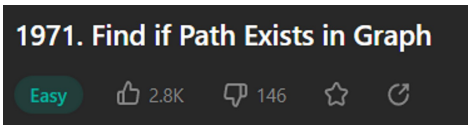
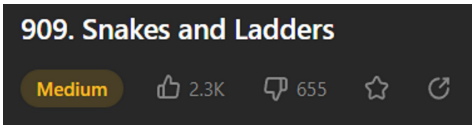
G-06 BFS Problem Type-1 II Simple BFS II Snakes an Ladders



https://leetcode.com/problems/find-if-path-exists-in-graph/



https://leetcode.com/problems/snakes-and-ladders/

909. Snakes and Ladders

Medium 2.3K 655

Companies

You are given an `n x n` integer matrix `board` where the cells are labeled from `1` to n^2 in a **Boustrophedon style** starting from the bottom left of the board (i.e. `board[n - 1][0]`) and alternating direction each row.

You start on square `1` of the board. In each move, starting from square `curr`, do the following:

- Choose a destination square `next` with a label in the range `[curr + 1, min(curr + 6, n^2)]`.
  - This choice simulates the result of a standard **6-sided die roll**: i.e., there are always at most 6 destinations, regardless of the size of the board.
- If `next` has a snake or ladder, you **must** move to the destination of that snake or ladder. Otherwise, you move to `next`.
- The game ends when you reach the square n^2.

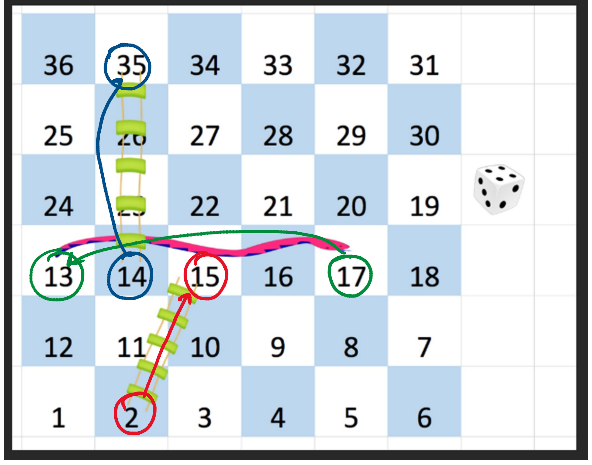
A board square on row `r` and column `c` has a snake or ladder if `board[r][c] != -1`. The destination of that snake or ladder is `board[r][c]`. Squares `1` and n^2 do not have a snake or ladder.

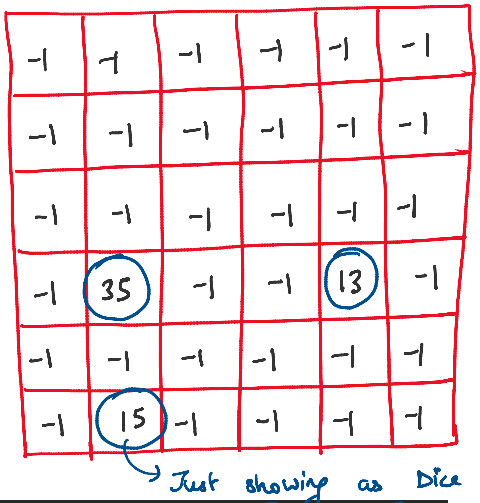
Note that you only take a snake or ladder at most once per move. If the destination to a snake or ladder is the start of another snake or ladder, you do **not** follow the subsequent snake or ladder.

- For example, suppose the board is `[[-1,4],[-1,3]]`, and on the first move, your destination square is `2`. You follow the ladder to square `3`, but do **not** follow the subsequent ladder to `4`.

Return *the least number of moves required to reach the square* n^2. *If it is not possible to reach the square, return* `-1`.

