118. Pascal's Triangle

Easy Array Dynamic Programming Leetcode Link

Problem Description

Given an integer `numRows`, the task is to return the first `numRows` of Pascal's triangle. In Pascal's triangle, each number is the sum of the two numbers directly above it except for the boundaries, which are always `1`. The triangle starts with a `1` at the top. Then, each subsequent row contains one more number than the previous row, and these numbers are positioned such that they form a triangle. The challenge here is to compute these numbers using the given property of Pascal's triangle and to represent the triangle in the form of a list of lists, where each inner list corresponds to a row in the triangle.

For example, the first 3 rows of Pascal's triangle look like this:

```
1               //First row (only one element)
1 1             //Second row (two 1s at the boundaries)
1 2 1           //Third row (1s at the boundaries, middle is sum of the two numbers above)
```

Intuition

The solution to generating Pascal's triangle is a direct application of its properties. The key idea is to start with the first row `[1]`, and then generate each following row based on the previous row. For any new row, aside from the first and last elements which are `1`, each element is obtained by adding the two numbers directly above it in the triangle. With this approach, we can iteratively construct each row and add it to the final result.

This can be implemented by initializing a list with the first row as its only element, then using a loop to construct each subsequent row. Inside the loop, we can use list comprehension to generate the middle values of the row by summing pairs of consecutive elements from the last row. We use the `pairwise` function (assuming Python 3.10+) to obtain the pairs, and then enclose the generated middle values with `[1]` on both ends to complete the row. After constructing each row, we append it to our result list. We repeat this process `numRows - 1` times since the first row is already initialized.

The Python code encapsulates this logic in the `generate` function, with `f` holding the result list of lists and `g` being the new row being constructed each iteration. Finally, the function returns `f` after the completion of the loop.

Solution Approach

In the provided code, the problem is approached by considering each row in Pascal's Triangle to be a list and the entire triangle to be a list of these lists.

The main steps of the implementation are as follows:

1. Initialize a list, `f`, with a single element that is the first row of Pascal's Triangle, which is simply `[1]`.
2. Loop from `0` to `numRows - 1` (the `-1` is because we already have the first row). This loop will build one row at a time to add to our `f` list.
3. For each iteration, construct a new row, `g`. The first element is always `1`. This is done by simply specifying `[1]` at the start.
4. The middle elements of the row `g` are created by taking pairs of consecutive elements from the last row available in `f` and summing them up. This is done using list comprehension and the `pairwise` function. `pairwise` takes an iterable and returns an iterator over pairs of consecutive elements. For example, `pairwise([1,2,3])` would produce `(1, 2), (2, 3)`.

   In the code, we have `[a + b for a, b in pairwise(f[-1])]`, which sums each pair of consecutive elements from the last row `f[-1]` to create the middle elements of the new row `g`.
5. After the middle elements are calculated, `[1]` is appended to the end of the list to create the last element of the row, which is also `1`.
6. Row `g` is then added to the list `f`.
7. This process repeats until all `numRows` rows are generated.
8. Finally, the list `f` is returned, which contains the representation of Pascal's Triangle up to `numRows`.

This solution uses nested lists to represent Pascal's Triangle, list comprehension to generate each row, and a loop to build the triangle one row at a time. The `pairwise` function streamlines the process of summing adjacent elements from the previous row to build the interior of each new row.

Here is the critical part of the Python solution:

```python
f = [[1]]  # Initial Pascal's Triangle with the first row.
for i in range(numRows - 1):
    g = [1] + [a + b for a, b in pairwise(f[-1])] + [1]  # Constructing the new row
    f.append(g)  # Appending the new row to the triangle
return f  # The complete Pascal's Triangle
```

The algorithm has a time complexity of O(n^2), where n is the number of rows, since each element of the triangle is computed exactly once.

Example Walkthrough

Let's walk through the solution with `numRows = 5` to illustrate the generation of Pascal's Triangle:

1. We start with the initial list `f` which already contains the first row `[1]`.
2. We begin the loop with `i` ranging from `0` to `numRows - 1`, which is `0` to `4` in this case. Remember, the first row is already accounted for, so we effectively need to generate `4` more rows.
3. For `i = 0`, we construct the second row:

   `g = [1] + [a + b for a, b in pairwise(f[-1])] + [1]`

   - `f[-1]` is `[1]`, so `pairwise(f[-1])` doesn't generate any pairs since there's only one element.
   - We add `[1]` at the beginning and end.

   So, `g = [1] + [] + [1]` which is simply `[1, 1]`. We append `[1, 1]` to `f`.
4. Our list `f` now looks like `[[1], [1, 1]]`.
5. For `i = 1`, we construct the third row:

   `f[-1]` is `[1, 1]`, `pairwise(f[-1])` yields one pair: `(1, 1)`.

