

# CS 213 Project Report

Arpan Banerjee - 150070011  
Arvind Singh Arya - 150070042  
Nihal Singh - 150040015  
Srivatsan Sridhar - 150070005

## Snakes and Ladders

April 26, 2017

Link to Code- [https://github.com/nihal111/DSA\\_Project](https://github.com/nihal111/DSA_Project)

### Problem Statement

To accept a standard 100 block snakes and ladder table and find the quickest way up. Display the shortest number of dice rolls required to reach "100" along with the path taken. The minimum number of die rolls taken would always be a unique number, meaning no smaller paths possible. However the shortest paths taken would be non-unique as multiple paths having the same number of dice rolls might be possible.

### Input Format

The first line contain N(Number of ladders) and after that N lines follow. Each of the N line contain 2 integer representing the starting point and the ending point of a ladder respectively.

The next line contain integer M(Number of snakes) and after that M line follow. Each of the M line contain 2 integer representing the starting point and the ending point of a snake respectively.

### Output Format

Display the minimum number of steps required. Also, display the path to get there along with the dice rolls required.

### Sample Input

For a board having 4 ladders and 9 snakes, with bottom and top for each defined subsequently.

```
4
8 52
6 80
26 42
2 72
9
51 19
39 11
37 29
81 3
59 5
79 23
53 7
43 33
77 21
```

### Sample Output

For a board having 5 as the minimum possible steps to reach 100. For the path being 0 to 80 (dice roll 6) to 86 (dice roll 6) to 92 (dice roll 6) to 98 (dice roll 6) to 100 (dice roll 2).

```
Shortest path is 5 steps
0 --[6]-> 80 --[6]-> 86 --[6]-> 92 --[6]-> 98 --[2]-> 100
```

## Description of the Algorithms used

### 1. [Algorithm 1](#) (a modified Dijkstra implementation)

This algorithm moves in the order of number of steps required to reach the squares. We maintain a list in which we add each square as we visit it. The list starts with the square '0'. Then we add to the list, all squares that can be reached from 0 in one step, i.e. squares 1 to 6 or tops of any ladders that may originate from these squares. We proceed then by adding to the list all squares that can be reached from each of the squares in the list, i.e. squares that can be reached in one step from '1' and so on. We stop when 100 is visited for the first time.

The property of the algorithm is that when each square is visited for the first time, we have the shortest path to that square from 0, and the shortest path to '100' must go through one of the shortest paths. The algorithm is essentially similar to Dijkstra's algorithm because at each stage we explore the node that can be reached in the shortest number of steps. In this case, the edges (dice throws) are unweighted, so the nodes are explored in a BFS manner.

#### **Implementation:**

A list called 'board' stores which square each index actually leads to. If there is a ladder or snake from  $x$  to  $y$  then  $\text{board}[x] = y$ ,  $\text{board}[x] = x$  otherwise. This array is populated at the beginning by reading the input file. There is another list 'isCalled' which maintains whether each square has been visited or not. A list called 'levels' contains the squares that have been visited, the squares along the shortest path and the sequence of dice throws required in that path.

In this case, although the problem really involves a graph, the implementation of the data structure is not very complex. This is because the nodes and vertices of the graph are implicit (other than the snakes and ladders), so it does not require additional storage. Only the array 'board' is required to store the structure of the graph.

**Pseudocode:**

```

Set board[i] = i for i = 0 to 100
For each ladder in the input file,
    Set board[bottom_of_ladder] = top_of_ladder
For each snake in the input file,
    Set board[mouth_of_snake] = tail_of_snake
Set isCalled[i] = 0 for i = 0 to 100
Set levels[] = [ (0, [0], []) ]
For each square index in levels[]:
    For x = index+1 to index+6
        If x<= 100 and isCalled[board[x]] = 0 then:
            Set isCalled[board[x]] = 1
            Add board[x] to the path to x
            Add x-index to the sequence of dice throws
            Add (board[x], path, dice throws) to
neighbours[]
            If board[x] = 100 then
                Print the path and dice throws and exit
            Add neighbours[] to levels[]

