

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

1971. Find if Path Exists in Graph

Easy

2.8K

146



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

909. Snakes and Ladders

Medium

2.3K

655



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

909. Snakes and Ladders

Medium



2.3K

655



Companies

You are given an $n \times n$ 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 .

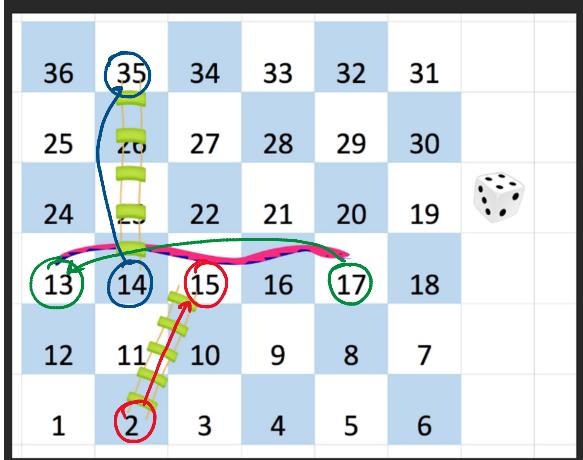
A board square on row r and column c has a snake or ladder if `board[r][c] != -1`. The destination of that snake or ladder is `board[r][c]`. Squares 1 and n^2 do not have a snake or ladder.

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 $\begin{bmatrix} [-1, 4], [-1, 3] \end{bmatrix}$, 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:



-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	15	-1	-1	-1	-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.

Example 1:

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

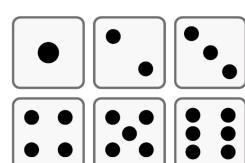
Reach n^2 from 1

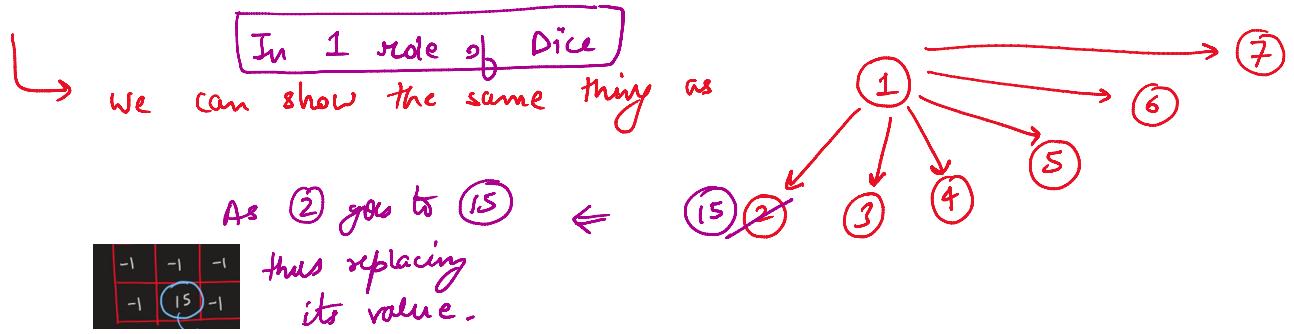


shortest
distance from
1 to n^2 .

