board configuration in Problem 1.

Processing

Determine all positions that are not reachable on the board. Same constraints apply as in Problem 1.

Suggested Approach:

- Starting from position 1, mark all positions that are reachable from position 1.
- Then, mark all positions reachable from those positions, etc.
- A position *Q* is reachable from position *P* if a roll of the dice, would result in a player on position *P* ending up on position *Q*.
- For example, a square with a snake's head or the foot of a ladder is never reachable because the player slides down the snake or climbs up the ladder if they land on that square.

Output

The program should print output a single line containing all positions that are NOT reachable on the board, in sorted order. If the board is not winnable, the * position should be the last position to be printed.

Examples

Example 1: Winnable Board		**Example 2: Nonwinnable Board**	
Input	**Output**	**Input**	**Output**

`10`
`4`
`SNAKE 9 1`
`LADDER 4 8`
`LADDER 2 5`
`SNAKE 7 3`
	`2 4 7 9`	`20`
		`8`
		`SNAKE 15 3`
		`SNAKE 17 11`
		`LADDER 2 6`
		`SNAKE 12 4`
		`SNAKE 14 5`
		`LADDER 18 19`
		`SNAKE 13 10`
		`SNAKE 16 7`
			`2 12 13 14 15 16 17 18 19 20 *`

Hints and Things to Remember

- If you code does not compile, you will receive a 0 on that portion of the assignment.
- Your code must be well commented and indented. Please see the (Assignments section in Brightspace) for Code Style Guidelines.
- All input will be correct. You do not need to handle incorrect input, for now.
- All the problems in this assignment have short solutions (≈ 60 lines of code).

What to Hand In

This assignment must be submitted in Codio via the Brightspace page.

Grading

The assignment will be graded based on three criteria:

Functionality: "Does it work according to specifications?". This is determined in an automated fashion by running your program on a number of inputs and ensuring that the outputs match the expected outputs. The score is determined based on the number of tests that your program passes. So, if your program passes t/T tests, you will receive that proportion of the marks.

Quality of Solution: "Is it a good solution?" This considers whether the approach and algorithm in your solution is correct. This is determined by visual inspection of the code. It is possible to get a good grade on this part even if you have bugs that cause your code to fail some of the tests.

Code Clarity: "Is it well written?" This considers whether the solution is properly formatted, well documented, and follows coding style guidelines. A single overall mark will be assigned for clarity. Please see the Java Style Guide in the Assignment section of the course in Brightspace.

If your program does not compile, it is considered non-functional and of extremely poor quality, meaning you will receive 0 for the solution.

The following grading scheme will be used:

<table>
<tr><th>Task</th><th>100%</th><th>80%</th><th>60%</th><th>40%</th><th>20%</th><th>0%</th></tr>
<tr><td>Functionality
(20 marks)</td><td colspan="5">Equal to the number of tests passed.</td><td rowspan="3">No code submitted or code is not a reasonable</td></tr>
<tr><td>Solution Quality
(20 marks)</td><td>Approach and algorithm are appropriate to meet requirements for Problem 1 and 2.</td><td>Approach and algorithm are appropriate to meet requirements for Problem 1, and some requirements for Problem 2.</td><td>Approach and algorithm are appropriate to meet requirements for Problem 1 but not Problem 2.</td><td>Approach and algorithm will meet some of the requirements of Problem 1.</td><td>Reasonable attempt made at solution for Problem 1.</td></tr>
<tr><td>Code Clarity
(10 marks)</td><td colspan="5">As outlined below</td></tr>
</table>

Code Clarity (out of 10)

- 2 marks for identification block at the top of code
- 2 marks for appropriate indentation when necessary
- 2 marks for use of blank lines to separate unrelated blocks of code
- 2 marks for comments throughout the code as needed
- 2 marks for consistent coding style throughout