<https://leetcode.com/problems/pascals-triangle/description/>

Given an integer numRows, return the first numRows of **Pascal's triangle**.

In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown:



**Example 1:**

**Input:** numRows = 5

**Output:** [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

**Example 2:**

**Input:** numRows = 1

**Output:** [[1]]

**Constraints:**

1. 1 <= numRows <= 30

**Attempt 1: 2024-11-29**

**Solution 1: Native DFS (30 min)**

class Solution {

    public List<List<Integer>> generate(int numRows) {

        return helper(numRows);

    }

    private List<List<Integer>> helper(int numRows) {

        if(numRows == 0) {

            return new ArrayList<>();

        }

        if(numRows == 1) {

            List<List<Integer>> result = new ArrayList<>();

            result.add(Arrays.asList(1));

            return result;

        }

        List<List<Integer>> prevRows = helper(numRows - 1);

        List<Integer> newRow = new ArrayList<>();

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

            newRow.add(1);

        }

        for(int i = 1; i < numRows - 1; i++) {

            newRow.set(i, prevRows.get(numRows - 2).get(i - 1) + prevRows.get(numRows - 2).get(i));

        }

        prevRows.add(newRow);

        return prevRows;

    }

}

Time Complexity: O(numRows^2)

Space Complexity: O(numRows^2)

**Solution 2: Combinatorial Formula (30 min)**

class Solution {

    public List<List<Integer>> generate(int numRows) {

        List<List<Integer>> result = new ArrayList<>();

        List<Integer> firstRow = new ArrayList<>();

        firstRow.add(1);

        result.add(firstRow);

        for(int i = 1; i < numRows; i++) {

            List<Integer> prevRow = result.get(i - 1);

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

            curRow.add(1);

            for(int j = 1; j < i; j++) {

                curRow.add(prevRow.get(j - 1) + prevRow.get(j));

            }

            curRow.add(1);

            result.add(curRow);

        }

        return result;

    }

}

Time Complexity: O(numRows^2)

Space Complexity: O(numRows^2)

**Refer to**

<https://leetcode.com/problems/pascals-triangle/solutions/4016203/three-approaches-beginner-friendly-full-explanation-c-java-python/>

**Intuition**

The problem of generating Pascal's triangle can be approached in various ways.

Here are some approaches and their intuition to solve the problem:

**Approach 1: Using Recursion**

Intuition: In Pascal's triangle, each element is the sum of the two elements directly above it. We can use a recursive approach to generate the triangle. The base case is when numRows is 1, in which case we return [[1]]. Otherwise, we recursively generate the triangle for numRows - 1 and then calculate the current row by summing the adjacent elements from the previous row.

**Approach 2: Using Combinatorial Formula**

Intuition: Pascal's triangle can also be generated using combinatorial formula C(n, k) = C(n-1, k-1) + C(n-1, k), where C(n, k) represents the binomial coefficient. We can calculate each element of the triangle using this formula. This approach avoids the need for storing the entire triangle in memory, making it memory-efficient.

**Approach 3: Using Dynamic Programming with 1D Array**

Intuition: We can use a dynamic programming approach with a 1D array to generate Pascal's triangle row by row. Instead of maintaining a 2D array, we can use a single array to store the current row and update it as we iterate through the rows. This approach reduces space complexity.

Here's a brief outline of each approach:

Recursion Approach:

Base case: If numRows is 1, return [[1]].

Recursively generate the triangle for numRows - 1.

Calculate the current row by summing adjacent elements from the previous row.

Combinatorial Formula Approach:

Use the binomial coefficient formula C(n, k) to calculate each element.

Build the triangle row by row using the formula.

Dynamic Programming with 1D Array:

Initialize a 1D array to store the current row.

Iterate through numRows and update the array for each row.

Complexity

In terms of time complexity, all three methods have the same overall time complexity of O(numRows^2) because we need to generate all the elements of Pascal's triangle. However, in terms of space complexity, Method 3 is the most efficient as it uses only O(numRows) space, while the other two methods use O(numRows^2) space due to storing the entire triangle or previous rows in recursion.

