Snakes and Ladders using BFS

Rohit Raval

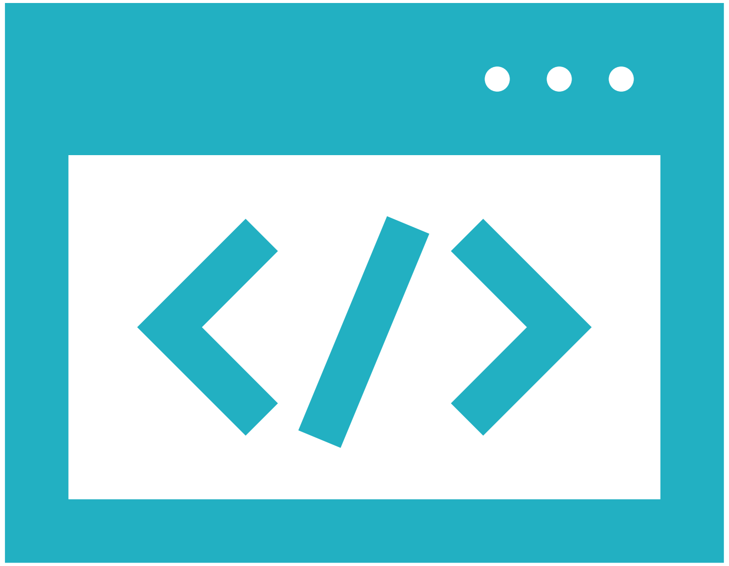
Aniruddh Kulkarni

Khushal Sharma

BFS

- In BFS, our input is graph G and start vertex is S
- Pseudocode of BFS is as follows

```
mark s as visited
add S to the queue

while queue not empty:
    currentVertex = queue.Dequeue

    for each adjacentVertex of currentVertex:
        if adjacentVertex is unvisited:
            add adjacentVertex to queue
            mark adjacentVertex as visited
```



Interpretation

Add	Remove	Add	Repeat
Add start vertex S to the queue	Remove an element from queue	Add all its unvisited adjacent vertices to queue	Repeat from step 2 until queue is empty

Diving Deep in the pseudocode

mark s as visited

add S to the queue

while queue not empty:

currentVertex = queue.Dequeue

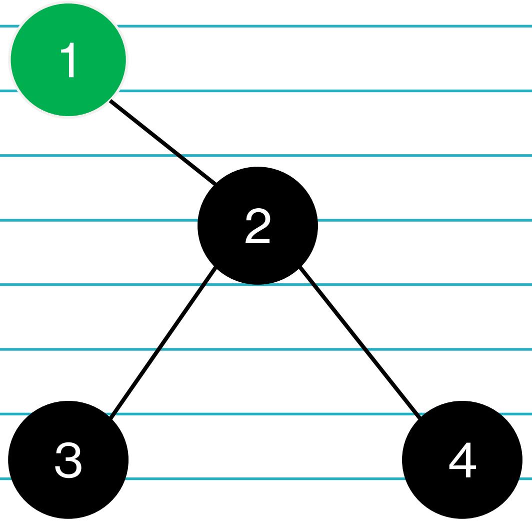
for each adjacentVertex of currentVertex:

if adjacentVertex is unvisited:

add adjacentVertex to queue

mark adjacentVertex as visited

queue = {1}



Initial State

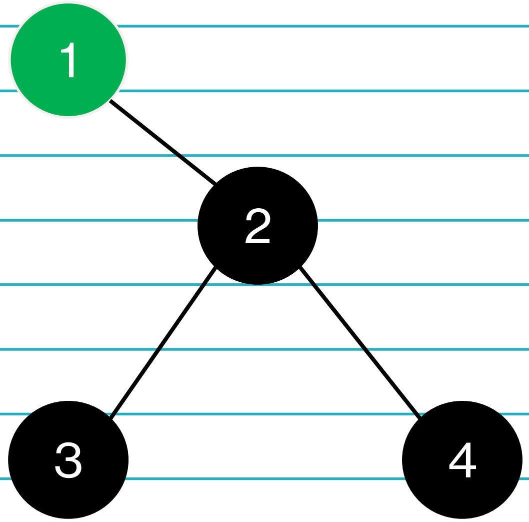
Say start vertex as S = 1

Mark vertex 1 as visited

Add vertex 1 to the queue

Diving Deep in the pseudocode

mark s as visited
add S to the queue
while queue not empty:
currentVertex = queue.Dequeue
for each adjacentVertex of currentVertex:
if adjacentVertex is unvisited:
add adjacentVertex to queue
mark adjacentVertex as visited

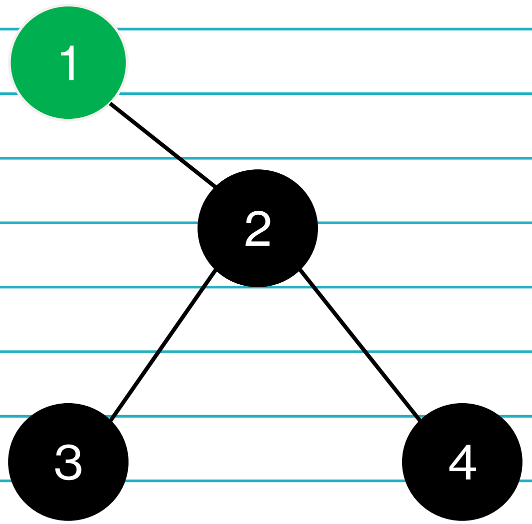


queue = { }

currentVertex = 1

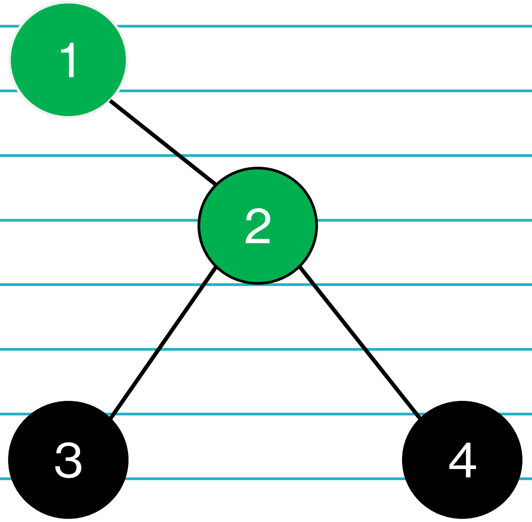
Outer loop iteration 1

Diving Deep in the pseudocode



mark s as visited		
add S to the queue		
while queue not empty:		
currentVertex = queue.Dequeue		
for each adjacentVertex of currentVertex: ←		
if adjacentVertex is unvisited:		
add adjacentVertex to queue		
mark adjacentVertex as visited		
	queue = { }	Outer loop iteration 1
	currentVertex = 1	Only Vertex 2 is adjacent to vertex 1
	adjacentVertex = 2	

Diving Deep in the pseudocode



mark s as visited

add S to the queue

while queue not empty:

currentVertex = queue.Dequeue

for each adjacentVertex of currentVertex:

if adjacentVertex is unvisited:

