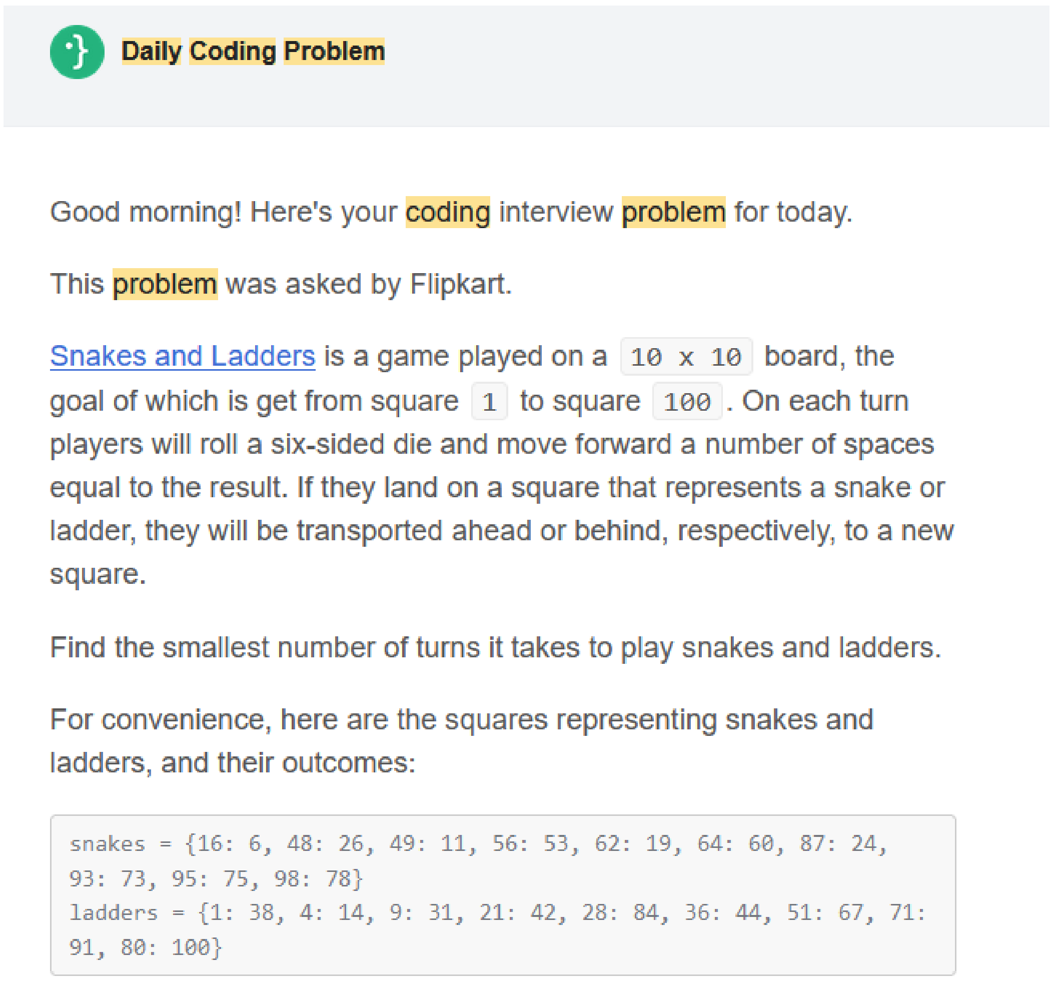

Daily Coding Problem

Good morning! Here's your coding interview problem for today.

This problem was asked by Flipkart.

Snakes and Ladders is a game played on a `10 x 10` board, the goal of which is get from square `1` to square `100`. On each turn players will roll a six-sided die and move forward a number of spaces equal to the result. If they land on a square that represents a snake or ladder, they will be transported ahead or behind, respectively, to a new square.

Find the smallest number of turns it takes to play snakes and ladders.

For convenience, here are the squares representing snakes and ladders, and their outcomes:

```
snakes = {16: 6, 48: 26, 49: 11, 56: 53, 62: 19, 64: 60, 87: 24,
93: 73, 95: 75, 98: 78}
ladders = {1: 38, 4: 14, 9: 31, 21: 42, 28: 84, 36: 44, 51: 67, 71:
91, 80: 100}
```

*** INITIAL THOUGHTS ****

I am yet to use a 3D array in any of my solutions.
It might be my lack of understanding, I have created one now.

```
int [][][] test = new int [][][] {   //first row  can have multiple arrays(of various sizes)...
                        { {1,2},  {3,4,5}          },
                        { {6}, {7,8,9},  {0}        }


                       };
```

```
System.out.println("Practice getting values out:");
System.out.println("should print 1: " + test[0][0][0]);
System.out.println("should print 2: " + test[0][0][1]);
System.out.println("should print 3: " + test[0][1][0]);
System.out.println("should print 4: " + test[0][1][1]);
System.out.println("should print 5: " + test[0][1][2]);
System.out.println("should print 5: " + test[1][0][0]);
System.out.println("should print 7: " + test[1][1][0]);
System.out.println("should print 8: " + test[1][1][1]);
System.out.println("should print 9: " + test[1][1][2]);
System.out.println("should print 0: " + test[1][2][0]);
```

I am still unsure of how content will be stored here.. My initial logic is:

[1] [ ] [ ] - this might be if a 1 is thrown on the dice...
now there would be several arrays {startPos, endPos}, {startPos, endPos}

We can potentially deduce that for int [1][][] on a 10 x 10 board, there is potentially a 1 thrown on the dice from each location (with exception of 100) since this is the endpoint.
We also know that 1 can not be thrown from any location where there is a ladder climb or point to which the snake slips...

