32. SNAKE AND LADDER GAME

The application should take as input (from the command line or a file):

- Number of snakes (s) followed by s lines each containing 2 numbers denoting the head and tail positions of the snake.
- Number of ladders (l) followed by l lines each containing 2 numbers denoting the start and end positions of the ladder.
- Number of players (p) followed by p lines each containing a name.

After taking these inputs, you should print all the moves in the form of the current player name followed by a random number between 1 to 6 denoting the die roll and the initial and final position based on the move.

Format: <player_name> rolled a <dice_value> and moved from <initial_position> to <final_position>

When someone wins the game, print that the player won the game.

Format: <player_name> wins the game

Rules of the game

- The board will have 100 cells numbered from 1 to 100.
- The game will have a six sided dice numbered from 1 to 6 and will always give a random number on rolling it.
- Each player has a piece which is initially kept outside the board (i.e., at position 0).
- Each player rolls the dice when their turn comes.
- Based on the dice value, the player moves their piece forward that number of cells. Ex: If the dice value is 5 and the piece is at position 21, the player will put their piece at position 26 now (21+5).

- A player wins if it exactly reaches the position 100 and the game ends there.
- After the dice roll, if a piece is supposed to move outside position 100, it does not move.
- The board also contains some snakes and ladders.
- Each snake will have its head at some number and its tail at a smaller number.
- Whenever a piece ends up at a position with the head of the snake, the piece should go down to the position of the tail of that snake.
- Each ladder will have its start position at some number and end position at a larger number.
- Whenever a piece ends up at a position with the start of the ladder, the piece should go up to the position of the end of that ladder.
- There could be another snake/ladder at the tail of the snake or the end position of the ladder and the piece should go up/down accordingly.

Assumptions you can take apart from those already mentioned in rules

- There won't be a snake at 100.
- There won't be multiple snakes/ladders at the same start/head point.
- It is possible to reach 100, i.e., it is possible to win the game.
- Snakes and Ladders do not form an infinite loop.

Optional Requirements

Please do these only if you've time left. You can write your code such that these could be accommodated without changing your code much.

The game is played with two dice instead of 1 and so the total dice value could be between 2 to 12 in a single move.

The board size can be customizable and can be taken as input before other input (snakes, ladders, players).

In case of more than 2 players, the game continues until only one player is left.

On getting a 6, you get another turn and on getting 3 consecutive 6s, all the three of those get cancelled.

On starting the application, the snakes and ladders should be created programmatically without any user input, keeping in mind the constraints mentioned in rules.

There could be different ways to design a solution for this. I'll mention how I would have approached it during an actual interview.

Let's dissect the problem statement to determine how to design a good solution for it.

The gist of the problem statement is that we need to create a snake and ladder game. The game will have multiple snakes and ladders and there will be multiple people playing the game. The game is played automatically based on certain rules and ends when a player wins.

Here I've decided that I'll keep a Driver class to take input from the user and a SnakeAndLadderService to orchestrate the game.

In the interview, I'll not focus on solving with the optional requirements but will try to keep my design such that if I've time, I can do minimal changes to add them. Let's look at each of those requirements one-by-one and see which ones I would want to ignore to keep my design clean. The problem is supposed to be solved in 2 hours and so I do not want to get stuck in something that is time-consuming but optional.

For optional requirement #1, I need to keep an option to do multiple dice rolls. Looks manageable!

For optional requirement #2, the board should be initializable with a size as well and the game should end based on that.

For optional requirement #3, I can pass whether the game should continue until the last player or not to the SnakeAndLadderService.

For optional requirement #4, I can pass whether there should be another turn on getting a 6 or not to the SnakeAndLadderService.

For optional requirement #5, this can be done as part of the Driver class and does not influence the core design.

So, I can keep my code extensible for all the 5 optional requirements!

Then, I'll note down the entities (models) that will be involved in the design.

- Player
- Snake
- Ladder
- SnakeAndLadderBoard

Then, I'll create the models for all of these.

Snake.java

```
package com.workattech.snl.models;


public class Snake {

    // Each snake will have its head at some number and its tail at a smaller number.

    private int start;

    private int end;
    public Snake(int start, int end) {

        this.start = start;

        this.end = end;

    }


    public int getStart() {

        return start;

    }


    public int getEnd() {

        return end;

    }

}
```

Ladder.java

```
package com.workattech.snl.models;


public class Ladder {

    // Each ladder will have its start position at some number and end position at a larger number.

    private int start;

    private int end;


    public Ladder(int start, int end) {

        this.start = start;

        this.end = end;
    }


    public int getStart() {

        return start;

    }


    public int getEnd() {

        return end;

    }

}
```