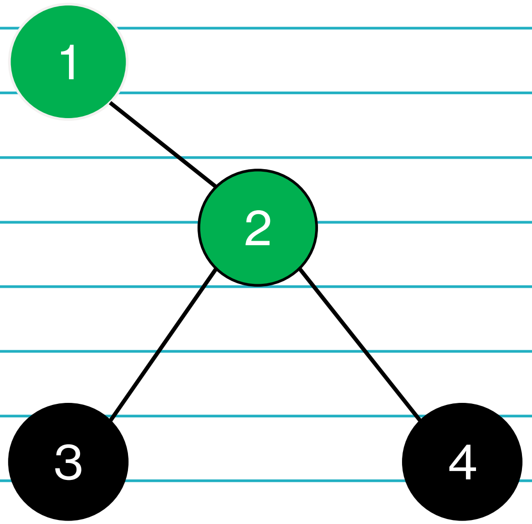
add adjacentVertex to queue

mark adjacentVertex as visited

queue = {2}	Outer loop iteration 1
currentVertex = 1	Inner loop iteration 1
adjacentVertex = 2	Vertex 2 is unvisted
	Mark Vertex 2 as visited and add it to the queue

Diving Deep in the pseudocode

```
mark s as visited
add S to the queue
while queue not empty:
currentVertex = queue.Dequeue
for each adjacentVertex of currentVertex:
if adjacentVertex is unvisited:
add adjacentVertex to queue
mark adjacentVertex as visited
```



queue = { }

Outer loop iteration 2

currentVertex = 2

Diving Deep in the pseudocode

mark s as visited

add S to the queue

while queue not empty:

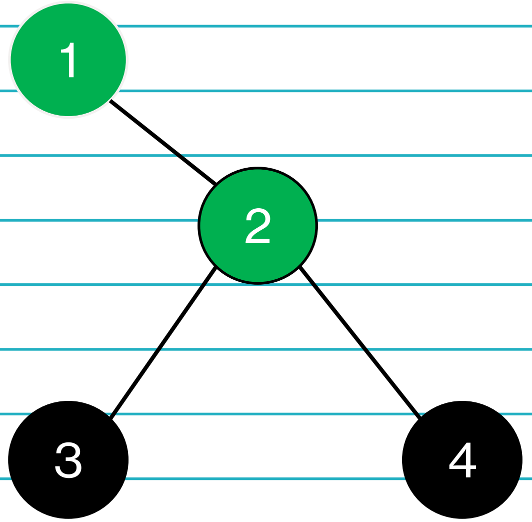
currentVertex = queue.Dequeue

for each adjacentVertex of currentVertex:

if adjacentVertex is unvisited:

add adjacentVertex to queue

mark adjacentVertex as visited



queue = { }

currentVertex = 2

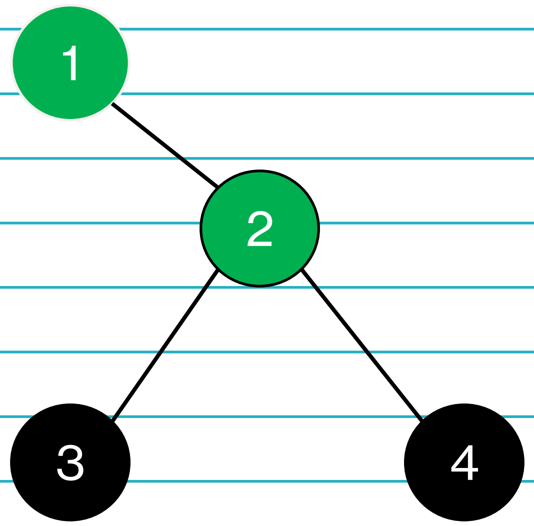
Outer loop iteration 2

Verices 1, 3, 4 are adjacent to Vertex 2

Let's go by that order

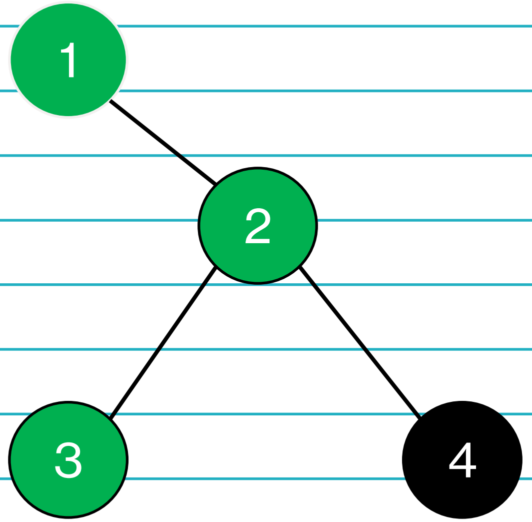
Diving Deep in the pseudocode

mark s as visited		
add S to the queue		
while queue not empty:		
currentVertex = queue.Dequeue		
for each adjacentVertex of currentVertex:		
if adjacentVertex is unvisited:		
add adjacentVertex to queue		
mark adjacentVertex as visited		
	queue = { }	Outer loop iteration
	currentVertex = 2	Inner loop iteration
	adjacentVertex = 1	Vertex 1 is already marked as visited. So, we don’t do anything



Diving Deep in the pseudocode

mark s as visited
add S to the queue
while queue not empty:
currentVertex = queue.Dequeue
for each adjacentVertex of currentVertex:
if adjacentVertex is unvisited:
add adjacentVertex to queue
mark adjacentVertex as visited

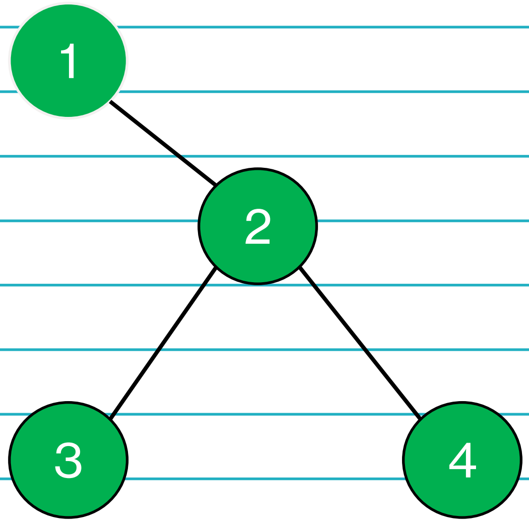


queue = { 3 }
currentVertex = 2
adjacentVertex = 3

Outer loop iteration 2
Inner loop iteration 2
Vertex 3 not visted
Add it to the queue and mark it as visited

Diving Deep in the pseudocode

mark s as visited
add S to the queue
while queue not empty:
currentVertex = queue.Dequeue
for each adjacentVertex of currentVertex:
if adjacentVertex is unvisited:
add adjacentVertex to queue
mark adjacentVertex as visited

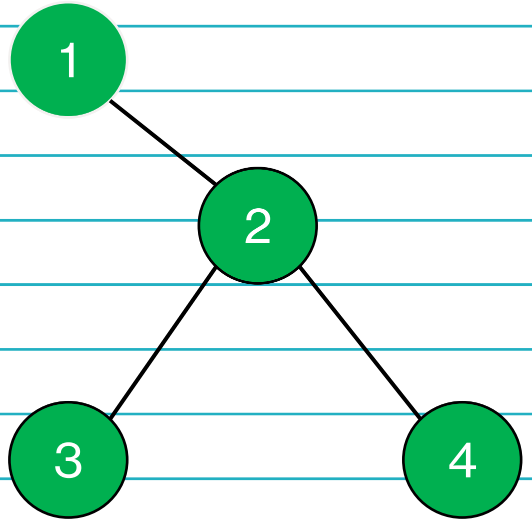


queue = { 3, 4 }
currentVertex = 2
adjacentVertex = 4

Outer for loop iteration 2
Inner loop iteration 3
Vertex 4 Is not visited
Add it to queue and mark it as visited

