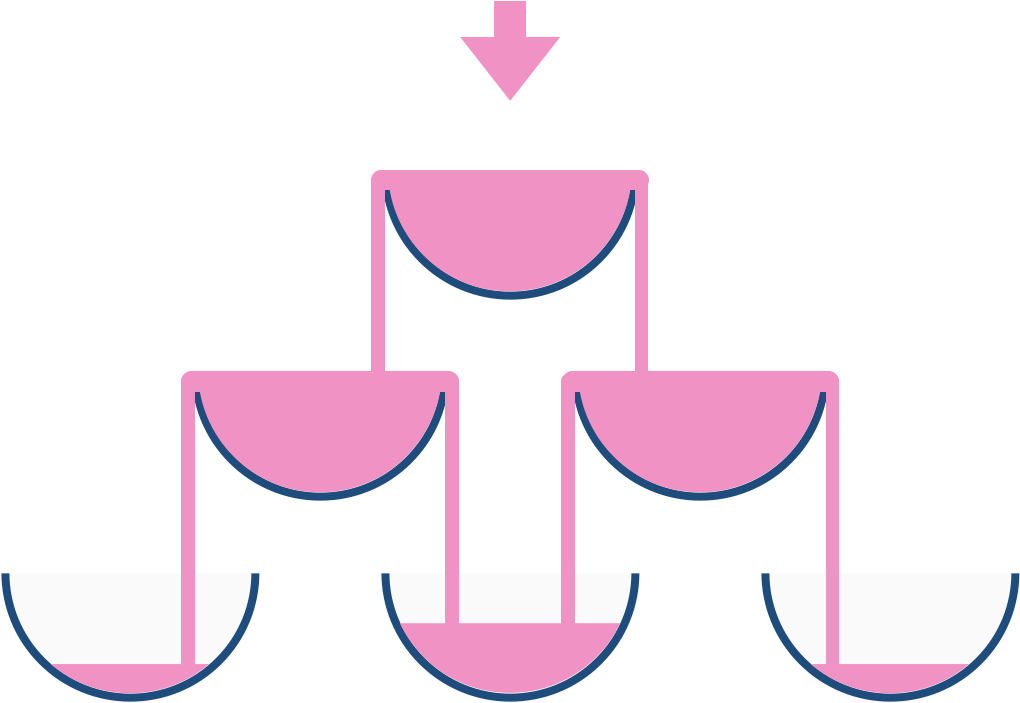
<https://leetcode.com/problems/champagne-tower/description/>

We stack glasses in a pyramid, where the first row has 1 glass, the second row has 2 glasses, and so on until the 100th row. Each glass holds one cup of champagne.

Then, some champagne is poured into the first glass at the top. When the topmost glass is full, any excess liquid poured will fall equally to the glass immediately to the left and right of it. When those glasses become full, any excess champagne will fall equally to the left and right of those glasses, and so on. (A glass at the bottom row has its excess champagne fall on the floor.)

For example, after one cup of champagne is poured, the top most glass is full. After two cups of champagne are poured, the two glasses on the second row are half full. After three cups of champagne are poured, those two cups become full - there are 3 full glasses total now. After four cups of champagne are poured, the third row has the middle glass half full, and the two outside glasses are a quarter full, as pictured below.



Now after pouring some non-negative integer cups of champagne, return how full the jth glass in the ith row is (both i and j are 0-indexed.)

**Example 1:**

**Input:** poured = 1, query\_row = 1, query\_glass = 1

**Output:** 0.00000

**Explanation:** We poured 1 cup of champange to the top glass of the tower (which is indexed as (0, 0)). There will be no excess liquid so all the glasses under the top glass will remain empty.

**Example 2:**

**Input:** poured = 2, query\_row = 1, query\_glass = 1

**Output:** 0.50000

**Explanation:** We poured 2 cups of champange to the top glass of the tower (which is indexed as (0, 0)). There is one cup of excess liquid. The glass indexed as (1, 0) and the glass indexed as (1, 1) will share the excess liquid equally, and each will get half cup of champange.