Example 1:





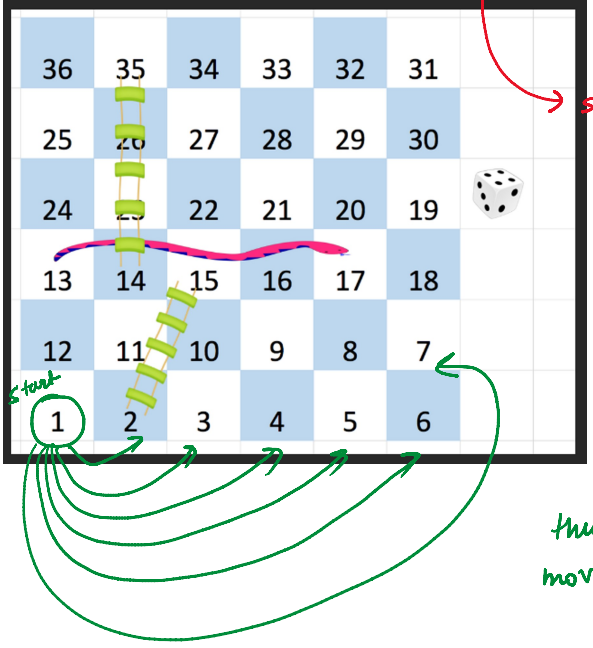
Just showing as Dice arrives cell (2) it will reach cell (15).

Input: board = [[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,35,-1,-1,13,-1],[-1,-1,-1,-1,-1,-1],[-1,15,-1,-1,-1,-1]]
Output: 4
Explanation:
In the beginning, you start at square 1 (at row 5, column 0).
You decide to move to square 2 and must take the ladder to square 15.
You then decide to move to square 17 and must take the snake to square 13.
You then decide to move to square 14 and must take the ladder to square 35.
You then decide to move to square 36, ending the game.
This is the lowest possible number of moves to reach the last square, so return 4.

Return the *least number of moves* required to reach *the square* n^2. If it is not possible to reach the square, return -1.

→ Reach n^2 from 1

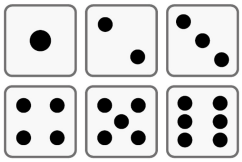
Example 1:



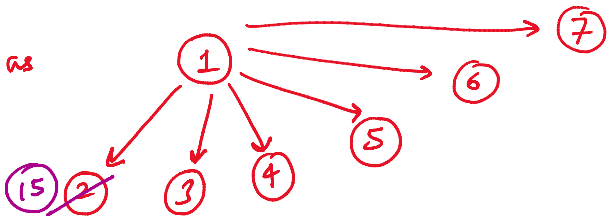
shortest distance from 1 to n^2.

As I role a dice possible options are :-

thus 9 can move 1, 2, 3, 4, 5, 6 steps....



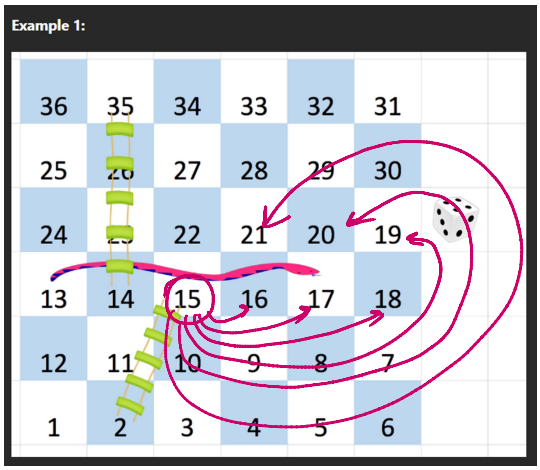
↳ In 1 role of Dice we can show the same thing as