Diving Deep in the pseudocode

```
mark s as visited
add S to the queue
while queue not empty:
currentVertex = queue.Dequeue
for each adjacentVertex of currentVertex:  ←
if adjacentVertex is unvisited:
add adjacentVertex to queue
mark adjacentVertex as visited
```



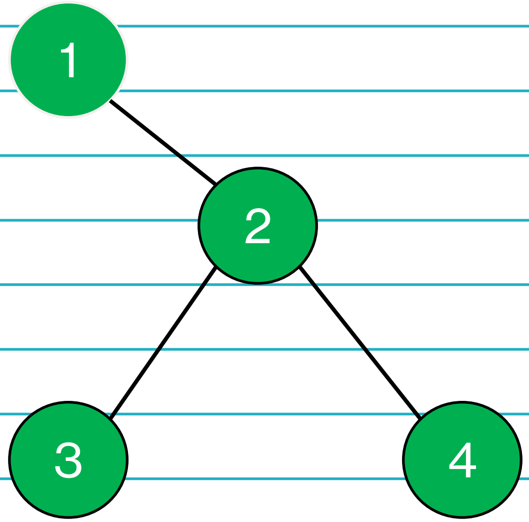
queue = { 4 }

Outer loop iteration 3

currentVertex = 3

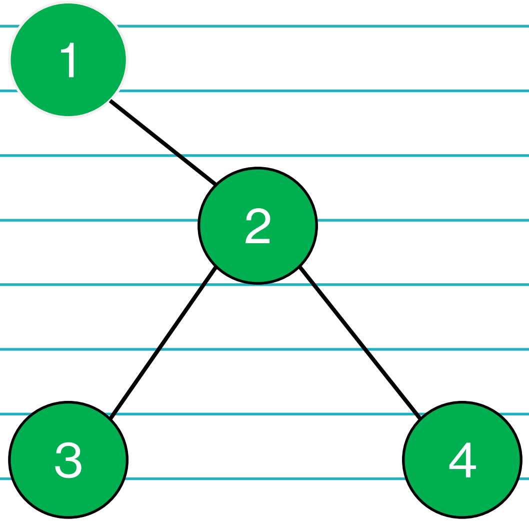
Diving Deep in the pseudocode

mark s as visited
add S to the queue
while queue not empty:
currentVertex = queue.Dequeue
for each adjacentVertex of currentVertex:
if adjacentVertex is unvisited: ←
add adjacentVertex to queue
mark adjacentVertex as visited



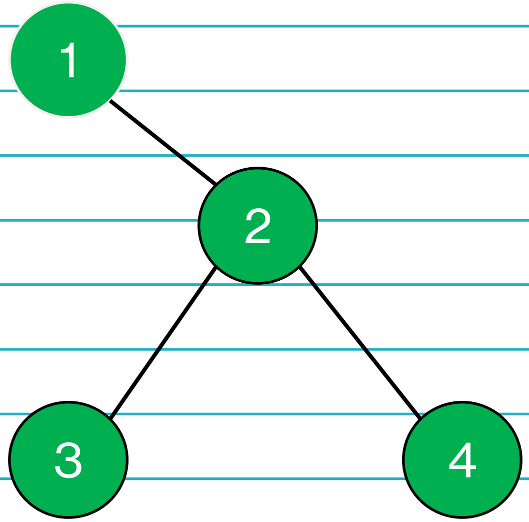
queue = { 4 }	Outer loop iteration 3
currentVertex = 3	Inner loop iteration 1
adjacentVertex = 2	Vertex 2 is adjacent to Vertex 3

Diving Deep in the pseudocode



mark s as visited		
add S to the queue		
while queue not empty:		
currentVertex = queue.Dequeue		
for each adjacentVertex of currentVertex:		
if adjacentVertex is unvisited: ←		
add adjacentVertex to queue		
mark adjacentVertex as visited		
	queue = { 4 }	Outer loop iteration 3
	currentVertex = 3	Inner loop iteration 1
	adjacentVertex = 2	Vertex 2 is already marked as visited. So, we don't do anything

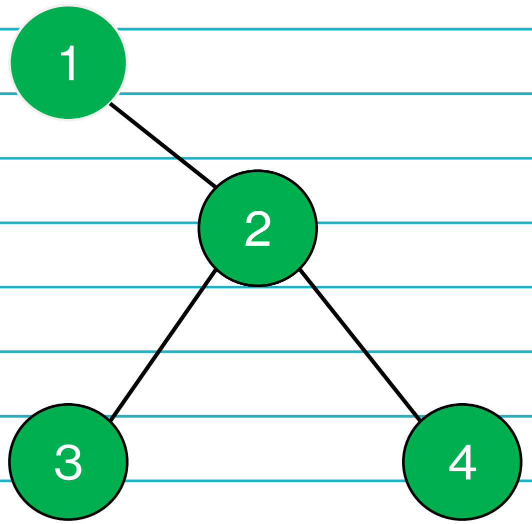
Diving Deep in the pseudocode



mark s as visited		
add S to the queue		
while queue not empty:		
currentVertex = queue.Dequeue		
for each adjacentVertex of currentVertex:		
if adjacentVertex is unvisited:		
add adjacentVertex to queue		
mark adjacentVertex as visited		
	queue = { 4 }	Outer loop iteration 4
	currentVertex = 3	Inner loop iteration 2
	adjacentVertex = 4	Vertex 4 is already marked as visited. So, we don't do anything

Diving Deep in the pseudocode

mark s as visited
add S to the queue
while queue not empty:
currentVertex = queue.Dequeue
for each adjacentVertex of currentVertex:
if adjacentVertex is unvisited:
add adjacentVertex to queue
mark adjacentVertex as visited



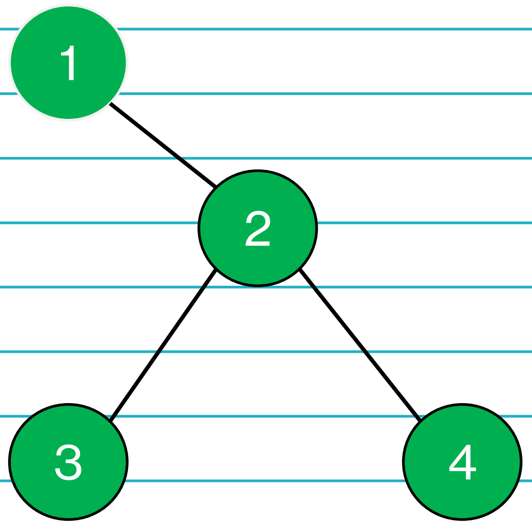
queue = { }

Outer loop iteration 4

currentVertex = 4

Diving Deep in the pseudocode

mark s as visited
add S to the queue
while queue not empty:
currentVertex = queue.Dequeue
for each adjacentVertex of currentVertex:
if adjacentVertex is unvisited:
add adjacentVertex to queue
mark adjacentVertex as visited



queue = { }
currentVertex = 4

Outer loop iteration 4
Vertex 2 and 3 are adjacent to Vertex 4

Diving Deep in the pseudocode

mark s as visited

add S to the queue

while queue not empty:

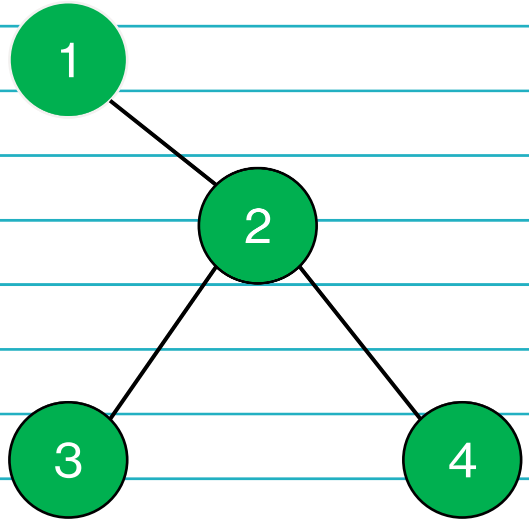
currentVertex = queue.Dequeue

for each adjacentVertex of currentVertex:

if adjacentVertex is unvisited:

add adjacentVertex to queue

mark adjacentVertex as visited



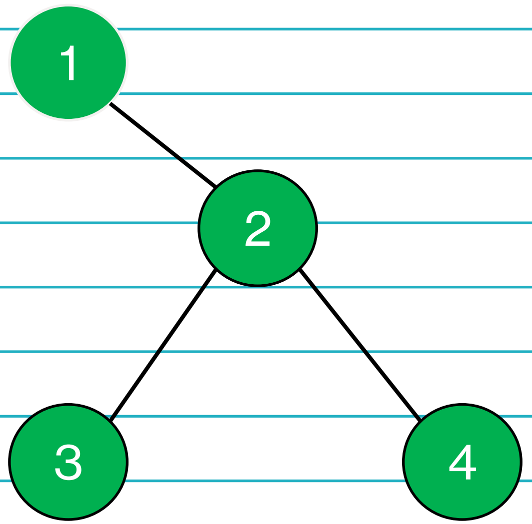
queue = { }
Outer loop iteration 4

currentVertex = 4
Inner loop iteration 1

Vertex 2, is already marked as visited. So we don't do anything

Diving Deep in the pseudocode

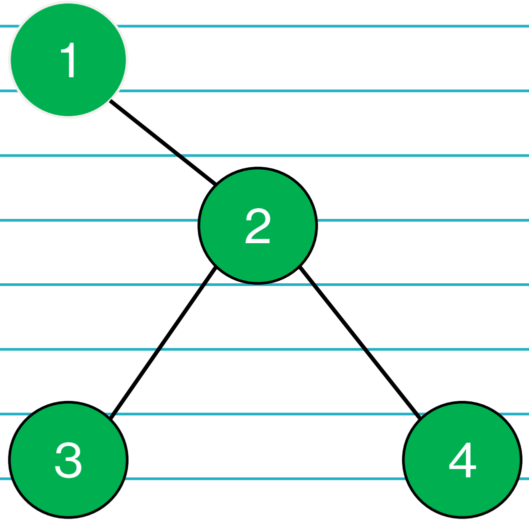
mark s as visited
add S to the queue
while queue not empty:
currentVertex = queue.Dequeue
for each adjacentVertex of currentVertex:
if adjacentVertex is unvisited:
add adjacentVertex to queue
mark adjacentVertex as visited



queue = { }	Outer loop iteration 4
currentVertex = 4	Inner loop iteration 1
adjacentVertex = 3	Vertex 3, is already marked as visited. So we don't do anything

Diving Deep in the pseudocode

```
mark s as visited
add S to the queue
while queue not empty:
currentVertex = queue.Dequeue
for each adjacentVertex of currentVertex:
if adjacentVertex is unvisited:
add adjacentVertex to queue
mark adjacentVertex as visited
```



queue = { }

Our queue is empty

So, we don't have an iteration 5

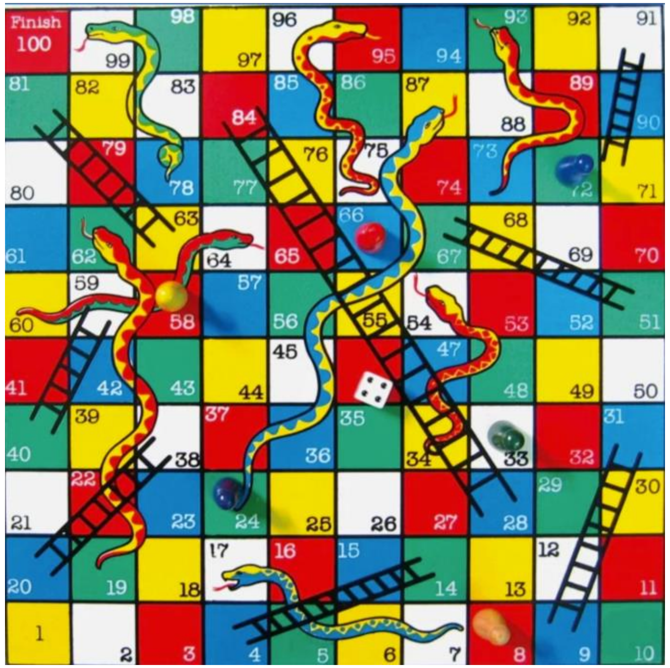
We're done

Performance

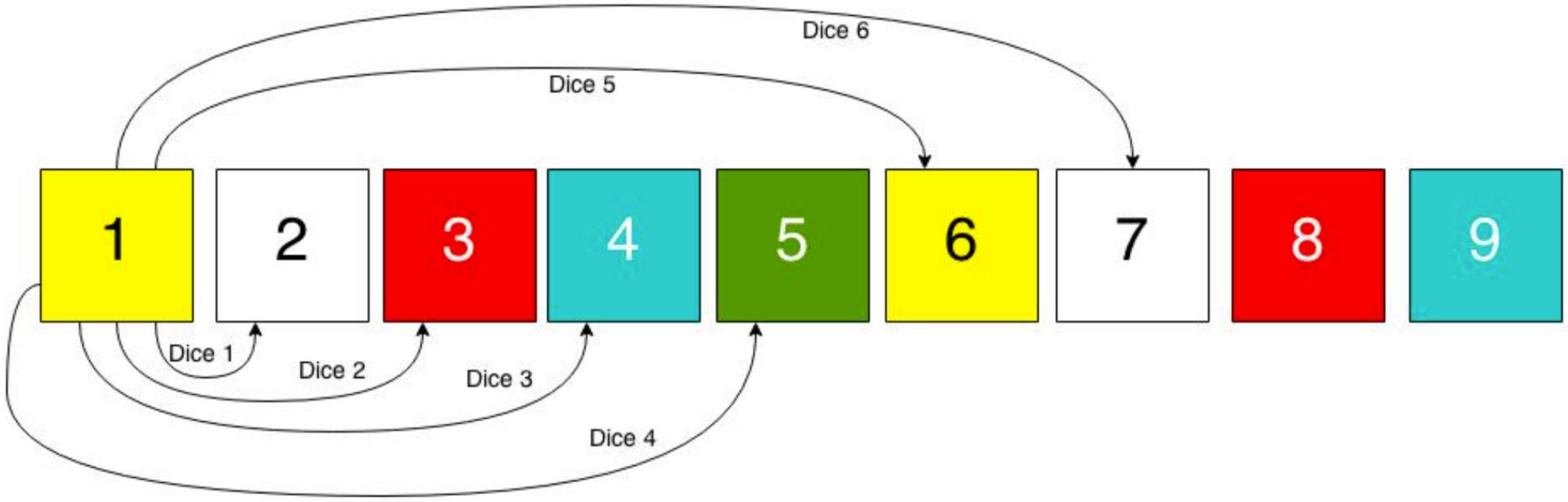
Time complexity of Breadth First Search is O(V+E)