**Example 3:**

**Input:** poured = 100000009, query\_row = 33, query\_glass = 17

**Output:** 1.00000

**Constraints:**

0 <= poured <= 10^9

0 <= query\_glass <= query\_row < 100

**Attempt 1: 2024-12-03**

**Solution 1: Math + Fixed dimension 2D array (120 min)**

class Solution {

    public double champagneTower(int poured, int query\_row, int query\_glass) {

        // 0 <= query\_glass <= query\_row < 100

        double[][] result = new double[101][101];

        // Top glass of the tower index as [0, 0]

        result[0][0] = poured;

        // query\_row < 100

        for(int i = 0; i < 100; i++) {

            // 0 <= query\_glass <= query\_row

            for(int j = 0; j <= i; j++) {

// Check for overflow happen or not

                // If the glass has poured > 1, we should equally

                // split the diff (glass - 1) into next level

                if(result[i][j] > 1) {

                    result[i + 1][j] += (result[i][j] - 1) / 2.0;

                    result[i + 1][j + 1] += (result[i][j] - 1) / 2.0;

// Current glass capped at 1

                    result[i][j] = 1;

                }

            }

        }

        return result[query\_row][query\_glass];

    }

}

Time Complexity: O(n^2)

Space Complexity: O(n^2)

**Refer to**

<https://leetcode.com/problems/champagne-tower/solutions/118660/20ms-c-easy-understand-solution/>

We use a table to record the result.

Simple idea:

If the glass >=1, we should split the diff (glass - 1) into next level.

double champagneTower(int poured, int query\_row, int query\_glass) {

double result[101][101] = {0.0};

result[0][0] = poured;

for (int i = 0; i < 100; i++) {

for (int j = 0; j <= i; j++) {

if (result[i][j] >= 1) {

result[i + 1][j] += (result[i][j] - 1) / 2.0;

result[i + 1][j + 1] += (result[i][j] - 1) / 2.0;

result[i][j] = 1;

}

}

}

return result[query\_row][query\_glass];

}

**Solution 2: Math + Dynamic length 1D list (60 min, similar to L118.Pascal's Triangle)**

class Solution {

    public double champagneTower(int poured, int query\_row, int query\_glass) {

        // Initial pour into the first glass

        List<Double> curRow = new ArrayList<>();

        curRow.add((double) poured);

        for(int i = 0; i <= query\_row; i++) {

            List<Double> nextRow = new ArrayList<>();

            // Initialize the next row with 0 values next row

            // always has 1 more glass than value of 'query\_row'

            // as i, so range is (0, i + 1)

            for(int k = 0; k <= i + 1; k++) {

                nextRow.add(0.0);

            }

            // Process each glass in the current row

            // 0 <= query\_glass <= query\_row

            for(int j = 0; j <= i; j++) {

                // Check for overflow happen or not

                // If the glass has poured > 1, we should equally

                // split the diff (glass - 1) into next level

                if(curRow.get(j) > 1) {

                    double overflow = (curRow.get(j) - 1) / 2.0;

                    nextRow.set(j, nextRow.get(j) + overflow);

                    nextRow.set(j + 1, nextRow.get(j + 1) + overflow);

                    // Current glass capped at 1

                    curRow.set(j, 1.0);

                }

            }

            // Move to the next row if not at the query row

            if(i != query\_row) {

                curRow = nextRow;

            }

        }

        // Return champagne amount in the desired glass

        return curRow.get(query\_glass);

    }

}

Time Complexity: O(n^2)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/champagne-tower/solutions/1818599/full-visual-explanation-dp-beginner-friendly-easy-and-simple-c/>

We will be using **DP** here. But how?

Lets first understand the logic.

**LOGIC:**

• First of all, imagine that the whole champagne is being poured into the first row (1 glass in the first row)

• Now, whatever is being overflowed from this glass, goes to the 2 glasses just below this glass, equally, in the second row. Equally means, half of the champagne being overflowed goes to the first glass and half of that goes to the second glass.

• Now, in the second row, from each glass whatever is being overflowed goes to the 2 glasses just below each glass, equally, in the third row. So from each glass in second row, half of what is being overflowed goes to the left glass below it and half goes to the right glass.

• And this goes on...

**ALGORITHM:**

Using the above concept, we will make our dp matrix.

How will we implement our dp?