```

## 2. [Algorithm 2](#) (Calculating shortest path for each square sequentially)

In this algorithm, we calculate the shortest path to each square sequentially from 1 to 100. The core idea is that shortest path to any square is the minimum of the already calculated path, and the shortest paths to the six squares before it plus one. Initially, the shortest paths to the squares '1' to '6' (or the tops of ladders originating from them) are set to 1. The shortest paths to all other squares are initially set to 101, so that the first valid path found will be shorter than it. The algorithm then calculates the shortest path from 7 to 100. In addition, for each square we also store the previous square through which it must reach this square for the shortest path. This will help us retrace the shortest path taken to reach 100.

This algorithm works because before reaching any square, the shortest path to squares before it have been computed. However, if a snake head is reached and the shortest path to its tail is changed, we need to update the shortest path of all squares between its tail and head.

### Implementation:

This algorithm is implemented much more easily. It requires the array 'board' to keep the snakes and ladders. Two arrays 'minStepsTo' and 'prevSquareFor' are required to store the computed shortest length and the square to come from for the shortest path.

### Pseudocode:

```
Set board[i] = i for i = 0 to 100
For each ladder in the input file,
    Set board[bottom_of_ladder] = top_of_ladder
For each snake in the input file,
    Set board[mouth_of_snake] = tail_of_snake
Set minStepsTo[i] = 101 for i = 1 to 100 and 0 for i = 0
Set prevSquareFor[i] = (-1,-1) for i = 0 to 100
Set minStepsTo[board[i]] = 1 for i = 1 to 6
```

```
Set prevSquareFor[board[i]] = (0,sq) for i = 1 to 6
For squares sq = 7 to 100:
  Set changed = 0
  For step = 1 to 6:
    If a better path is found through square (sq - step):
      Update minStepsTo[board[sq]] and prevSquareFor[board[sq]]
    If changed = 1 then
      calculate minStepsTo and prevSquare for all squares
      between
        snake tail and snake head
Print the minimum throws and path to 100
```

## Time Complexity Analysis

### For algorithm 1

#### Operations -

1. **isCalled** is set 0 for all indices =  $O(N)$ .
2. For each index =  $O(1*N) = O(N)$ 
  - a. **isCalled** for the next (six) neighbours is checked =  $O(1*6) = O(1)$ .

The algorithm is stopped once 100 is reached, however for the worst case, when there are no ladders, all the N indices have to be visited.

**Overall Time Complexity =  $O(N)$**

### For algorithm 2

#### Operations -

1. **minStepsTo** and **prevSquareFor** is set for all indices =  $O(N)$ .
2. For each index =
  - a. The previous 6 indices are used to set a value =  $O(1*6) = O(1)$
  - b. If a snake is encountered, update value at snake tail and re-run through the loop for the snake length =  $O(1*snakeLength)$ .