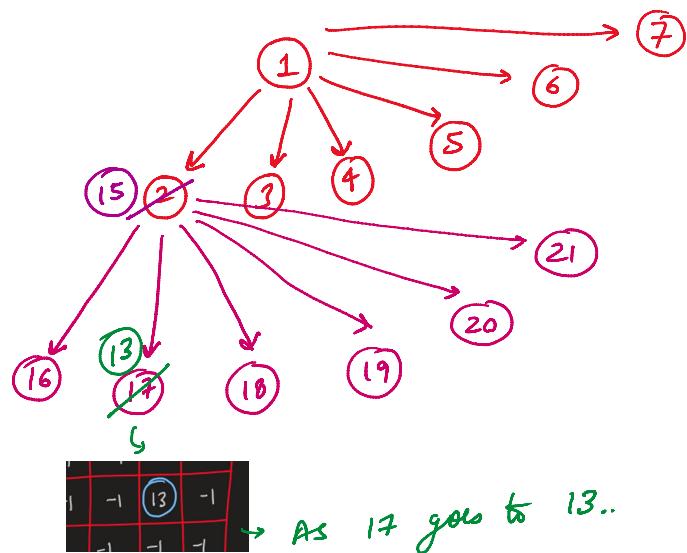
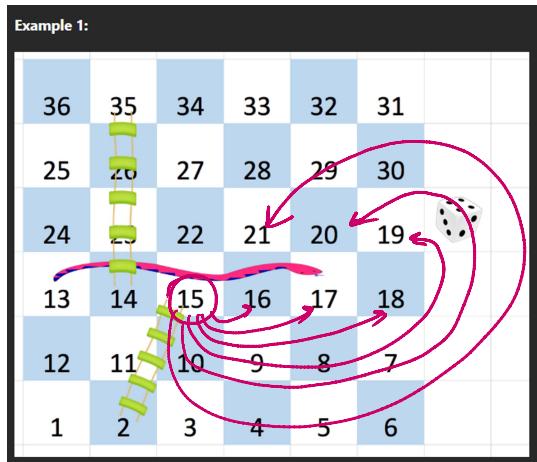
As I role a dice possible options are :-