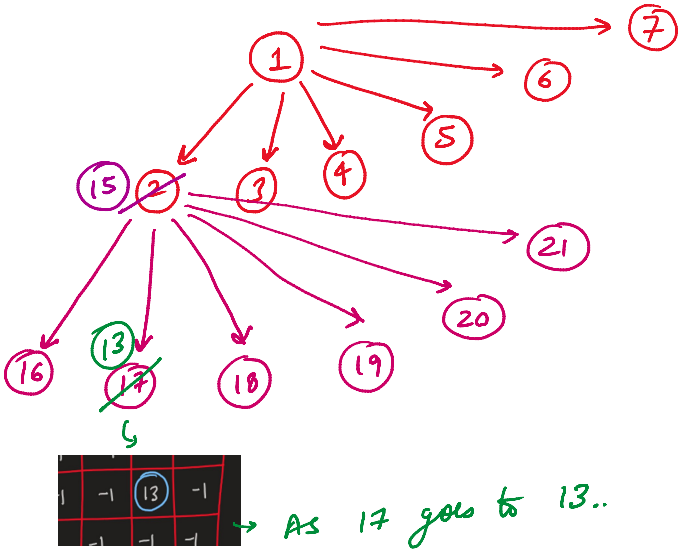


As (2) goes to (15) ⇐

thus replacing its value.

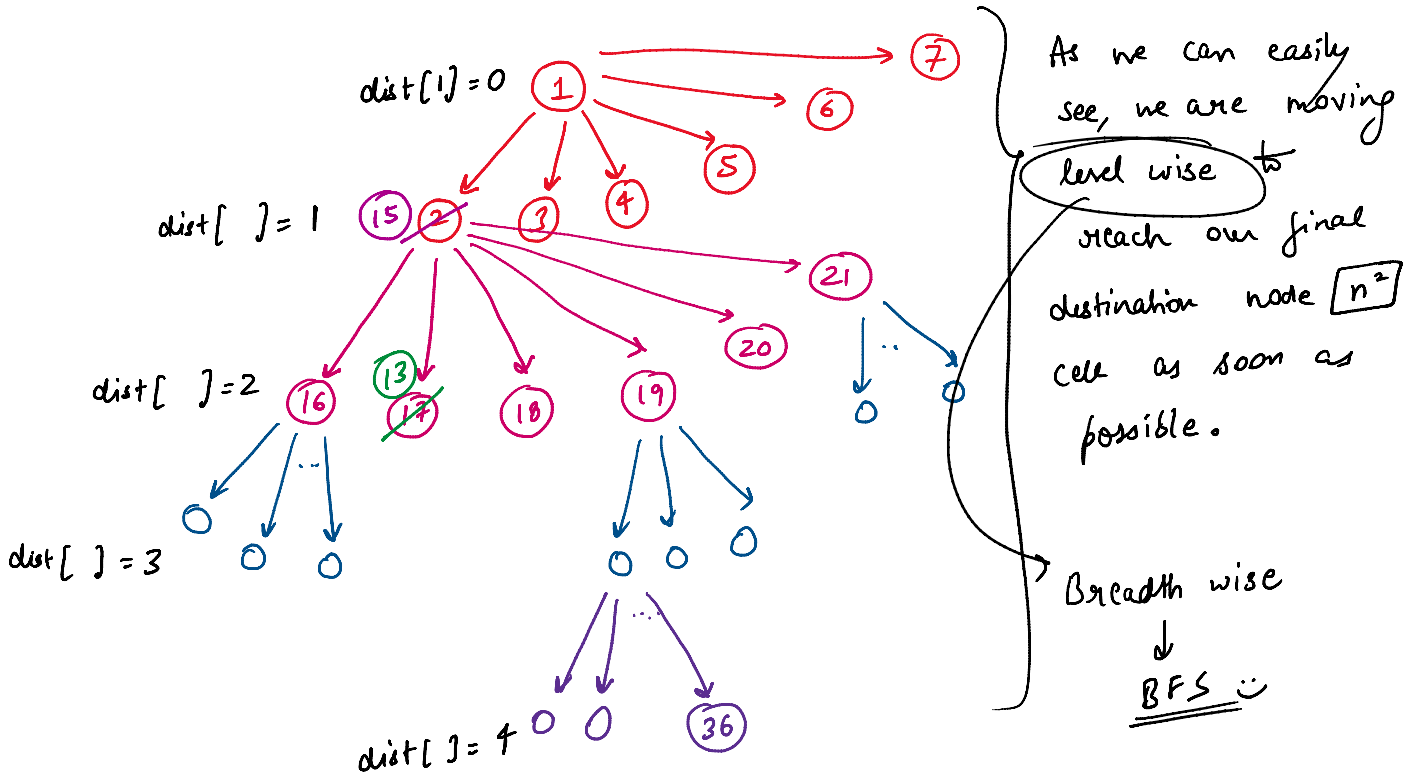
↳ In 2nd role of Dice it can move from all its possible locations.