Lets see.

• We will first of all fill the first glass with the whole champagne. That means champagneInGlass[0][0]=poured

• Now we will check, if any champagne is being overflowed from this glass, then we will fill half of the overflowing champagne to the left glass and half to the right glass. That means, whatever is being poured into this glass, 1 cup will be stored in this glass and the rest will get overflowed and this overflowed champagne will get equally distributed among the 2 glass below this glass.

champagne stored in this glass = 1 cup (every glass can store only 1 cup of champagne)

champagne being overflowed from glass in ith row and jth column = champagneInGlass[j] of ith row - 1 (as 1 cup is stored in the glass and the rest is being overflowed)

champagne being poured into the left glass below it, because of this glass = champagne being overflowed / 2 = (champagneInGlass[j] of ith row-1) / 2

similarly, champagne being poured into the right glass below it, because of this glass = (champagneInGlass[j] of ith row-1) / 2.

**Code**

class Solution {

public:

double champagneTower(int poured, int query\_row, int query\_glass) {

vector<double> currRow(1, poured);

for(int i=0; i<=query\_row; i++){ //we need to make the dp matrix only till query row. No need to do after that

vector<double> nextRow(i+2, 0); //If we are at row 0, row 1 will have 2 glasses. So next row will have currRow number + 2 number of glasses.

for(int j=0; j<=i; j++){ //each row will have currRow number + 1 number of glasses.

if(currRow[j]>=1){ //if the champagne from the current glass is being overflowed.

nextRow[j] += (currRow[j]-1)/2.0; //fill the left glass with the overflowing champagne

nextRow[j+1] += (currRow[j]-1)/2.0; //fill the right glass with the overflowing champagne

currRow[j] = 1; //current glass will store only 1 cup of champagne

}

}

if(i!=query\_row) currRow = nextRow; //change the currRow for the next iteration. But if we have already reached the query\_row, then the next iteration will not even take place, so the currRow is the query\_row itself. So don't change as we need the currRow only.

}

return currRow[query\_glass];

}

};

**Refer to chatGPT**

import java.util.ArrayList;

import java.util.List;

class Solution {

public double champagneTower(int poured, int query\_row, int query\_glass) {

List<Double> currRow = new ArrayList<>();

currRow.add((double) poured); // Initial pour into the first glass

for (int i = 0; i <= query\_row; i++) {

List<Double> nextRow = new ArrayList<>();

// Initialize the next row with 0 values

for (int k = 0; k <= i + 1; k++) {

nextRow.add(0.0);

}

// Process each glass in the current row

for (int j = 0; j <= i; j++) {

if (currRow.get(j) >= 1) { // Check for overflow

double overflow = (currRow.get(j) - 1.0) / 2.0;

nextRow.set(j, nextRow.get(j) + overflow); // Left glass

nextRow.set(j + 1, nextRow.get(j + 1) + overflow); // Right glass

currRow.set(j, 1.0); // Current glass capped at 1

}

}

// Move to the next row if not at the query row

if (i != query\_row) {

currRow = nextRow;

}

}

return currRow.get(query\_glass); // Return champagne amount in the desired glass

}

}

**Solution 3: Math + 1D array (180 min)**

class Solution {

    public double champagneTower(int poured, int query\_row, int query\_glass) {

        // Array to store champagne amounts in each row

        // Since 0 <= query\_glass <= query\_row < 100, dp array size as query\_row + 2

        // will make sure contains all potential query\_glass positions

        double[] result = new double[query\_row + 2];

        result[0] = poured; // Initial amount poured into the first glass

        // Iterate through each row

        for (int row = 1; row <= query\_row; row++) {

            // Iterate from the last glass to the first in reverse to prevent overwriting

            for (int col = row - 1; col >= 0; col--) {

                // Calculate overflow for the current glass

                // If there's overflow, distribute it to the glasses below

                double overflow = (result[col] - 1.0 > 0 ? (result[col] - 1.0) / 2.0 : 0);

                if(col < query\_row) {

                    result[col + 1] += overflow;

                }

                result[col] = overflow;

            }

        }

        // Return the champagne in the queried glass (capped at 1.0)

        return Math.min(1.0, result[query\_glass]);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**如何缩减 2D array 到 1D array 呢？**