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

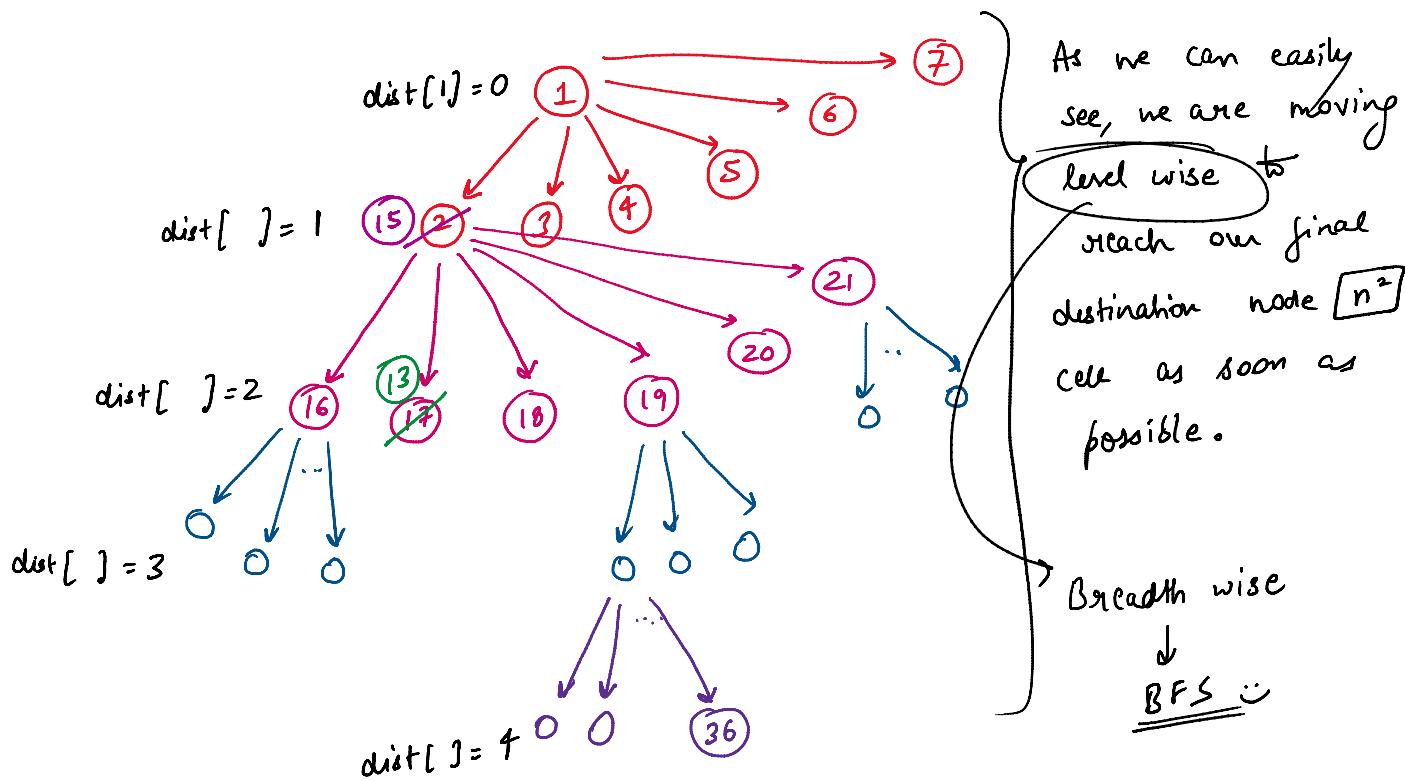




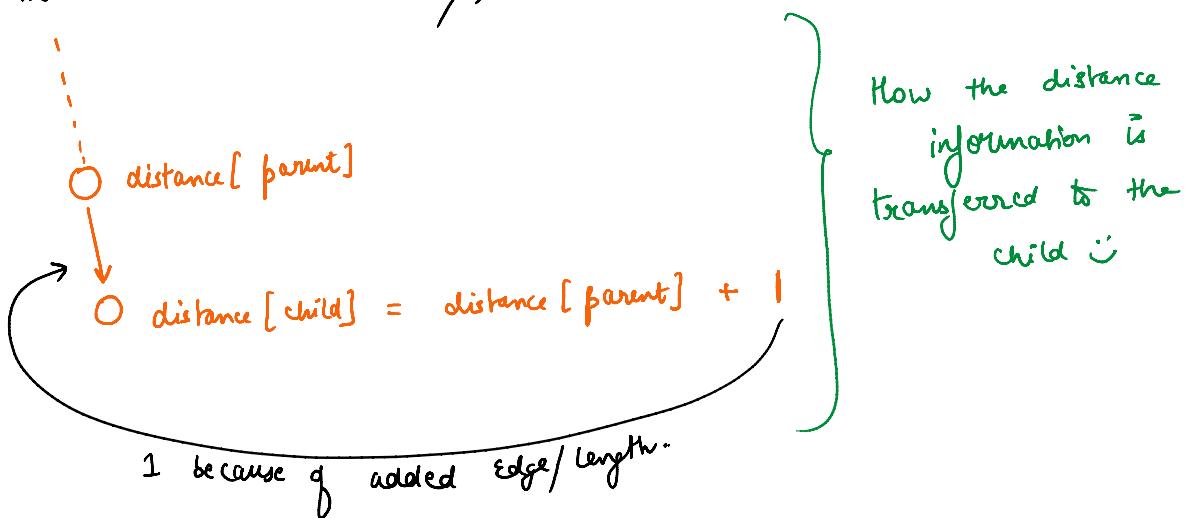
↳ In 2nd side of Dice, it can move from all its possible locations.



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



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



```
class Solution {
public:
    // Implementation details
}
```

To go from cell Number to actual Board Cell :

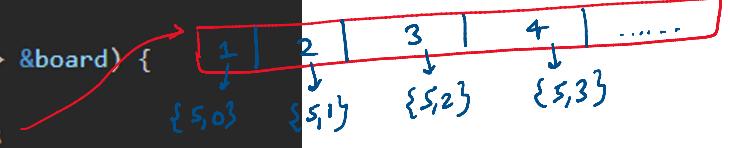
1 | 2 | 3 | 4 | ...

```
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 :-

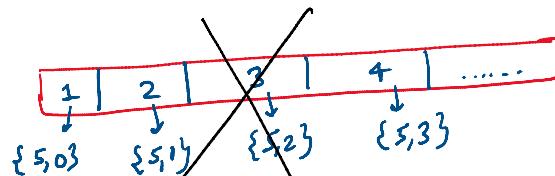


As we move



```
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 back to actually get the co-ordinates by just cell number.



