115. Unique Paths II ☆

Description | Notes | >_ Testcase | Judge

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as `1` and `0` respectively in the grid.

Notice

m and *n* will be at most 100.

Have you met this question in a real interview? Yes

Example

Example

For example,

There is one obstacle in the middle of a 3x3 grid as illustrated below.

```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```

The total number of unique paths is `2`.

```
public class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        if (obstacleGrid == null || obstacleGrid.length == 0 || obstacleGrid[0].length == 0) {
            return 0;
        }

        int n = obstacleGrid.length;
        int m = obstacleGrid[0].length;
        int[][] paths = new int[n][m];

        for (int i = 0; i < n; i++) {
            if (obstacleGrid[i][0] != 1) {
                paths[i][0] = 1;
            } else {
                break;
            }
        }

        for (int i = 0; i < m; i++) {
            if (obstacleGrid[0][i] != 1) {
                paths[0][i] = 1;
            } else {
                break;
            }
        }

        for (int i = 1; i < n; i++) {
            for (int j = 1; j < m; j++) {
                if (obstacleGrid[i][j] != 1) {
                    paths[i][j] = paths[i - 1][j] + paths[i][j - 1];
                } else {
                    paths[i][j] = 0;
                }
            }
        }

        return paths[n - 1][m - 1];
    }
}

// 方法二
public class Solution {
    /**
     * @param obstacleGrid: A list of lists of integers
     * @return: An integer
```

```
     */
    public int uniquePathsWithObstacles(int[][] A) {
        int m = A.length;
        if (m == 0) {
            return 0;
        }
        int n = A[0].length;
        if (n == 0) {
            return 0;
        }

        if (A[0][0] == 1 || A[m-1][n-1] == 1) {
            return 0;
        }

        int[][] f = new int[2][n];
        int i, j, old, now;
        now = 0;

        for (i = 0; i < m; ++i) {
            old = now;
            now = 1 - now;
            for (j = 0; j < n; ++j) {
                f[now][j] = 0;
                if (A[i][j] == 1) {
                    f[now][j] = 0;
                }
                else {
                    if (i == 0 && j == 0) {
                        f[now][j] = 1;
                    }
                    if (i > 0) {
                        f[now][j] += f[old][j];
                    }
                    if (j > 0) {
                        f[now][j] += f[now][j-1];
                    }
                }
            }
        }

        return f[now][n-1];
    }
}
```

114. Unique Paths ☆

Description | Notes | >_ Testcase | Judge

A robot is located at the top-left corner of a *m* x *n* grid.

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid.

How many possible unique paths are there?

Notice

m and *n* will be at most 100.

Have you met this question in a real interview? Yes

Example

Given m = `3` and n = `3`, return `6`.
Given m = `4` and n = `5`, return `35`.

```
public class Solution {
    /*
     * @param m: positive integer (1 <= m <= 100)
     * @param n: positive integer (1 <= n <= 100)
     * @return: An integer
     */
    public int uniquePaths(int m, int n) {
        // write your code here
        if (m == 0 || n == 0) {
            return 1;
        }

        int[][] sum = new int[m][n];
        for (int i = 0; i < m; i++) {
            sum[i][0] = 1;
        }
        for (int i = 0; i < n; i++) {
            sum[0][i] = 1;
        }
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                sum[i][j] = sum[i - 1][j] + sum[i][j - 1];
            }
        }
        return sum[m - 1][n - 1];
    }
}

// public class Solution {
//     /**
//      * @param n, m: positive integer (1 <= n ,m <= 100)
//      * @return an integer
//      */
//     public int uniquePaths(int m, int n) {
//         int[][] f = new int[m][n];
//         int i, j;
//         for (i = 0; i < m; ++i) {
//             for (j = 0; j < n; ++j) {
//                 if (i == 0 || j == 0) {
//                     f[i][j] = 1;
//                 }
//                 else {
//                     f[i][j] = f[i-1][j] + f[i][j-1];
//                 }
```

```
//             }
//         }

//         return f[m-1][n-1];
//     }
// }
```

111. Climbing Stairs ☆

Description | Notes | >_ Testcase | Judge

You are climbing a stair case. It takes ***n*** steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Have you met this question in a real interview? Yes

Example

Given an example n=3 , 1+1+1=2+1=1+2=3

return 3

Tags