Player.java

```
package com.workattech.snl.models;

import java.util.UUID;

public class Player {

    private String name;

    private String id;

    public Player(String name) {

        this.name = name;

        this.id = UUID.randomUUID().toString();

    }

    public String getName() {

        return name;

    }

    public String getId() {

        return id;
        }
}
```

SnakeAndLadderBoard.java

```
package com.workattech.snl.models;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
public class SnakeAndLadderBoard {
        private int size;
        private List<Snake> snakes; // The board also contains some snakes and ladders.
        private List<Ladder> ladders;
        private Map<String, Integer> playerPieces;
        public SnakeAndLadderBoard(int size) {
                this.size = size;
                this.snakes = new ArrayList<Snake>();
                this.ladders = new ArrayList<Ladder>();
                this.playerPieces = new HashMap<String, Integer>();
        }
        public int getSize() {
                return size;
        }
        public List<Snake> getSnakes() {
                return snakes;
        }
```

```
    public void setSnakes(List<Snake> snakes) {

        this.snakes = snakes;

    }

    public List<Ladder> getLadders() {

        return ladders;

    }

    public void setLadders(List<Ladder> ladders) {

        this.ladders = ladders;

    }

    public Map<String, Integer> getPlayerPieces() {

        return playerPieces;

    }

    public void setPlayerPieces(Map<String, Integer> playerPieces) {

        this.playerPieces = playerPieces;

    }

}
```

Now that I have created the models, let me create the services which will be used to talk to the models and to each other.

Here, I'm creating the DiceService which for now just has a method to roll the dice and get the value.

DiceService.java

```
package com.workattech.snl;

import java.util.Random;

public class DiceService {
        public static int roll() {

                return new Random().nextInt(6) + 1; // The game will have a six sided dice numbered from
1 to 6 and will always give a random number on rolling it.

        }

}
```

After that, I'll create the SnakeAndLadderService which will be the orchestrator of the entire game.

SnakeAndLadderService.java

```
package com.workattech.snl;

import com.workattech.snl.models.*;

import java.util.*;

public class SnakeAndLadderService {
        private SnakeAndLadderBoard snakeAndLadderBoard;
        private int initialNumberOfPlayers;
        private Queue<Player> players; // Comment: Keeping players in game service as they
are specific to this game and not the board. Keeping pieces in the board instead.
        private boolean isGameCompleted;

        private int noOfDices; //Optional Rule 1
        private boolean shouldGameContinueTillLastPlayer; //Optional Rule 3
        private boolean shouldAllowMultipleDiceRollOnSix; //Optional Rule 4

        private static final int DEFAULT_BOARD_SIZE = 100; //The board will have 100 cells
numbered from 1 to 100.
        private static final int DEFAULT_NO_OF_DICES = 1;

        public SnakeAndLadderService(int boardSize) {
                this.snakeAndLadderBoard = new SnakeAndLadderBoard(boardSize);    //Optional
```