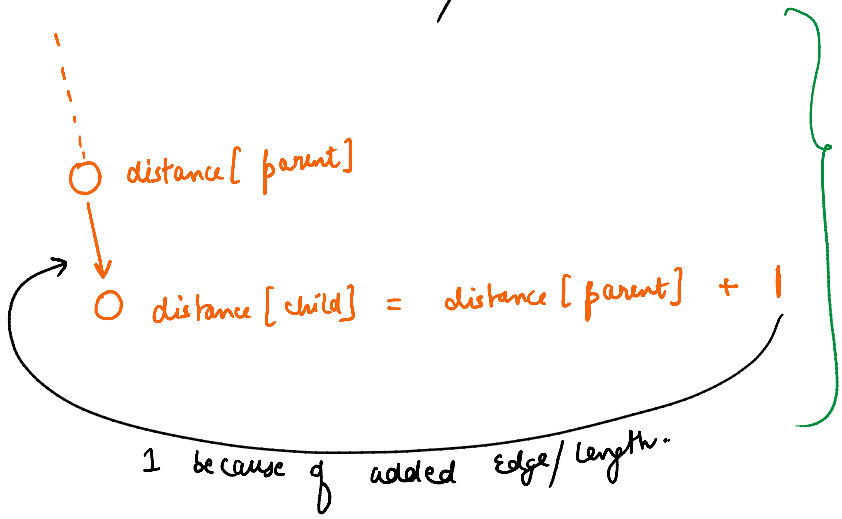


→ As 17 goes to 13..

↳ Thus we keep on rolling until we finally reach n^2 cell.



⇒ we maintain distance array, as we move down to the child.

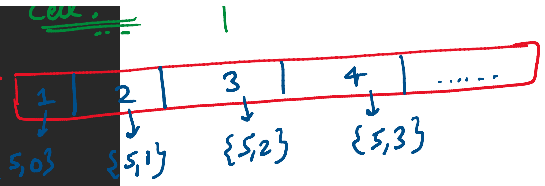


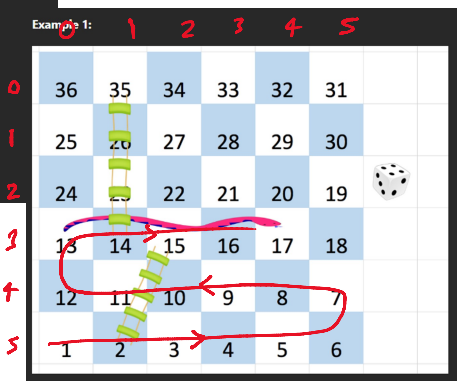
To go from cell Number to actual Board Cell.