```
public class Solution {
    /**
     * @param n: An integer
     * @return: An integer
     */
    public int climbStairs(int n) {
        // write your code here
         if (n <= 1) {
            return n;
        }
        int last = 1, lastlast = 1;
        int now = 0;
        for (int i = 2; i <= n; i++) {
            now = last + lastlast;
            lastlast = last;
            last = now;
        }
        return now;
    }
}
```

110. Minimum Path Sum ☆

Description | Notes | Testcase | Judge

Given a ***m* x *n*** grid filled with non-negative numbers, find a path from top left to bottom right which ***minimizes*** the sum of all numbers along its path.

i Notice

You can only move either down or right at any point in time.

Have you met this question in a real interview? Yes

Example

```
public class Solution {
    /*
     * @param grid: a list of lists of integers
     * @return: An integer, minimizes the sum of all numbers along its path
     */
    public int minPathSum(int[][] grid) {
        // write your code here
        if (grid == null || grid.length == 0 || grid[0].length == 0) {
            return 0;
        }

        int M = grid.length;
        int N = grid[0].length;
        int[][] sum = new int[M][N];

        sum[0][0] = grid[0][0];

        for (int i = 1; i < M; i++) {
            sum[i][0] = sum[i - 1][0] + grid[i][0];
        }

        for (int i = 1; i < N; i++) {
            sum[0][i] = sum[0][i - 1] + grid[0][i];
        }

        for (int i = 1; i < M; i++) {
            for (int j = 1; j < N; j++) {
                sum[i][j] = Math.min(sum[i - 1][j], sum[i][j - 1]) + grid[i][j];
            }
        }

        return sum[M - 1][N - 1];
    }
}
```

109. Triangle ☆

Description | Notes | >_ Testcase | Judge

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

i Notice

Bonus point if you are able to do this using only O(n) extra space, where n is the total number of rows in the triangle.

Have you met this question in a real interview? Yes

Example

Given the following triangle:

```
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

```
public class Solution {
    /*
     * @param triangle: a list of lists of integers
     * @return: An integer, minimum path sum
     */
    public int minimumTotal(int[][] triangle) {
        // write your code here
            if (triangle == null || triangle.length == 0) {
            return -1;
        }
        if (triangle[0] == null || triangle[0].length == 0) {
            return -1;
        }

        // state: f[x][y] = minimum path value from 0,0 to x,y
        int n = triangle.length;
        int[][] f = new int[n][n];

        // initialize
        f[0][0] = triangle[0][0];
        for (int i = 1; i < n; i++) {
            f[i][0] = f[i - 1][0] + triangle[i][0];
            f[i][i] = f[i - 1][i - 1] + triangle[i][i];
        }

        // top down
        for (int i = 1; i < n; i++) {
            for (int j = 1; j < i; j++) {
                f[i][j] = Math.min(f[i - 1][j], f[i - 1][j - 1]) + triangle[i][j];
            }
        }

        // answer
        int best = f[n - 1][0];
        for (int i = 1; i < n; i++) {
            best = Math.min(best, f[n - 1][i]);
        }
        return best;
    }
}
```

603. Largest Divisible Subset ☆

Description | Notes | >_ Testcase | Judge

Given a set of `distinct positive` integers, find the largest subset such that every pair `(Si, Sj)` of elements in this subset satisfies: `Si % Sj = 0` or `Sj % Si = 0`.

Notice

If there are multiple solutions, return any subset is fine.

Have you met this question in a real interview? Yes

Example

Given nums = `[1,2,3]`, return `[1,2]` or `[1,3]`

Given nums = `[1,2,4,8]`, return `[1,2,4,8]`

```
public class Solution {
    /*
     * @param nums: a set of distinct positive integers
     * @return: the largest subset
     */
    public List<Integer> largestDivisibleSubset(int[] nums) {
        // write your code here
        Arrays.sort(nums);
        int[] f = new int[nums.length];
        int[] pre = new int[nums.length];
        for (int i = 0; i < nums.length; i++) {
            f[i] = 1;
            pre[i] = i;
            for (int j = 0; j < i; j++) {
                if (nums[i] % nums[j] == 0 && f[i] < f[j] + 1) {
                    f[i] = f[j] + 1;
                    pre[i] = j;
                }
            }
        }

        List<Integer> ans = new ArrayList<Integer>();
        if (nums.length == 0) {
            return ans;
        }
        int max = 0;
        int max_i = 0;
        for (int i = 0; i < nums.length; i++) {
            if (f[i] > max) {
                max = f[i];
                max_i = i;
            }
        }
        ans.add(nums[max_i]);
        while (max_i != pre[max_i]) {
            max_i = pre[max_i];
            ans.add(nums[max_i]);
        }
        Collections.reverse(ans);
        return ans;
    }
}
```

513. Perfect Squares ☆

Description | Notes | >_ Testcase | Judge

Given a positive integer `n`, find the least number of perfect square numbers (for example, `1, 4, 9, 16, ...`) which sum to n.