Rule 2

```
		this.players = new LinkedList<Player>();
		this.noOfDices = SnakeAndLadderService.DEFAULT_NO_OF_DICES;
	}

	public SnakeAndLadderService() {
		this(SnakeAndLadderService.DEFAULT_BOARD_SIZE);
	}

	/**
	 * ====Setters for making the game more extensible====
	 */

	public void setNoOfDices(int noOfDices) {
		this.noOfDices = noOfDices;
	}

	public void setShouldGameContinueTillLastPlayer(boolean
shouldGameContinueTillLastPlayer) {
		this.shouldGameContinueTillLastPlayer = shouldGameContinueTillLastPlayer;
	}

	public void setShouldAllowMultipleDiceRollOnSix(boolean
shouldAllowMultipleDiceRollOnSix) {
		this.shouldAllowMultipleDiceRollOnSix = shouldAllowMultipleDiceRollOnSix;
	}

	/**
	 * =================Initialize board==================
	 */

	public void setPlayers(List<Player> players) {
		this.players = new LinkedList<Player>();
		this.initialNumberOfPlayers = players.size();
```

```
        Map<String, Integer> playerPieces = new HashMap<String, Integer>();
        for (Player player : players) {
            this.players.add(player);
            playerPieces.put(player.getId(), 0); //Each player has a piece which is initially
kept outside the board (i.e., at position 0).
        }
        snakeAndLadderBoard.setPlayerPieces(playerPieces); //    Add pieces to board
    }

    public void setSnakes(List<Snake> snakes) {
        snakeAndLadderBoard.setSnakes(snakes); // Add snakes to board
    }

    public void setLadders(List<Ladder> ladders) {
        snakeAndLadderBoard.setLadders(ladders); // Add ladders to board
    }

    /**
     * ==========Core business logic for the game==========
     */

    private int getNewPositionAfterGoingThroughSnakesAndLadders(int newPosition) {
        int previousPosition;

        do {
            previousPosition = newPosition;
            for (Snake snake : snakeAndLadderBoard.getSnakes()) {
                if (snake.getStart() == newPosition) {
                    newPosition = snake.getEnd(); // Whenever a piece ends up at a
position with the head of the snake, the piece should go down to the position of the tail of
that snake.
                }
            }
            for (Ladder ladder : snakeAndLadderBoard.getLadders()) {
                if (ladder.getStart() == newPosition) {
                    newPosition = ladder.getEnd(); // Whenever a piece ends up at a position with the start of the ladder, the piece should go up to the position of the end of that ladder.
                }
            }
        } while (newPosition != previousPosition); // There could be another snake/ladder at the tail of the snake or the end position of the ladder and the piece should go up/down accordingly.
        return newPosition;
    }


    private void movePlayer(Player player, int positions) {
        int oldPosition = snakeAndLadderBoard.getPlayerPieces().get(player.getId());
        int newPosition = oldPosition + positions; // Based on the dice value, the player moves their piece forward that number of cells.


        int boardSize = snakeAndLadderBoard.getSize();


        // Can modify this logic to handle side case when there are multiple dices (Optional requirements)
        if (newPosition > boardSize) {
            newPosition = oldPosition; // After the dice roll, if a piece is supposed to move outside position 100, it does not move.
        } else {
            newPosition = getNewPositionAfterGoingThroughSnakesAndLadders(newPosition);
        }


        snakeAndLadderBoard.getPlayerPieces().put(player.getId(), newPosition);


        System.out.println(player.getName() + " rolled a " + positions + " and moved from " + oldPosition +" to " + newPosition);
    }
```

```
	private int getTotalValueAfterDiceRolls() {
		// Can use noOfDices and setShouldAllowMultipleDiceRollOnSix here to get total
value (Optional requirements)
		return DiceService.roll();
	}

	private boolean hasPlayerWon(Player player) {
		// Can change the logic a bit to handle special cases when there are more than
one dice (Optional requirements)
		int playerPosition = snakeAndLadderBoard.getPlayerPieces().get(player.getId());
		int winningPosition = snakeAndLadderBoard.getSize();
		return playerPosition == winningPosition; // A player wins if it exactly reaches the
position 100 and the game ends there.
	}

	private boolean isGameCompleted() {
		// Can use shouldGameContinueTillLastPlayer to change the logic of determining if
game is completed (Optional requirements)
		int currentNumberOfPlayers = players.size();
		return currentNumberOfPlayers < initialNumberOfPlayers;
	}

	public void startGame() {
		while (!isGameCompleted()) {
			int totalDiceValue = getTotalValueAfterDiceRolls(); // Each player rolls the
dice when their turn comes.
			Player currentPlayer = players.poll();
			movePlayer(currentPlayer, totalDiceValue);
			if (hasPlayerWon(currentPlayer)) {
				System.out.println(currentPlayer.getName() + " wins the game");
				snakeAndLadderBoard.getPlayerPieces().remove(currentPlayer.getId());
			} else {
				players.add(currentPlayer);
			}
		}
```

```
    }

    /**
     * ===========================================================
     */
}
```

Note: I could have optionally extracted PlayerService from SnakeAndLadderService as well and moved some dice related logic to DiceService but without the optional requirements, I felt that this is also fine. I could also have used interfaces at certain places but I avoided it here for simplicity.

Now, that our core design is done, let's create the Driver class where we would take user input followed by initializing and executing the SnakeAndLadderService.

Driver.java

```
package com.workattech.snl;

import com.workattech.snl.models.Ladder;
import com.workattech.snl.models.Player;
import com.workattech.snl.models.Snake;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class Driver {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int noOfSnakes = scanner.nextInt();
        List<Snake> snakes = new ArrayList<Snake>();
        for (int i = 0; i < noOfSnakes; i++) {
            snakes.add(new Snake(scanner.nextInt(), scanner.nextInt()));
        }

        int noOfLadders = scanner.nextInt();
        List<Ladder> ladders = new ArrayList<Ladder>();
        for (int i = 0; i < noOfLadders; i++) {
            ladders.add(new Ladder(scanner.nextInt(), scanner.nextInt()));
        }

        int noOfPlayers = scanner.nextInt();
        List<Player> players = new ArrayList<Player>();
        for (int i = 0; i < noOfPlayers; i++) {
            players.add(new Player(scanner.next()));
        }

        SnakeAndLadderService snakeAndLadderService = new SnakeAndLadderService();
        snakeAndLadderService.setPlayers(players);
        snakeAndLadderService.setSnakes(snakes);
        snakeAndLadderService.setLadders(ladders);

        snakeAndLadderService.startGame();
    }
}
```

So, this is how I would have designed Snake and Ladder during an actual machine coding round for the given problem statement. This may not be a perfect solution but I guess it would be good enough to clear the interview given that there are only two hours to solve this.