Snake and Ladder Problem

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell. Basically, the player has total control over outcome of dice throw and wants to find out minimum number of throws required to reach last cell.

If the player reaches a cell which is base of a ladder, the player has to climb up that ladder and if reaches a cell is mouth of the snake, has to go down to the tail of snake without a dice throw.

For example consider the board shown on right side (taken from here), the minimum number of dice throws required to reach cell 30 from cell 1 is 3. Following are steps.

a) First throw two on dice to reach cell number 3 and then ladder to reach 22
b) Then throw 6 to reach 28.
c) Finally through 2 to reach 30.

There can be other solutions as well like (2, 2, 6), (2, 4, 4), (2, 3, 5).. etc.

We strongly recommend to minimize the browser and try this yourself first.

The idea is to consider the given snake and ladder board as a directed graph with number of vertices equal to the number of cells in the board. The problem reduces to finding the shortest path in a graph. Every vertex of the graph has an edge to next six vertices if next 6 vertices do not have a snake or ladder. If any of the next six vertices has a snake or ladder, then the edge from current vertex goes to the top of the ladder or tail of the snake. Since all edges are of equal weight, we can efficiently find shortest path using Breadth First Search of the graph.

Following is C++ implementation of the above idea. The input is represented by two things, first is 'N' which is number of cells in the given board, second is an array 'move[0…N-1]' of size N. An entry move[i] is -1 if there is no snake and no ladder from i, otherwise move[i] contains index of destination cell for the snake or the ladder at i.

```
// C++ program to find minimum number of dice throws required to
// reach last cell from first cell of a given snake and ladder
// board
#include<iostream>
#include <queue>
using namespace std;

// An entry in queue used in BFS
struct queueEntry
{
    int v;     // Vertex number
    int dist;  // Distance of this vertex from source
};

// This function returns minimum number of dice throws required to
// Reach last cell from 0'th cell in a snake and ladder game.
// move[] is an array of size N where N is no. of cells on board
// If there is no snake or ladder from cell i, then move[i] is -1
// Otherwise move[i] contains cell to which snake or ladder at i
// takes to.
int getMinDiceThrows(int move[], int N)
{
    // The graph has N vertices. Mark all the vertices as
```

```
    // not visited
    bool *visited = new bool[N];
    for (int i = 0; i < N; i++)
        visited[i] = false;

    // Create a queue for BFS
    queue<queueEntry> q;

    // Mark the node 0 as visited and enqueue it.
    visited[0] = true;
    queueEntry s = {0, 0};  // distance of 0't vertex is also 0
    q.push(s);  // Enqueue 0'th vertex

    // Do a BFS starting from vertex at index 0
    queueEntry qe;  // A queue entry (qe)
    while (!q.empty())
    {
        qe = q.front();
        int v = qe.v; // vertex no. of queue entry

        // If front vertex is the destination vertex,
        // we are done
        if (v == N-1)
            break;

        // Otherwise dequeue the front vertex and enqueue
        // its adjacent vertices (or cell numbers reachable
        // through a dice throw)
        q.pop();
        for (int j=v+1; j<=(v+6) && j<N; ++j)
        {
            // If this cell is already visited, then ignore
            if (!visited[j])
            {
                // Otherwise calculate its distance and mark it
                // as visited
                queueEntry a;
                a.dist = (qe.dist + 1);
                visited[j] = true;

                // Check if there a snake or ladder at 'j'
                // then tail of snake or top of ladder
                // become the adjacent of 'i'
                if (move[j] != -1)
                    a.v = move[j];
                else
                    a.v = j;
                q.push(a);
            }
        }
    }

    // We reach here when 'qe' has last vertex
    // return the distance of vertex in 'qe'
    return qe.dist;
}

// Driver program to test methods of graph class
int main()
{
    // Let us construct the board given in above diagram
    int N = 30;
    int moves[N];
    for (int i = 0; i<N; i++)
        moves[i] = -1;

    // Ladders
    moves[2] = 21;
    moves[4] = 7;
    moves[10] = 25;
```

```
    moves[19] = 28;

    // Snakes
    moves[26] = 0;
    moves[20] = 8;
    moves[16] = 3;
    moves[18] = 6;

    cout << "Min Dice throws required is " << getMinDiceThrows(moves, N);
    return 0;
}
```

Run on IDE

Output:

```
Min Dice throws required is 3
```

Time complexity of the above solution is O(N) as every cell is added and removed only once from queue. And a typical enqueue or dequeue operation takes O(1) time.