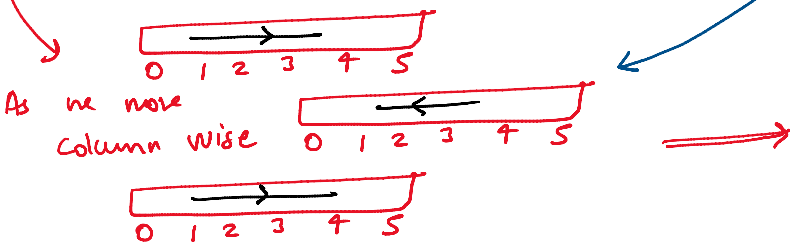
```
class Solution {
public:
```

```
class Solution {
public:
    int snakesAndLadders(vector<vector<int>> &board) {
        int n = board.size(), lbl = 1;
        vector<pair<int, int>> cells(n*n+1);
        vector<int> columns(n);
        for(int i=0;i<n;i++) columns[i] = i;
        for (int row = n - 1; row >= 0; row--) {
            for (int column : columns) {
                cells[lbl++] = {row, column};
            }
            reverse(columns.begin(), columns.end());
        }
```

actual Board cell.

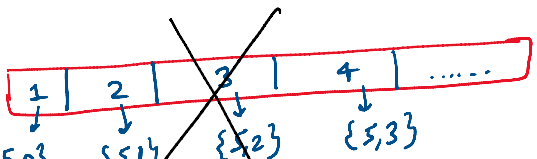







```
pair<int,int> pos (int x) {
        int r = (x-1) / n;
        int c = (x-1) % n;
        if (r&1)    c = (n-1)-c;
       r = (n-1)-r;
        return {r,c};
    }
```

Quick hack to actually grab the co-ordinates by just cell number.




```
vector<pair<int, int>> cells(n*n+1);
```

This would'nt be required !!

```
    vector<int> dist(n*n+1, -1);
    dist[1] = 0;
    queue<int> q;
    q.push(1);
    while (!q.empty()) {
        int curr = q.front();
        q.pop();
        for (int next = curr + 1; next <= min(curr+6, n*n); next++) {
            auto [row, column] = cells[next];
            int destination = board[row][column] != -1 ? board[row][column] : next;
```

Distance array to get Min distance to that node from 1.

Starting Node

Standard BFS

Number of children can be At Max 6

known

```
            for (int next = curr + 1; next <= min(curr+6, n*n); next++) {
                auto [row, column] = cells[next];
                int destination = board[row][column] != -1 ? board[row][column] : next;
                if (dist[destination] == -1) {
                    dist[destination] = dist[curr] + 1;
                    q.push(destination);
                }
            }
        }

        return dist[n*n];
    }
};
```

Grab row & column to know if that cell has some Ladder or Snake ☺

If already not visited (i.e. dist == -1)

↓

Just mark distance as parent + 1 as it will be shortest.

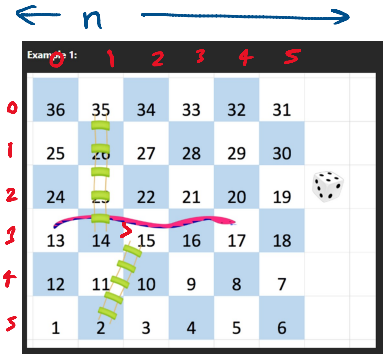
Return this distance to reach n^2 cell.

time : $O(n^2)$ → Going on to every cell via BFS.

space :- $O(n^2)$

↳ Primarily distance array / cells array ...

↳ Not required though.