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

This wouldn't be required !!

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

```
vector<int> dist(n*n+1, -1);
dist[1] = 0; starting Node
queue<int> q;
q.push(1);
while (!q.empty()) { Standard BFS
    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) { know
            dist[destination] = dist[curr] + 1;
            q.push(destination);
        }
    }
}
```

Number of children can be At Max 6

```

        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];
};



Return Min distance to reach  $n^2$  cell.



if already not visited (i.e. dist == -1)  
Just mark distance as parent + 1 as it will be shortest.



Grab row & column to know if that cell has some ladder or snake :-)


```

time: $O(n^2)$ going on to every cell via BFS!

Space: $O(n^2)$

↳ Primarily distance array / cells array ...

$\leftarrow n \rightarrow$

Example 1:						
0	1	2	3	4	5	
0	36	35	34	33	32	31
1	25	20	27	28	29	30
2	24	23	22	21	20	19
3	13	14	15	16	17	18
4	12	11	10	9	8	7
5	1	2	3	4	5	6

Not required though.

```
i C++ ▾ • Auto
```

```
1 class Solution {
2 public:
3     int snakesAndLadders(vector<vector<int>> &board) {
4         int n = board.size(), lbl = 1;
5         vector<pair<int, int>> cells(n*n+1);
6         vector<int> columns(n);
7         for(int i=0;i<n;i++) columns[i] = i;
8         for (int row = n - 1; row >= 0; row--) {
9             for (int column : columns) {
10                 cells[lbl++] = {row, column};
11             }
12         reverse(columns.begin(), columns.end());
13     }
14
15     vector<int> dist(n*n+1, -1);
16     dist[1] = 0;
17     queue<int> q;
18     q.push(1);
19     while (!q.empty()) {
20         int curr = q.front();
21         q.pop();
22         for (int next = curr + 1; next <= min(curr+6, n*n); next++) {
23             auto [row, column] = cells[next];
24             int destination = board[row][column] != -1 ? board[row][column] : next;
25             if (dist[destination] == -1) {
26                 dist[destination] = dist[curr] + 1;
27                 q.push(destination);
28             }
29         }
30     }
31     return dist[n*n];
32 }
33 };
34 }
```

```
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];
};
```

The screenshot shows a Java code editor interface with a dark theme. The top bar includes tabs for 'Java' and 'Auto', and icons for search, file operations, and help. The main area displays the following Java code:

```
1 import java.util.*;
2
3 class Solution {
4     public int snakesAndLadders(int[][] board) {
5         int n = board.length, lbl = 1;
6         List<Pair<Integer, Integer>> cells = new ArrayList<>();
7         Integer[] columns = new Integer[n];
8         for (int i = 0; i < n; i++) columns[i] = i;
9
10
11        cells.add(new Pair<>(0, 0));
12        for (int row = n - 1; row >= 0; row--) {
13            for (int column : columns) {
14                cells.add(new Pair<>(row, column));
15            }
16            Collections.reverse(Arrays.asList(columns));
17        }
18
19        int[] dist = new int[n * n + 1];
20        Arrays.fill(dist, -1);
21        dist[1] = 0;
22        Queue<Integer> q = new LinkedList<>();
23        q.add(1);
24        while (!q.isEmpty()) {
25            int curr = q.poll();
26            for (int next = curr + 1; next <= Math.min(curr + 6, n * n); next++) {
27                Pair<Integer, Integer> cell = cells.get(next);
28                int row = cell.getKey();
29                int column = cell.getValue();
30                int destination = board[row][column] != -1 ? board[row][column] : next;
31                if (dist[destination] == -1) {
32                    dist[destination] = dist[curr] + 1;
33                    q.add(destination);
34                }
35            }
36        }
37
38        return dist[n * n];
39    }
40 }
```

```
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];
    }
}
```