The Problem

- For a given configuration of a Snakes and Ladders board, find the shortest path to finish the game
- Steps to solve this
- Represent the board as a graph
- Run BFS on the graph



Dice roll for Block 1



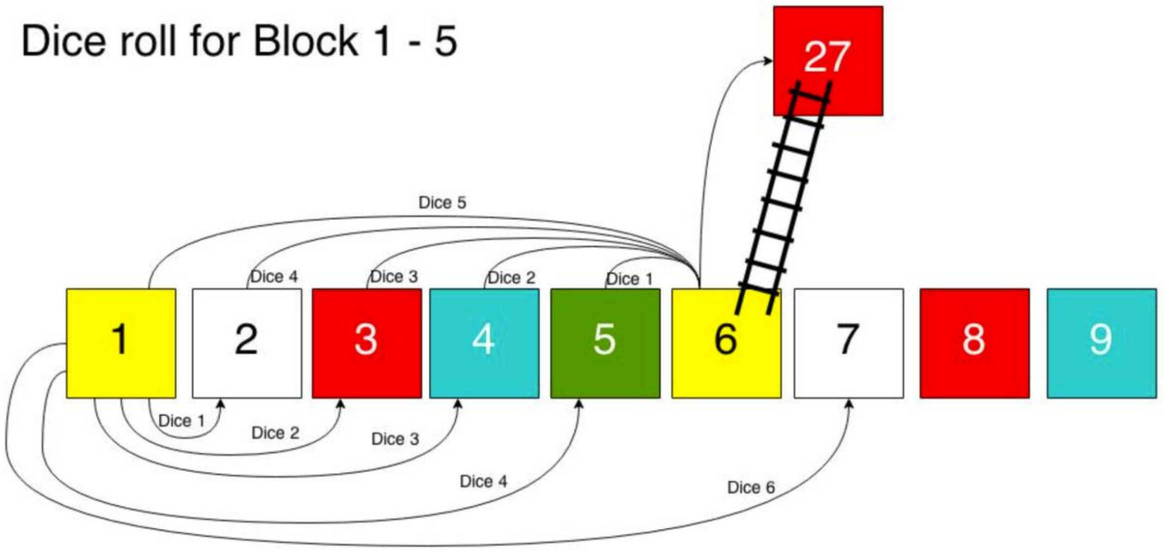
Graph Representation

- Each block from 1 to 100 is a vertex
- Each possible move from a dice roll, is an edge
- Start vertex is 1. End vertex is 100

Dice roll for Block 1-5

If you roll 5 from block 1 you will jump directly to block 27. So is for block 2 when you roll out 4 or block 3 when you roll out 3 and so on. Now, “logically” speaking, the block 6 does not exists in our graph…!

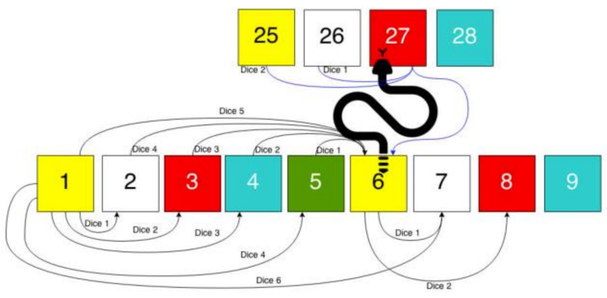
Think about the statement for a while. Whenever you reach block, you are directly jumping to block 27, you don’t stay there.



Dice Roll for 25

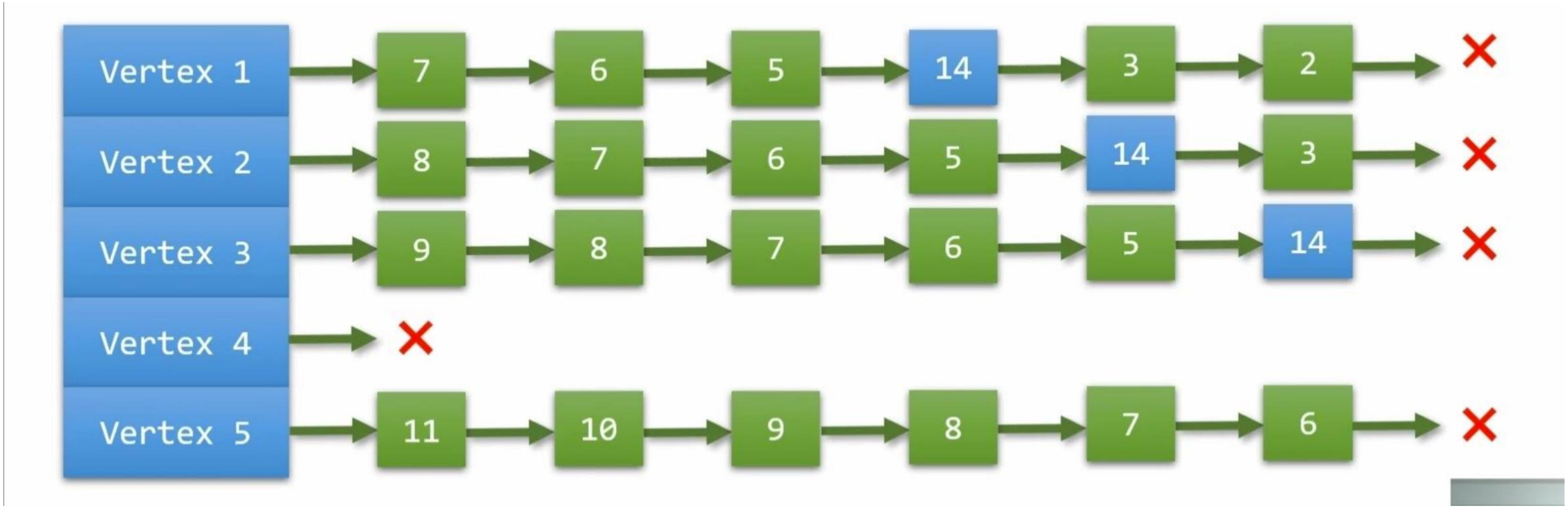
We assume that getting caught by a snake is always unfavorable and will not add to the progress of a short path. Just to get a better idea of the scenario of a ladder and snake, I have depicted what the Adjacency List would look like for the above two examples shown in the pictures.