   - We then add the sums of these pairs to `[1]` and enclose with `[1]` at the end.

   So, `g = [1] + [1 + 1] + [1]` which gives us `[1, 2, 1]`. We append `[1, 2, 1]` to `f`.
6. Our list `f` now looks like `[[1], [1, 1], [1, 2, 1]]`.
7. For `i = 2`, constructing the fourth row:

   `f[-1]` is `[1, 2, 1]`, `pairwise(f[-1])` gives `(1, 2)` and `(2, 1)`.

   - The sums are `[1 + 2, 2 + 1]` which is `[3, 3]`.

   So, `g = [1] + [3, 3] + [1]` which forms `[1, 3, 3, 1]`. We append `[1, 3, 3, 1]` to `f`.
8. Our list `f` now looks like `[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1]]`.
9. For `i = 3`, the final loop to construct the fifth row:

   `f[-1]` is `[1, 3, 3, 1]`, `pairwise(f[-1])` yields `(1, 3)`, `(3, 3)`, and `(3, 1)`.

   - The sums are `[1 + 3, 3 + 3, 3 + 1]` which is `[4, 6, 4]`.

   `g = [1] + [4, 6, 4] + [1]`, forming the row `[1, 4, 6, 4, 1]`. We append this row to `f`.
10. Our final list `f`, representing the first `5` rows of Pascal's Triangle, now looks like `[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]`.

The function would then return this list `f`.

Executing these steps with a proper function such as the given `generate` function will yield the desired Pascal's Triangle up to the specified row. Each iteration builds upon the previous, aligning perfectly with the properties of Pascal's Triangle.

Python Solution

```python
class Solution:
    def generate(self, numRows: int) -> List[List[int]]:
        # Initialize the first row of Pascal's Triangle
        pascal_triangle = [[1]]

        # Generate each row of Pascal's Triangle
        for i in range(1, numRows):
            # The first element of each row is always 1
            new_row = [1]

            # Calculate the intermediate elements of the row
            # by adding the pairs of adjacent elements from the previous row
            for j in range(1, i):
                sum_of_elements = pascal_triangle[i - 1][j - 1] + pascal_triangle[i - 1][j]
                new_row.append(sum_of_elements)

            # The last element of each row is also always 1
            new_row.append(1)

            # Append the newly generated row to Pascal's Triangle
            pascal_triangle.append(new_row)

        # Return the completed Pascal's Triangle
        return pascal_triangle

```

Java Solution

```java
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;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to generate Pascal's Triangle with the given number of rows.
    vector<vector<int>> generate(int numRows) {
        // Initialize a 2D vector to hold the rows of Pascal's Triangle.
        vector<vector<int>> pascalsTriangle;

        // The first row of Pascal's Triangle is always [1].
        pascalsTriangle.push_back(vector<int>(1, 1));

        // Generate the subsequent rows of Pascal's Triangle.
        for (int rowIndex = 1; rowIndex < numRows; ++rowIndex) {
            // Initialize the new row starting with a '1'.
            vector<int> newRow;
            newRow.push_back(1);

            // Fill in the values between the first and last '1' of the row.
            for (int elementIndex = 0; elementIndex < pascalsTriangle[rowIndex - 1].size() - 1; ++elementIndex) {
                // The new value is the sum of the two values directly above it.
                newRow.push_back(pascalsTriangle[rowIndex - 1][elementIndex] + pascalsTriangle[rowIndex - 1][elementIndex + 1]);
            }

            // The last value in a row of Pascal's Triangle is always '1'.
            newRow.push_back(1);

            // Append the newly created row to Pascal's Triangle.
            pascalsTriangle.push_back(newRow);
        }

        // Return the fully generated Pascal's Triangle.
        return pascalsTriangle;
    }
};

```

Typescript Solution

```typescript
// Generates Pascal's Triangle with a given number of rows
function generate(numRows: number): number[][] {
    // Initialize the triangle with the first row
    const triangle: number[][] = [[1]];

    // Populate the triangle row by row
    for (let rowIndex = 1; rowIndex < numRows; ++rowIndex) {
        // Initialize the new row starting with '1'
        const row: number[] = [1];

        // Calculate each value for the current row based on the previous row
        for (let j = 0; j < triangle[rowIndex - 1].length - 1; ++j) {
            row.push(triangle[rowIndex - 1][j] + triangle[rowIndex - 1][j + 1]);
        }

        // End the current row with '1'
        row.push(1);

        // Add the completed row to the triangle
        triangle.push(row);
    }

    // Return the fully generated 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)`.