**Method 1: Using Recursion**

class Solution {

public List<List<Integer>> generate(int numRows) {

if (numRows == 0) return new ArrayList<>();

if (numRows == 1) {

List<List<Integer>> result = new ArrayList<>();

result.add(Arrays.asList(1));

return result;

}

List<List<Integer>> prevRows = generate(numRows - 1);

List<Integer> newRow = new ArrayList<>();

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

newRow.add(1);

}

for (int i = 1; i < numRows - 1; i++) {

newRow.set(i, prevRows.get(numRows - 2).get(i - 1) + prevRows.get(numRows - 2).get(i));

}

prevRows.add(newRow);

return prevRows;

}

}

**Method 2: Using Combinatorial Formula**

class Solution {

public List<List<Integer>> generate(int numRows) {

List<List<Integer>> result = new ArrayList<>();

if (numRows == 0) {

return result;

}

List<Integer> firstRow = new ArrayList<>();

firstRow.add(1);

result.add(firstRow);

for (int i = 1; i < numRows; i++) {

List<Integer> prevRow = result.get(i - 1);

List<Integer> currentRow = new ArrayList<>();

currentRow.add(1);

for (int j = 1; j < i; j++) {

currentRow.add(prevRow.get(j - 1) + prevRow.get(j));

}

currentRow.add(1);

result.add(currentRow);

}

return result;

}

}

**Refer to**

<https://algo.monster/liteproblems/118>

class Solution {

public List<List<Integer>> generate(int numRows) {

// Initialize the main list that will hold all rows of Pascal's Triangle

List<List<Integer>> triangle = new ArrayList<>();

// The first row of Pascal's Triangle is always [1]

triangle.add(List.of(1));

// Loop through each row (starting from the second row)

for (int rowIndex = 1; rowIndex < numRows; ++rowIndex) {

// Initialize the list to hold the current row's values

List<Integer> row = new ArrayList<>();

// The first element in each row is always 1

row.add(1);

// Compute the values within the row (excluding the first and last element)

for (int j = 0; j < triangle.get(rowIndex - 1).size() - 1; ++j) {

// Calculate each element as the sum of the two elements above it

row.add(triangle.get(rowIndex - 1).get(j) + triangle.get(rowIndex - 1).get(j + 1));

}

// The last element in each row is always 1

row.add(1);

// Add the computed row to the triangle list

triangle.add(row);

}

// Return the fully constructed list of rows of Pascal's Triangle

return triangle;

}

}

**Time and Space Complexity**

The provided code generates Pascal's Triangle with numRows levels. Here is the analysis of its time and space complexity:

**Time Complexity**

The time complexity of the code can be understood by analyzing the operations inside the for-loop that generates each row of Pascal's Triangle.

There are numRows - 1 iterations since the first row is initialized before the loop.

Inside the loop, pairwise(f[-1]) generates tuples of adjacent elements from the last row which takes O(j) time, where j is the number of elements in the previous row (since every step inside the enumeration would be constant time).

The list comprehension [a + b for a, b in pairwise(f[-1])] performs j - 1 additions, so this is also O(j) where j is the size of the previous row.

Appending the first and last 1 is O(1) each for a total of O(2) which simplifies to O(1).

As the rows of Pascal's Triangle increase by one element each time, summing up the operations over all rows gives us a total time of approximately O(1) for the first row, plus O(2) + O(3) + ... + O(numRows) for the numRows - 1 remaining rows. This results in a time complexity of O(numRows^2).

Therefore, the overall time complexity is O(numRows^2).

**Space Complexity**

The space complexity is determined by the space required to store the generated rows of Pascal's Triangle.

f is initialized with a single row containing one 1 which we can consider O(1).

In each iteration of the loop, a new list with i + 2 elements is created (since each new row has one more element than the previous one) and appended to f.

Summing this up, the space required will be O(1) for the first row, plus O(2) + O(3) + ... + O(numRows) for the rest of the rows.

This results in a space complexity of O(numRows^2) since the space required is proportional to the square of the number of rows due to the nature of Pascal's Triangle.

Therefore, the overall space complexity is O(numRows^2).