It is worth drawing the board to bring this logic together
It will definitely involve recursion again... since it will involve moving across board while notAtEnd
There has to be combination involving C(6,1) from each grid except four scenarios scenarios:
1) If square leads to a ladder climb.
2) If the square has Snake head.
3) where end user has reached 100 (finish point)
4) Where the move is not viable such as throwing a 6 whilst on 95...

To simplify the entirety based on validity of the above:

[1-6] [ ] [ ] - this would be dice value thrown

```
{
{startPos,  endPos   }    { diceValue, endPos    },


}
```

Each StartPos will generate endpoint (both stored in same subset). The endPoint will then search through the entire 3D array for other identical startPos.... And perform a roll to reach endPoint... The cycle will continue..... In terms of combinations, it has issues recording primitive data types in Set and all arrays.. It will keep a String keep of the transactions to avoid duplication.

Another alternative for consolidating all logic would be:

[startPos] [ ] [ ] - this might be startPos on the grid between 1-99 (excluding squares with Snake Head or base of ladder)...

```
{
{diceValue,  endPos   }    { diceValue, endPos    },


}
```

Not entirely sure which would be better option. The one above seems suitable to meet more validations...

I have now finished the challenge. I am now at a point where it completes the board and also it continues process until it fills every possible valid move from every grid position. But there are struggles in validating exactly when it has been reached…

But strictly speaking, this does not answer nature of the challenge. It states smallest number of turns.
In principle, without intervention of the snakes/ladders it would be 16 x 6 = 96
1 x 4 = 100 **(17 moves total)**

I will now perform a simulation which visually strike out moving across ladders and snakes based on my observations of the data.

Throwing 1 (moving 1 to 38)
But it can be observed immediately (moving 28 to 84) was not considered.
So is the cost of turns traversing to 28 worthwhile for a more rewarding climb.

It has to be observed for all the routes to 28 now

This in itself raises several questions:
The simple option is throwing 4 x 6 and then 1 x 4 (5 throws). This will then ladder (28,84). Now there is no snake nearby to get back to 80 and perform Snake(80,100). So best route is obtaining 16 (6,6,4).... So this is:
8 moves total.

Alternative option is throwing 1 x 1 resulting in climb to (1,38). Throwing (5 and 5) or (6 and 4), this will then result in Snake(48,26).
Then throwing a 2 Ladder(28,84). (4 throws). As above, the best route is obtaining 16 (6,6,4).... So this is: **7 moves total.**
So this shows how a snake can be beneficial on the cost.

It is difficult to ascertain if there are any further cost savings getting to 28. But it would be worthwhile exploring passing through 38.

Throwing 1 x 1   Ladder(1,38)
Throwing (6,1,6) Ladder (51,67)
Throwing 4   Ladder (71,91)
Throwing (6,3)   reach end

7 total

Throwing 1 x 1   Ladder(1,38)
Throwing (6,1,6) Ladder (51,67)
Throwing 6,6,1 or 6,4,3 or 6,5,2 or 6,3,3   Ladder (80,100)

7 total

It appears only outstanding technique to complete this problem is as follows:

* I have officially recorded all viable moves

* Perform a C(total number snakes and ladders=20, r)
It would select the values from enum PositionsSnakesLadders)

So range would be   C(20,0)   to C(20,20)...   It appears that all of these are computational....

We already know that landing on more than 7 snakes or ladders will be more expensive than calculation above. However this was ascertained via manually performing calculations. Code would have no awareness.
For efficiency, the code execute to next combination if totalMoves is greater than existing minimum...

A solution is that number turns can not exceed 17, since this is route bypassing all snakes and ladders...
But this is hypothetical since it assumes that at:
grid 6, grid 12, grid 18.... up to grid 96 (there are no snakes or ladders interfering).

So whilst it moves across the board, for instance if the combination is {SNAKE1, LADDER2, SNAKE3, LADDER5}, it would move between those locations ONLY, land on those locations EXACTLY (in that order), accepting the highest thrown die values where possible.

AND it would only land on those snakes or ladders. If the die throws a value that would land on another intermediate one (not part of the combination above) such as SNAKE6, the die would be thrown again.

So it would perform:

** CAN BE PERFORMED RECURSION **

(snake/ladder (startPos) **-** currentPosition)

This would be the amount of moves required... It will keep throwing a six to see if it can fit before snake(startPos).
It has to ensure that throwing a six does not land on an intermediate snake/ladder, otherwise it will fault the outcome... If it interferes, the next highest die value is considered.
It has to also be considered that if a snake/ladder (startPos, endPos) is incurred, the endPos has to be less than the subsequent ladder/snake (startPos). Otherwise it will be impossible to reach it.... In this case, it would discard this combination...

The remainder left will be moved forward to reach Snake/ladder(startPos) (as per the combination above).

** CAN BE PERFORMED RECURSION **

Once it reaches the last destination LADDER5(endPos), it would then perform same operations of throwing highest dice value (as long as it does not interfere with other snakes and ladders), UNTIL it reaches grid 100 EXACTLY.

NOTE:

There is no longer a requirement to use the random number.
I introduced this to ensure it replicated a real world simulated scenario first. For instance, does the design warrant playing..
And also in the code output, it demonstrates the array deepString for every dice roll undertaken on each grid.. Any voids itself (apart from grids with snake head or ladder base) is indicator that it simply is not possible to exercise every move readily.

So in this respect, I will simulate the moves as per the algorithm above to derive the best combination..

*** OUTPUT SO FAR (RANDOM SIMULATION) ****

See attachment

It appears that:

*******ANALYSIS OF ALL THE MOVES UNDERTAKEN [Start Position => End Position]******

Minimum turns is: 7

Maximum turns is: 177

Average turns is: 37