**注意观察 2D array 的写法中下一行 (i + 1) 中的某一列 (j) 的数值基于正上方 (i, j - 1) 或者左上方 (j) 的数值来计算，如下：**

result[i + 1][j - 1] += (result[i][j - 1] - 1) / 2.0; // (i + 1, j - 1)基于正上方(i, j - 1)的数值来计算

result[i + 1][j] += (result[i][j - 1] - 1) / 2.0; // (i + 1, j)基于左上方(i, j)的数值来计算

**那么在压缩掉 "行" 这个维度后，根据上述细节，计算当前行当前列的数值依赖方向是 for 循环从左往右，那么根据 DP 计算原则，用于计算当前数值的部分必须是固定不变的值，那么如果这些用于下一轮计算的值在用于计算前被更新了，就无法继续使用，所以如果要计算当前行当前列的数值，该列左边的部分是不能变动的，只能把更新的方向改为从右往左，需要 for 循环从右往左**

**最终效果:**

**用单一 1D DP 数组多次从右往左扫描，连续表达 champagne tower 每一行的结果，后一行的结果从右往左覆盖前一行的结果**

下面如图所示，测试的过程使用了 poured = 5, query\_row = 2, query\_glass = 11. 为了保证能覆盖所有的 glass，并且根据 0 <= query\_glass <= query\_row < 100，我们需要初始化一个长度为 query\_row + 1 的一维数组 result

// Array to store champagne amounts in each row

// Since 0 <= query\_glass <= query\_row < 100, dp array size as query\_row + 2

// will make sure contains all potential query\_glass positions

double[] result = new double[query\_row + 2];

初始化第 0 行 (query\_row = 0), 所有 poured = 5 给到唯一一个杯子

// Initial amount poured into the first glass

result[0] = poured;

第 1 行到第 query\_row 行的从右往左迭代赋值

// Iterate through each row

for (int row = 1; row <= query\_row; row++) {

// Iterate from the last glass to the first in reverse to prevent overwriting

for (int col = row - 1; col >= 0; col--) {

// Calculate overflow for the current glass

// If there's overflow, distribute it to the glasses below

double overflow = (result[col] - 1.0 > 0 ? (result[col] - 1.0) / 2.0 : 0);

if(col < query\_row) {

result[col + 1] += overflow;

}

result[col] = overflow;

}

}

用测试数据 poured = 5, query\_row = 2, query\_glass = 11 解释如何迭代赋值

Initial:

result = [5.0, 0.0, 0.0, 0.0]

Round 1: query\_row = 1

The initial glass at [0,0] reserve 1, overflow 5 - 1 = 4 equally split into query\_row = 1's

two glasses [1,0] and [1,1] if in 2D DP array, but since we compress 2D DP array into 1D DP

array, we reuse the 1D DP array and assign value for query\_row = 1's two glasses 1D positions

as [0,0] and [0,1] from right to left as first [0,1] = 4 / 2 = 2.0 then [0,0] = 4 / 2 = 2.0

result = [2.0, 2.0, 0.0, 0.0]

Round 2: query\_row = 2

[2.0, 2.0, 0.0, 0.0]

----> 2.0 at [0,1] reserve 1, overflow 2 - 1 = 1 equally split into query\_row = 2's two glasses

[2,1] and [2,2] if in 2D DP array, but since we compress 2D DP array into 1D DP array, we

reuse the 1D DP array and assign value for query\_row = 2's two glasses 1D positions as

[0,1] and [0,2] from right to left as first [0,2] = 1 / 2 = 0.5 then [0,1] = 1 / 2 = 0.5

2.0 at [0,0] reserve 1, overflow 2 - 1 = 1 equally split into query\_row = 2's two glasses

[2,0] and [2,1] if in 2D DP array, but since we compress 2D DP array into 1D DP array, we