C++ Auto

```cpp
class Solution {
public:
    int snakesAndLadders(vector<vector<int>> &board) {
        int n = board.size(), lbl = 1;
        vector<pair<int, int>> cells(n*n+1);
        vector<int> columns(n);
        for(int i=0;i<n;i++) columns[i] = i;
        for (int row = n - 1; row >= 0; row--) {
            for (int column : columns) {
                cells[lbl++] = {row, column};
            }
            reverse(columns.begin(), columns.end());
        }

        vector<int> dist(n*n+1, -1);
        dist[1] = 0;
        queue<int> q;
        q.push(1);
        while (!q.empty()) {
            int curr = q.front();
            q.pop();
            for (int next = curr + 1; next <= min(curr+6, n*n); next++) {
                auto [row, column] = cells[next];
                int destination = board[row][column] != -1 ? board[row][column] : next;
                if (dist[destination] == -1) {
                    dist[destination] = dist[curr] + 1;
                    q.push(destination);
                }
            }
        }

        return dist[n*n];
    }
};
```


```cpp
class Solution {
public:
    int snakesAndLadders(vector<vector<int>> &board) {
        int n = board.size(), lbl = 1;
        vector<pair<int, int>> cells(n*n+1);
        vector<int> columns(n);
        for(int i=0;i<n;i++) columns[i] = i;
        for (int row = n - 1; row >= 0; row--) {
            for (int column : columns) {
                cells[lbl++] = {row, column};
            }
            reverse(columns.begin(), columns.end());
        }
        vector<int> dist(n*n+1, -1);
        dist[1] = 0;
        queue<int> q;
        q.push(1);
```

```
        while (!q.empty()) {
            int curr = q.front();
            q.pop();
            for (int next = curr + 1; next <= min(curr+6, n*n); next++) {
                auto [row, column] = cells[next];
                int destination = board[row][column] != -1 ? board[row][column] : next;
                if (dist[destination] == -1) {
                    dist[destination] = dist[curr] + 1;
                    q.push(destination);
                }
            }
        }

        return dist[n*n];
    }
};
```

Java Auto

```java
import java.util.*;

class Solution {
    public int snakesAndLadders(int[][] board) {
        int n = board.length, lbl = 1;
        List<Pair<Integer, Integer>> cells = new ArrayList<>();
        Integer[] columns = new Integer[n];
        for (int i = 0; i < n; i++) columns[i] = i;


        cells.add(new Pair<>(0, 0));
        for (int row = n - 1; row >= 0; row--) {
            for (int column : columns) {
                cells.add(new Pair<>(row, column));
            }
            Collections.reverse(Arrays.asList(columns));
        }

        int[] dist = new int[n * n + 1];
        Arrays.fill(dist, -1);
        dist[1] = 0;
        Queue<Integer> q = new LinkedList<>();
        q.add(1);
        while (!q.isEmpty()) {
            int curr = q.poll();
            for (int next = curr + 1; next <= Math.min(curr + 6, n * n); next++) {
                Pair<Integer, Integer> cell = cells.get(next);
                int row = cell.getKey();
                int column = cell.getValue();
                int destination = board[row][column] != -1 ? board[row][column] : next;
                if (dist[destination] == -1) {
                    dist[destination] = dist[curr] + 1;
                    q.add(destination);
                }
            }
        }

        return dist[n * n];
    }
}
```


```java
import java.util.*;
class Solution {
    public int snakesAndLadders(int[][] board) {
        int n = board.length, lbl = 1;
        List<Pair<Integer, Integer>> cells = new ArrayList<>();
        Integer[] columns = new Integer[n];
        for (int i = 0; i < n; i++) columns[i] = i;

        cells.add(new Pair<>(0, 0));
        for (int row = n - 1; row >= 0; row--) {
            for (int column : columns) {
                cells.add(new Pair<>(row, column));
            }
            Collections.reverse(Arrays.asList(columns));
        }
        int[] dist = new int[n * n + 1];
```

```
        Arrays.fill(dist, -1);
        dist[1] = 0;
        Queue<Integer> q = new LinkedList<>();
        q.add(1);
        while (!q.isEmpty()) {
            int curr = q.poll();
            for (int next = curr + 1; next <= Math.min(curr + 6, n * n); next++) {
                Pair<Integer, Integer> cell = cells.get(next);
                int row = cell.getKey();
                int column = cell.getValue();
                int destination = board[row][column] != -1 ? board[row][column] : next;
                if (dist[destination] == -1) {
                    dist[destination] = dist[curr] + 1;
                    q.add(destination);
                }
            }
        }
        return dist[n * n];
    }
}
```