Dice roll for Block 25

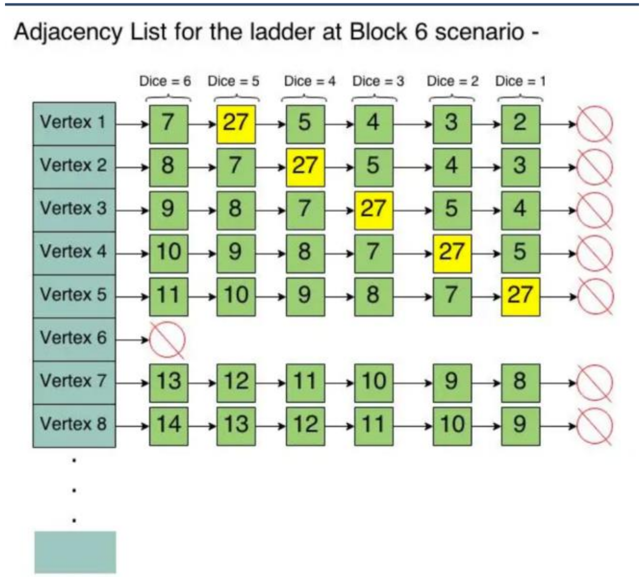


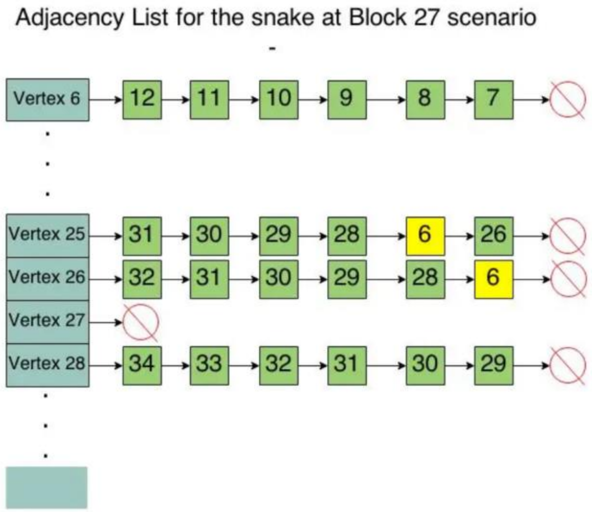
Linked List

- Since there’s a ladder at vertex 4, as there’s a ladder which takes the player from 4 to 14
- To generalize the vertices having the ladder on the Gameboard, the linked list will be empty



Adjacency List for the ladder at Block 6 scenario





Problem Statement

Here, we are trying to obtain the minimum number of dice rolls required to finish a game of Snakes and Ladders. The program takes inputs from the user to build a user defined board for the BFS algorithm to run on. The BFS algorithm is combined with the implementation of adjacency list for ease of execution of the algorithm.

Synopsis of our approach

We are using queues and adjacency lists for executing our snakes and ladders code. Queues are used for storing the next vertex that is supposed to be explored. Adjacency lists are used to store the neighbouring vertices that can be hopped onto after reached the current vertex. The hops are achieved at the dice roll. We are also storing the parent vertex of the current vertex.

Our Implementation: replaceEdge function

This function is used to replace the 6 edges that contain the vertex which is ladder or snake. The adjacent elements of that vertex are erased and it is redirected to the top of the ladder or the bottom of the snake as the case may be.

```
void replaceEdgeFor6PreceedingVertices(vector< list<int> >& adjacencyList, int startVertex, int oldEdge, int newEdge)
{
    // For the 6 vertices preceeding 'startVertex' do the edge replacement
    for (int i = startVertex - 1; i >= startVertex - 6 && i > 0; --i) {
        std::replace(adjacencyList[i].begin(), adjacencyList[i].end(), oldEdge, newEdge);
    }
}
```

Our Implementation: printPath function

This function is used to print the path from source to destination i.e. From position 1 to 100 in a custom print style

```
void printPathFromSourceToDestination(int parent[], int destination)
{
    if (parent[destination] == -1) {
        // We have reached the source vertex
        cout<<destination<<" -> ";
    }
    else {
        printPathFromSourceToDestination(parent, parent[destination]);
        cout<<destination<<" -> ";
    }
}
```

Our Implementation: BFS function

This is the main BFS algorithm where it searches for the minimum number of dice rolls required according to the given board to go from start till end.

```
void breadthFirstSearch(vector< list<int> > adjacencyList, int parent[], int level[], int start)
{
    list<int>::iterator itr;

    // Level of start vertex will be 0, the level of all its adjcent
    // vertices will be 1, their adjacent vertices will be 2, and so on
    level[start] = 0;

    list<int> queue; // Queue of vertices to be processed

    queue.push_back(start); // Add start vertex to the queue

    while (!queue.empty()) // While there are vertices to be processed
    {
        // Get the first vertex in the queue.
        // Note - .front() does not remove the front element.
        int newVertex = queue.front();

        // Iterator to explore all the vertices adjacent to it
        itr = adjacencyList[newVertex].begin();

        while (itr != adjacencyList[newVertex].end()) {
            if (level[*itr] == -1) {                    // Check if it is an unvisited vertex
                level[*itr] = level[newVertex] + 1; // Set level of adjacent vertex
                parent[*itr] = newVertex;           // Set parent of adjacent vertex
                queue.push_back(*itr);              // Add the adjacent vertex to queue
            }

            ++itr;
        }

        queue.pop_front(); // Pop out the processed vertex
    }
}
```

Execution: Inputs

- We gave a custom board for execution

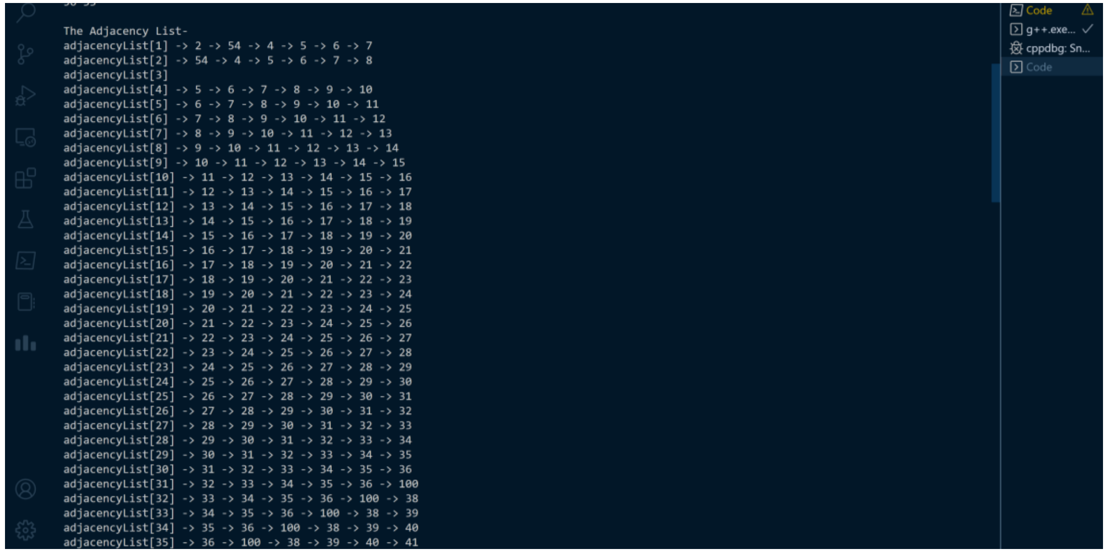
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
Windows PowerShell