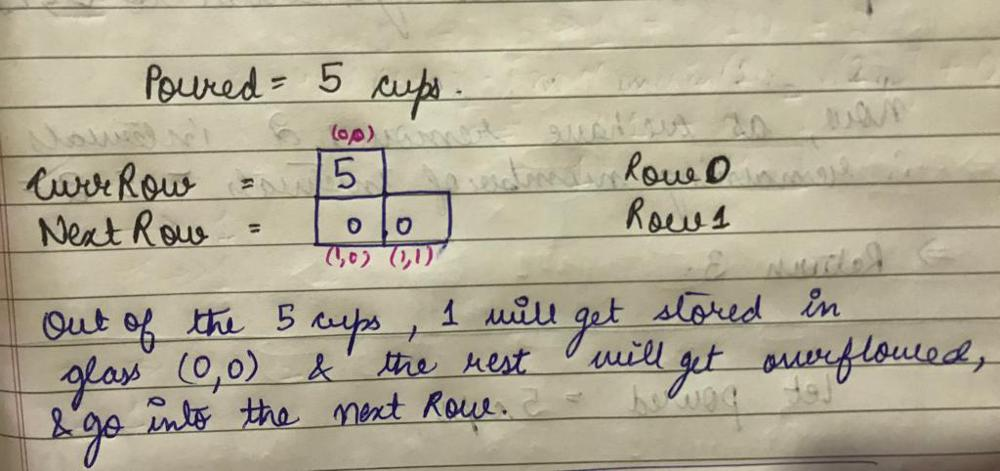
reuse the 1D DP array and assign value for query\_row = 2's two glasses 1D positions as

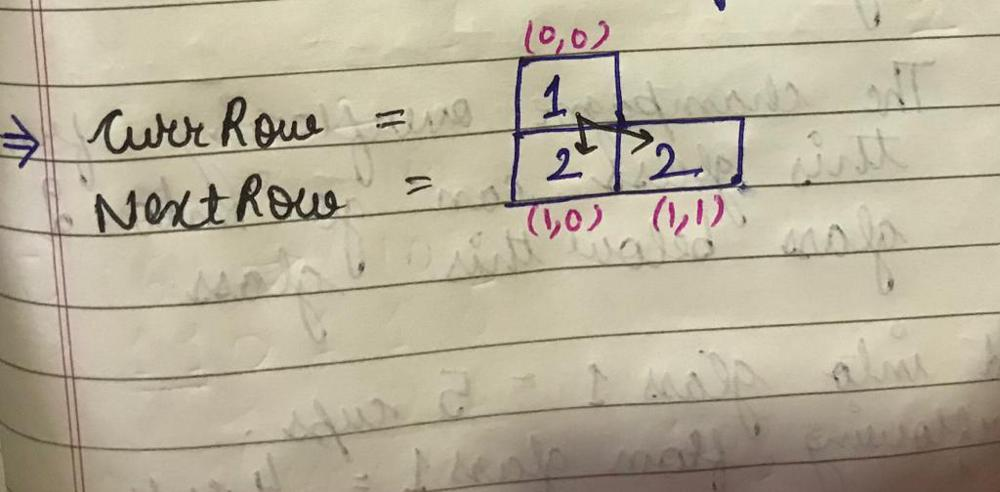
[0,0] and [0,1] from right to left as first [0,1] = 1 / 2 = 0.5 then [0,0] = 1 / 2 = 0.5