Have you met this question in a real interview? Yes

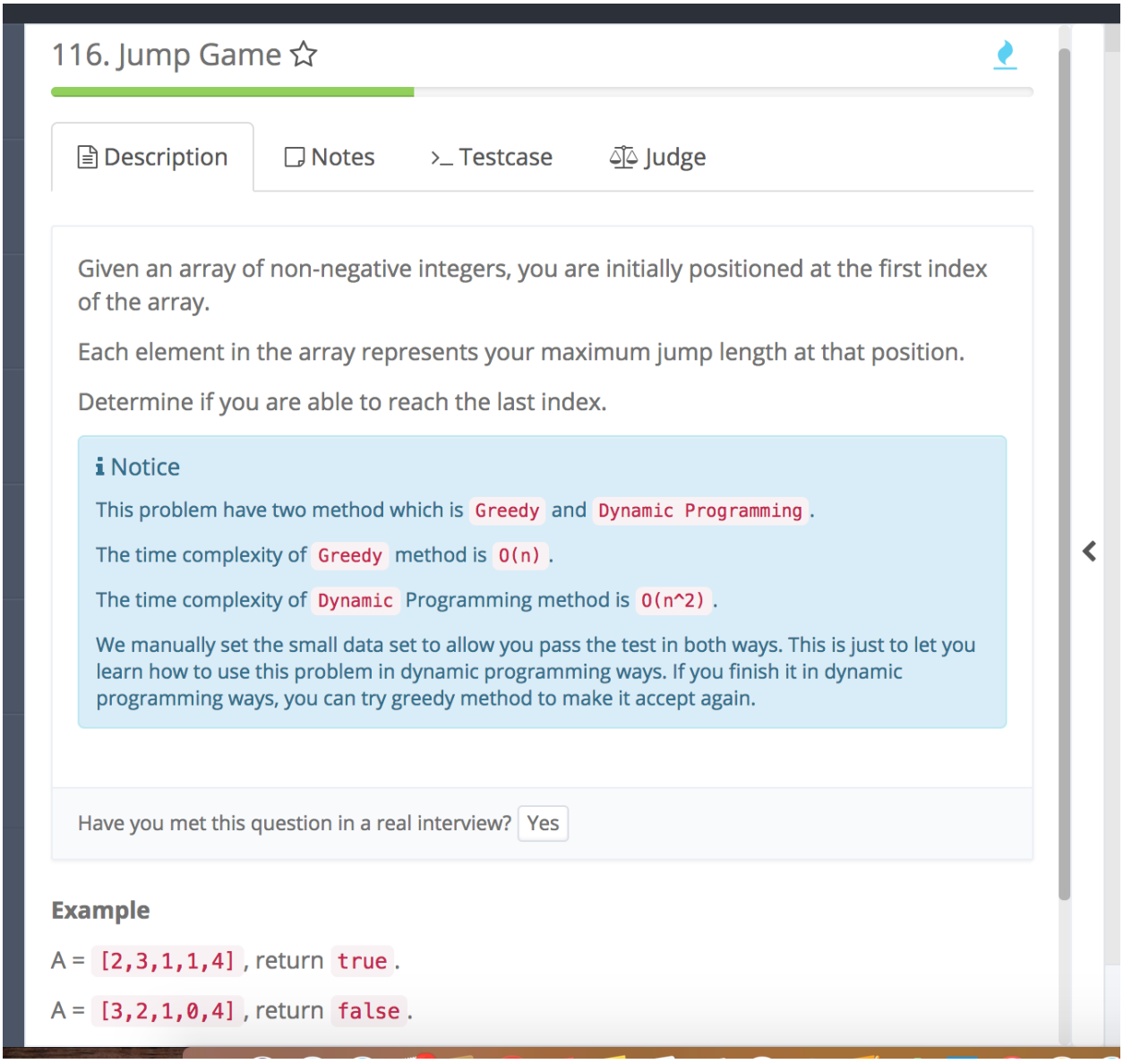
Example

Given n = `12`, return `3` because `12 = 4 + 4 + 4`
Given n = `13`, return `2` because `13 = 4 + 9`

```
public class Solution {
    /*
     * @param n: a positive integer
     * @return: An integer
     */
    public int numSquares(int n) {
        // write your code here
        int[] dp = new int[n + 1];
        Arrays.fill(dp, Integer.MAX_VALUE);
        for(int i = 0; i * i <= n; ++i) {
            dp[i * i] = 1;
        }

        for (int i = 0; i <= n; ++i) {
            for (int j = 1; j * j <= i; ++j) {
                dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
            }
        }

        return dp[n];
    }
}
```