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\aniku> cd "c:\Users\aniku\Downloads\" ; if ($?) { g++ SnakesandLadders.cpp -o SnakesandLadders } ; if ($?) { .\SnakesandLadders
 }
c:/mingw/bin/../lib/gcc/x86_64-w64-mingw32/9.2.0/../../../../x86_64-w64-mingw32/bin/ld.exe: cannot open output file SnakesandLadders.exe: P
ermission denied
collect2.exe: error: ld returned 1 exit status
PS C:\Users\aniku\Downloads> cd "c:\Users\aniku\Downloads\" ; if ($?) { g++ SnakesandLadders.cpp -o SnakesandLadders } ; if ($?) { .\Snakes
andLadders }
Enter number of ladders -
2
Enter the ladder which goes from V1 -> V2
3 54
37 100
Enter number of snakes -
1
Enter the snake which goes from V1 -> V2
56 33
```

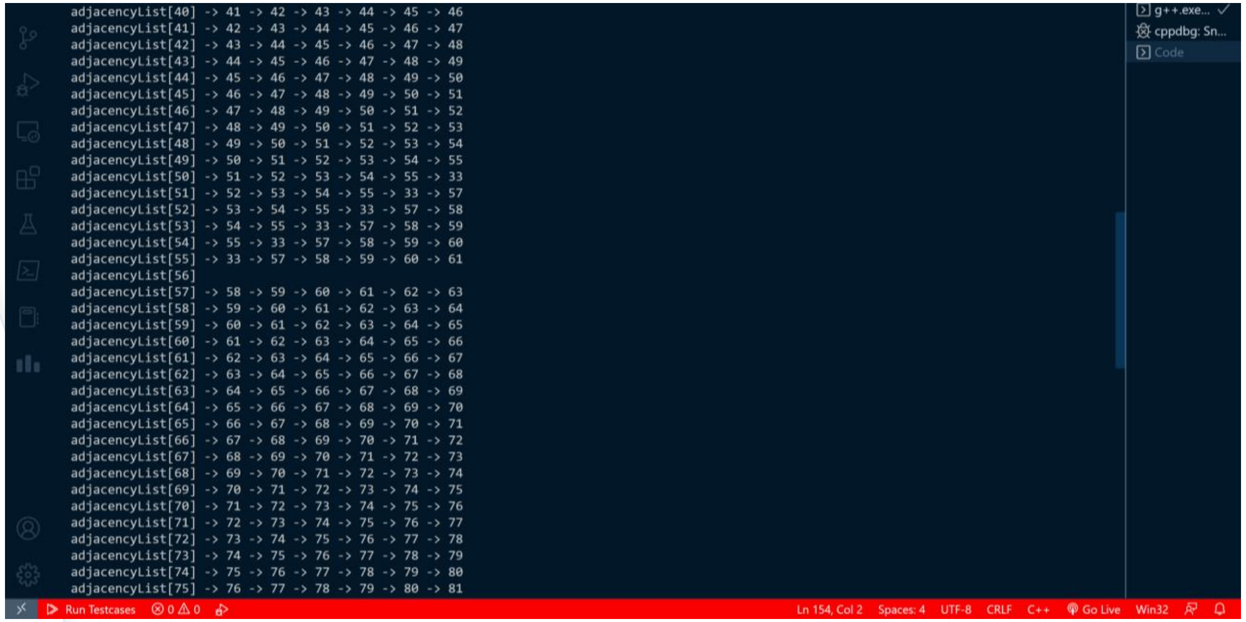
Code
Code
g++.exe...
cppdbg: Sn...
Code

Execution: Output

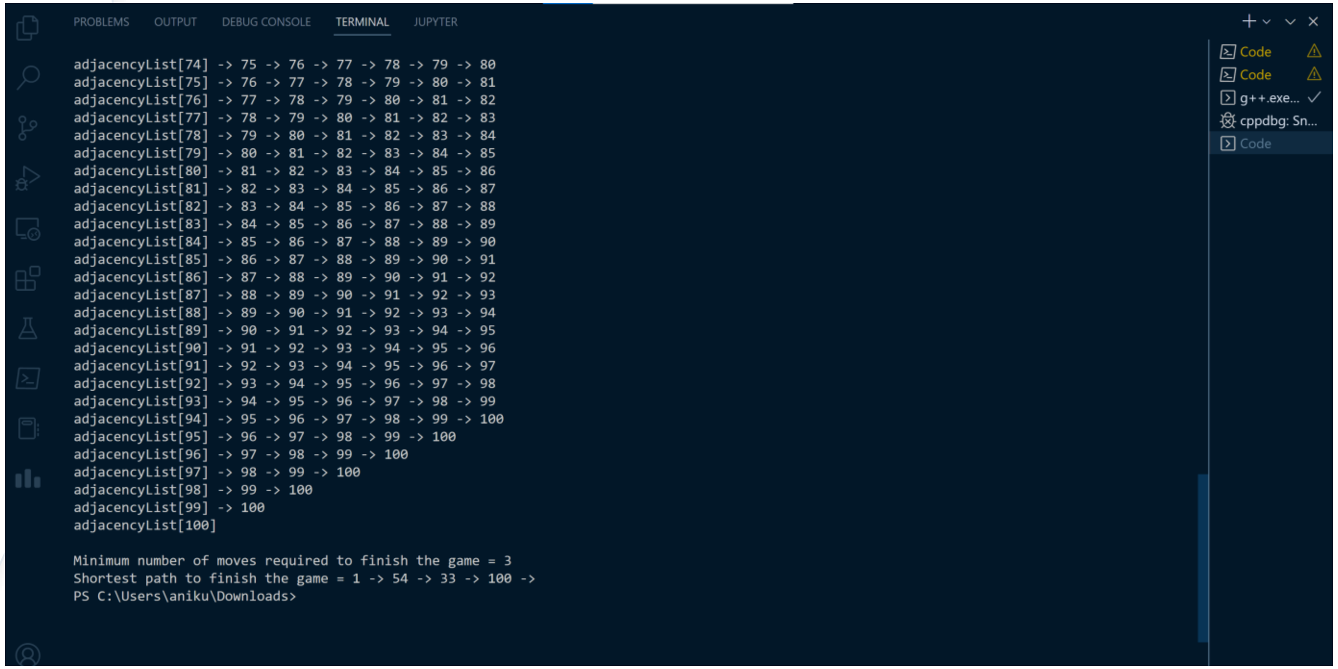
- Here is the output of the given snakes and ladders board



```
The Adjacency List-
adjacencyList[1] -> 2 -> 54 -> 4 -> 5 -> 6 -> 7
adjacencyList[2] -> 54 -> 4 -> 5 -> 6 -> 7 -> 8
adjacencyList[3]
adjacencyList[4] -> 5 -> 6 -> 7 -> 8 -> 9 -> 10
adjacencyList[5] -> 6 -> 7 -> 8 -> 9 -> 10 -> 11
adjacencyList[6] -> 7 -> 8 -> 9 -> 10 -> 11 -> 12
adjacencyList[7] -> 8 -> 9 -> 10 -> 11 -> 12 -> 13
adjacencyList[8] -> 9 -> 10 -> 11 -> 12 -> 13 -> 14
adjacencyList[9] -> 10 -> 11 -> 12 -> 13 -> 14 -> 15
adjacencyList[10] -> 11 -> 12 -> 13 -> 14 -> 15 -> 16
adjacencyList[11] -> 12 -> 13 -> 14 -> 15 -> 16 -> 17
adjacencyList[12] -> 13 -> 14 -> 15 -> 16 -> 17 -> 18
adjacencyList[13] -> 14 -> 15 -> 16 -> 17 -> 18 -> 19
adjacencyList[14] -> 15 -> 16 -> 17 -> 18 -> 19 -> 20
adjacencyList[15] -> 16 -> 17 -> 18 -> 19 -> 20 -> 21
adjacencyList[16] -> 17 -> 18 -> 19 -> 20 -> 21 -> 22
adjacencyList[17] -> 18 -> 19 -> 20 -> 21 -> 22 -> 23
adjacencyList[18] -> 19 -> 20 -> 21 -> 22 -> 23 -> 24
adjacencyList[19] -> 20 -> 21 -> 22 -> 23 -> 24 -> 25
adjacencyList[20] -> 21 -> 22 -> 23 -> 24 -> 25 -> 26
adjacencyList[21] -> 22 -> 23 -> 24 -> 25 -> 26 -> 27
adjacencyList[22] -> 23 -> 24 -> 25 -> 26 -> 27 -> 28
adjacencyList[23] -> 24 -> 25 -> 26 -> 27 -> 28 -> 29
adjacencyList[24] -> 25 -> 26 -> 27 -> 28 -> 29 -> 30
adjacencyList[25] -> 26 -> 27 -> 28 -> 29 -> 30 -> 31
adjacencyList[26] -> 27 -> 28 -> 29 -> 30 -> 31 -> 32
adjacencyList[27] -> 28 -> 29 -> 30 -> 31 -> 32 -> 33
adjacencyList[28] -> 29 -> 30 -> 31 -> 32 -> 33 -> 34
adjacencyList[29] -> 30 -> 31 -> 32 -> 33 -> 34 -> 35
adjacencyList[30] -> 31 -> 32 -> 33 -> 34 -> 35 -> 36
adjacencyList[31] -> 32 -> 33 -> 34 -> 35 -> 36 -> 100
adjacencyList[32] -> 33 -> 34 -> 35 -> 36 -> 100 -> 38
adjacencyList[33] -> 34 -> 35 -> 36 -> 100 -> 38 -> 39
adjacencyList[34] -> 35 -> 36 -> 100 -> 38 -> 39 -> 40
adjacencyList[35] -> 36 -> 100 -> 38 -> 39 -> 40 -> 41
```