**Overall Time Complexity =  $O(N + snakeLengths)$**

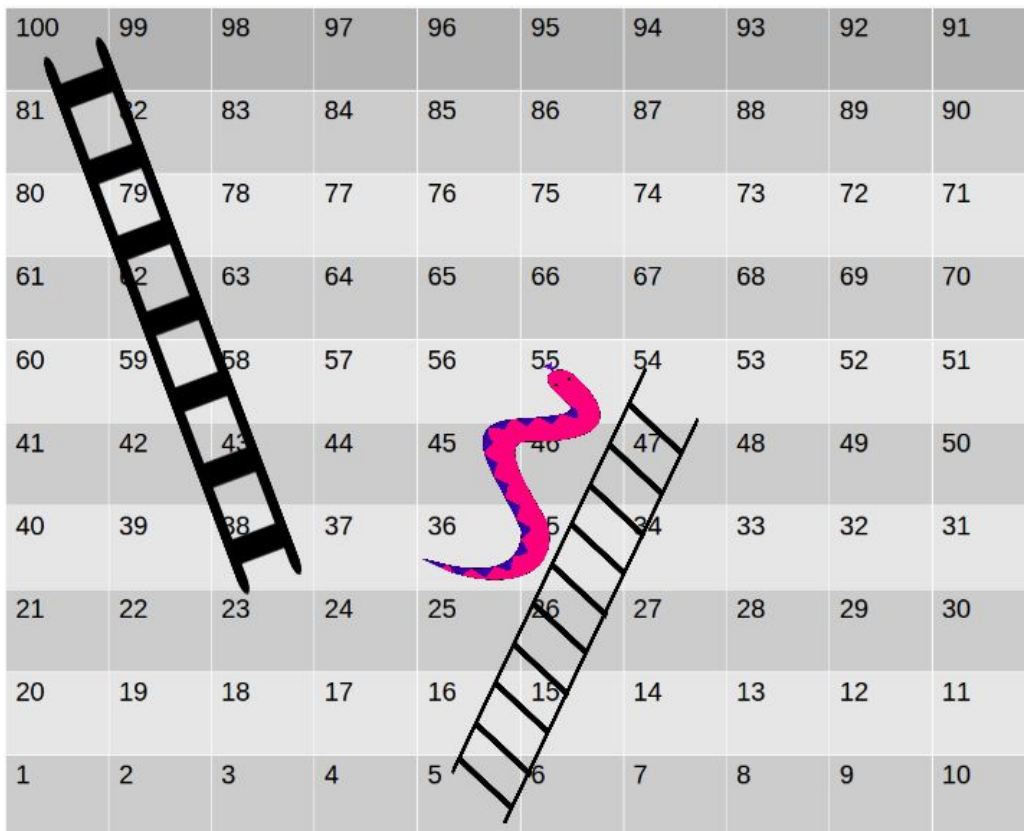
Both the algorithms are linear in number of squares. The exact comparison depends on the time required for the individual operations and on the nature of the board.

## Challenges Faced

### Boundary Cases

#### Step back to move Ahead

On first glance, one of the best approaches we came up with was to break the board recursively, each time taking a ladder and finding minimum steps taken from start to ladder bottom and ladder top to end. In this approach we decided to avoid all snakes and the number of throws required to reach from a square A to B would be  $\text{floor}((B-A)/6)$ . (Considering no snake-heads at multiples of 6.) This approach solved the problem in time  $O(\text{ladder}^2)$



100	99	98	97	96	95	94	93	92	91
81	82	83	84	85	86	87	88	89	90
80	79	78	77	76	75	74	73	72	71
61	62	63	64	65	66	67	68	69	70
60	59	58	57	56	55	54	53	52	51
41	42	43	44	45	46	47	48	49	50
40	39	38	37	36	35	34	33	32	31
21	22	23	24	25	26	27	28	29	30
20	19	18	17	16	15	14	13	12	11
1	2	3	4	5	6	7	8	9	10



In the above, according to the said approach the answer with ignoring all snakes would come out to be:

**0 -> 6 -> 12 -> 18 -> 24 -> 30 -> 36 -> 100 (7 ROLLS)**

But if we take the first ladder, snake and then the other snake:

**0 -> 54 -> 36 -> 100 (3 ROLLS)**

It is clear that taking the snake can offer a better solution in some cases. *So is with life, to take a huge leap forward, at times we have to take a small step back.*

### **Continuous 6 six snakes in a line**

Firstly we didn't check for the case when there is no answer. When 6 snakes mouths appear continuously on the consecutive squares of the board for e.g. 94,95,96,97,98,99 and there is no ladder directly to 100. We hadn't accounted for this border case, so we had to adjust our code such that it gives an output of "No path found".

--

Made by-

Arpan Banerjee	- 150070011
Arvind Singh Arya	- 150070042
Nihal Singh	- 150040015
Srivatsan Sridhar	- 150070005