116. Jump Game ☆

Description | Notes | >_ Testcase | Judge

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

Notice

This problem have two method which is `Greedy` and `Dynamic Programming`.

The time complexity of `Greedy` method is `O(n)`.

The time complexity of `Dynamic` Programming method is `O(n^2)`.

We manually set the small data set to allow you pass the test in both ways. This is just to let you learn how to use this problem in dynamic programming ways. If you finish it in dynamic programming ways, you can try greedy method to make it accept again.

Have you met this question in a real interview? Yes

Example

A = `[2,3,1,1,4]`, return `true`.

A = `[3,2,1,0,4]`, return `false`.

```
// 这个方法，复杂度是 O(n^2) 可能会超时，但是依然需要掌握。
public class Solution {
    public boolean canJump(int[] A) {
        boolean[] can = new boolean[A.length];
        can[0] = true;

        for (int i = 1; i < A.length; i++) {
            for (int j = 0; j < i; j++) {
                if (can[j] && j + A[j] >= i) {
                    can[i] = true;
                    break;
                }
            }
        }

        return can[A.length - 1];
    }
}


// version 2: Greedy
public class Solution {
    public boolean canJump(int[] A) {
        // think it as merging n intervals
        if (A == null || A.length == 0) {
            return false;
        }
        int farthest = A[0];
        for (int i = 1; i < A.length; i++) {
            if (i <= farthest && A[i] + i >= farthest) {
                farthest = A[i] + i;
            }
        }
        return farthest >= A.length - 1;
    }
}
```

76. Longest Increasing Subsequence ☆

Description | Notes | >_ Testcase | Judge

Given a sequence of integers, find the longest increasing subsequence (LIS).

You code should return the length of the LIS.

Have you met this question in a real interview? Yes

Clarification

What's the definition of longest increasing subsequence?

- The longest increasing subsequence problem is to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous, or unique.
- https://en.wikipedia.org/wiki/Longest_increasing_subsequence

Example

For `[5, 4, 1, 2, 3]`, the LIS is `[1, 2, 3]`, return `3`
For `[4, 2, 4, 5, 3, 7]`, the LIS is `[2, 4, 5, 7]`, return `4`

Challenge

```
public class Solution {
    /**
     * @param nums: The integer array
     * @return: The length of LIS (longest increasing subsequence)
     */


    public int longestIncreasingSubsequence(int[] nums) {
        int []f = new int[nums.length];
        int max = 0;
        for (int i = 0; i < nums.length; i++) {
            f[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[j] < nums[i]) {
                    f[i] = f[i] > f[j] + 1 ? f[i] : f[j] + 1;
                }
            }
            if (f[i] > max) {
                max = f[i];
            }
        }
        return max;
    }
}


// O(nlogn) Binary Search
public class Solution {
    /**
     * @param nums: The integer array
     * @return: The length of LIS (longest increasing subsequence)
     */
    public int longestIncreasingSubsequence(int[] nums) {
        int[] minLast = new int[nums.length + 1];
        minLast[0] = Integer.MIN_VALUE;
        for (int i = 1; i <= nums.length; i++) {
            minLast[i] = Integer.MAX_VALUE;
        }

        for (int i = 0; i < nums.length; i++) {
            // find the first number in minLast >= nums[i]
            int index = binarySearch(minLast, nums[i]);
            minLast[index] = nums[i];
        }
```

```
        for (int i = nums.length; i >= 1; i--) {
            if (minLast[i] != Integer.MAX_VALUE) {
                return i;
            }
        }

        return 0;
    }

    // find the first number > num
    private int binarySearch(int[] minLast, int num) {
        int start = 0, end = minLast.length - 1;
        while (start + 1 < end) {
            int mid = (end - start) / 2 + start;
            if (minLast[mid] < num) {
                start = mid;
            } else {
                end = mid;
            }
        }

        return end;
    }
}
```