```
adjacencyList[40] -> 41 -> 42 -> 43 -> 44 -> 45 -> 46
adjacencyList[41] -> 42 -> 43 -> 44 -> 45 -> 46 -> 47
adjacencyList[42] -> 43 -> 44 -> 45 -> 46 -> 47 -> 48
adjacencyList[43] -> 44 -> 45 -> 46 -> 47 -> 48 -> 49
adjacencyList[44] -> 45 -> 46 -> 47 -> 48 -> 49 -> 50
adjacencyList[45] -> 46 -> 47 -> 48 -> 49 -> 50 -> 51
adjacencyList[46] -> 47 -> 48 -> 49 -> 50 -> 51 -> 52
adjacencyList[47] -> 48 -> 49 -> 50 -> 51 -> 52 -> 53
adjacencyList[48] -> 49 -> 50 -> 51 -> 52 -> 53 -> 54
adjacencyList[49] -> 50 -> 51 -> 52 -> 53 -> 54 -> 55
adjacencyList[50] -> 51 -> 52 -> 53 -> 54 -> 55 -> 33
adjacencyList[51] -> 52 -> 53 -> 54 -> 55 -> 33 -> 57
adjacencyList[52] -> 53 -> 54 -> 55 -> 33 -> 57 -> 58
adjacencyList[53] -> 54 -> 55 -> 33 -> 57 -> 58 -> 59
adjacencyList[54] -> 55 -> 33 -> 57 -> 58 -> 59 -> 60
adjacencyList[55] -> 33 -> 57 -> 58 -> 59 -> 60 -> 61
adjacencyList[56]
adjacencyList[57] -> 58 -> 59 -> 60 -> 61 -> 62 -> 63
adjacencyList[58] -> 59 -> 60 -> 61 -> 62 -> 63 -> 64
adjacencyList[59] -> 60 -> 61 -> 62 -> 63 -> 64 -> 65
adjacencyList[60] -> 61 -> 62 -> 63 -> 64 -> 65 -> 66
adjacencyList[61] -> 62 -> 63 -> 64 -> 65 -> 66 -> 67
adjacencyList[62] -> 63 -> 64 -> 65 -> 66 -> 67 -> 68
adjacencyList[63] -> 64 -> 65 -> 66 -> 67 -> 68 -> 69
adjacencyList[64] -> 65 -> 66 -> 67 -> 68 -> 69 -> 70
adjacencyList[65] -> 66 -> 67 -> 68 -> 69 -> 70 -> 71
adjacencyList[66] -> 67 -> 68 -> 69 -> 70 -> 71 -> 72
adjacencyList[67] -> 68 -> 69 -> 70 -> 71 -> 72 -> 73
adjacencyList[68] -> 69 -> 70 -> 71 -> 72 -> 73 -> 74
adjacencyList[69] -> 70 -> 71 -> 72 -> 73 -> 74 -> 75
adjacencyList[70] -> 71 -> 72 -> 73 -> 74 -> 75 -> 76
adjacencyList[71] -> 72 -> 73 -> 74 -> 75 -> 76 -> 77
adjacencyList[72] -> 73 -> 74 -> 75 -> 76 -> 77 -> 78
adjacencyList[73] -> 74 -> 75 -> 76 -> 77 -> 78 -> 79
adjacencyList[74] -> 75 -> 76 -> 77 -> 78 -> 79 -> 80
adjacencyList[75] -> 76 -> 77 -> 78 -> 79 -> 80 -> 81
```



```
adjacencyList[74] -> 75 -> 76 -> 77 -> 78 -> 79 -> 80
adjacencyList[75] -> 76 -> 77 -> 78 -> 79 -> 80 -> 81
adjacencyList[76] -> 77 -> 78 -> 79 -> 80 -> 81 -> 82
adjacencyList[77] -> 78 -> 79 -> 80 -> 81 -> 82 -> 83
adjacencyList[78] -> 79 -> 80 -> 81 -> 82 -> 83 -> 84
adjacencyList[79] -> 80 -> 81 -> 82 -> 83 -> 84 -> 85
adjacencyList[80] -> 81 -> 82 -> 83 -> 84 -> 85 -> 86
adjacencyList[81] -> 82 -> 83 -> 84 -> 85 -> 86 -> 87
adjacencyList[82] -> 83 -> 84 -> 85 -> 86 -> 87 -> 88
adjacencyList[83] -> 84 -> 85 -> 86 -> 87 -> 88 -> 89
adjacencyList[84] -> 85 -> 86 -> 87 -> 88 -> 89 -> 90
adjacencyList[85] -> 86 -> 87 -> 88 -> 89 -> 90 -> 91
adjacencyList[86] -> 87 -> 88 -> 89 -> 90 -> 91 -> 92
adjacencyList[87] -> 88 -> 89 -> 90 -> 91 -> 92 -> 93
adjacencyList[88] -> 89 -> 90 -> 91 -> 92 -> 93 -> 94
adjacencyList[89] -> 90 -> 91 -> 92 -> 93 -> 94 -> 95
adjacencyList[90] -> 91 -> 92 -> 93 -> 94 -> 95 -> 96
adjacencyList[91] -> 92 -> 93 -> 94 -> 95 -> 96 -> 97
adjacencyList[92] -> 93 -> 94 -> 95 -> 96 -> 97 -> 98
adjacencyList[93] -> 94 -> 95 -> 96 -> 97 -> 98 -> 99
adjacencyList[94] -> 95 -> 96 -> 97 -> 98 -> 99 -> 100
adjacencyList[95] -> 96 -> 97 -> 98 -> 99 -> 100
adjacencyList[96] -> 97 -> 98 -> 99 -> 100
adjacencyList[97] -> 98 -> 99 -> 100
adjacencyList[98] -> 99 -> 100
adjacencyList[99] -> 100
adjacencyList[100]

Minimum number of moves required to finish the game = 3
Shortest path to finish the game = 1 -> 54 -> 33 -> 100 ->
PS C:\Users\aniku\Downloads>
```