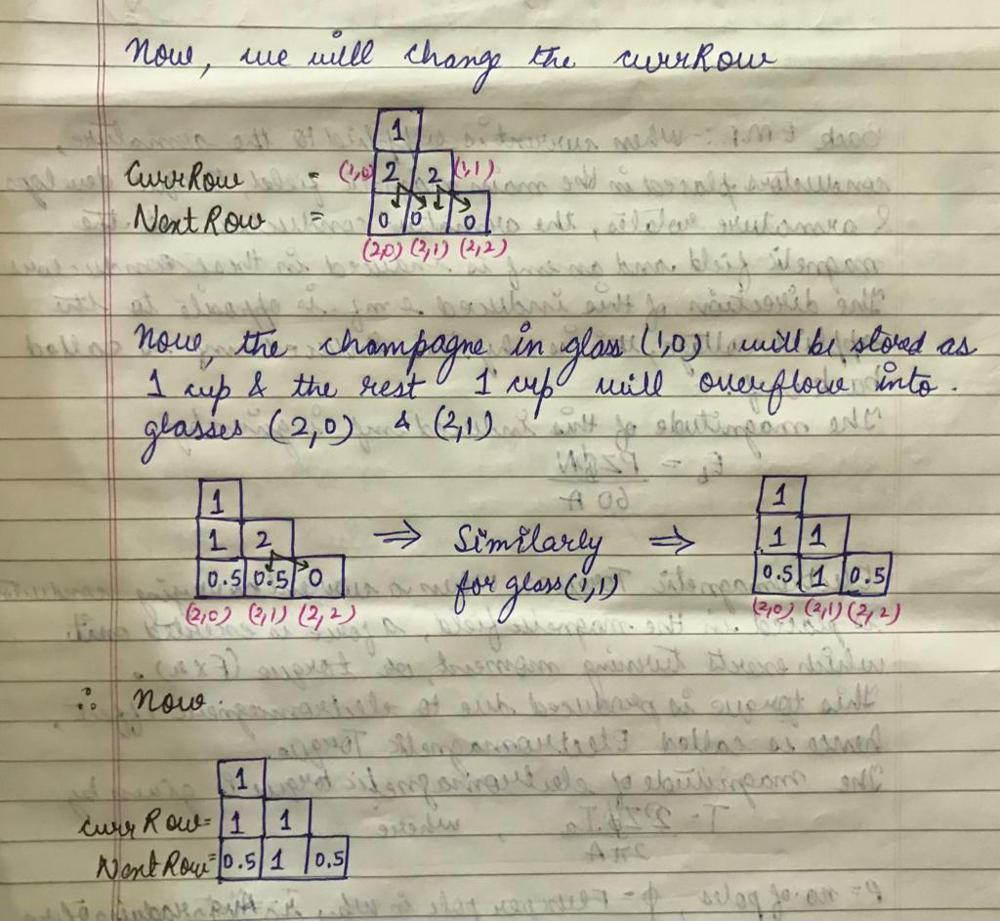
result = [0.5, 0.5 + 0.5, 0.5, 0.0] = [0.5, 1.0, 0.5, 0.0]

**Refer to**

<https://leetcode.com/problems/champagne-tower/solutions/1818599/full-visual-explanation-dp-beginner-friendly-easy-and-simple-c/>







**Test code refer to chatGPT**

public class Solution {

public double champagneTower(int poured, int query\_row, int query\_glass) {

        // Array to store champagne amounts in each row

        // Since 0 <= query\_glass <= query\_row < 100, dp array size as query\_row + 2

        // will make sure contains all potential query\_glass positions

        double[] result = new double[query\_row + 2];

        result[0] = poured; // Initial amount poured into the first glass

        // Iterate through each row

        for (int row = 1; row <= query\_row; row++) {

            // Iterate from the last glass to the first in reverse to prevent overwriting

            for (int col = row - 1; col >= 0; col--) {

                // Calculate overflow for the current glass

                // If there's overflow, distribute it to the glasses below

                double overflow = (result[col] - 1.0 > 0 ? (result[col] - 1.0) / 2.0 : 0);

                if(col < query\_row) {

                    result[col + 1] += overflow;

                }

                result[col] = overflow;

            }

        }

        // Return the champagne in the queried glass (capped at 1.0)

        return Math.min(1.0, result[query\_glass]);

    }

public static void main(String[] args) {

Solution so = new Solution();

int poured = 5;

int query\_row = 2;

int query\_glass = 1;

double result = so.champagneTower(poured, query\_row, query\_glass);

System.out.println(result);

}

}

**If based on Solution 1: Math + Fixed dimension 2D array, then code like below:**

**Refer to**

<https://leetcode.com/problems/champagne-tower/solutions/118660/20ms-c-easy-understand-solution/comments/118203>

double champagneTower(int poured, int query\_row, int query\_glass) {

double dp[101] = {0.0};

dp[0] = poured;

for(int row=1; row<=query\_row; row++)

for(int i=row; i>=0; i--)

dp[i+1] += dp[i] = max(0.0, (dp[i]-1)/2);

return min(dp[query\_glass], 1.0);

}

**Refer to**

[L118.Pascal's Triangle](note://WEB8e837a443a07348b549ff4e44c045c27)