119. Pascal's Triangle II

Easy | Array | Dynamic Programming

Leetcode Link

Problem Description

The task is to generate a specific row from Pascal's triangle, which is a triangular array of the binomial coefficients. Pascal's triangle is constructed such that each number in the triangle is the sum of the two numbers directly above it in the previous row. The rows of Pascal's triangle are conventionally enumerated starting with row `0` at the top (`1`), with each row getting progressively longer. The problem requires us not to construct the entire triangle but to return only the row at the specified index (`rowIndex`), which is zero-indexed.

Intuition

To arrive at the solution approach without constructing the entire Pascal's triangle, we exploit the fact that each element in a row can be calculated based on the previous row's elements. For the algorithm provided, we initialize a list, `f`, to contain `1`s with a length of `rowIndex + 1`, representing the number of elements in the target row. The first and last elements in any row of Pascal's triangle are always `1`.

Then we iterate through the list starting from the third row (since the first two rows are always `[1]` and `[1, 1]`), and we move backwards inside each row to calculate the new value for each position based on the previous value and the value before it (`f[j] += f[j - 1]`). This in-place update allows us to build the current row directly from the values of the previous row without extra space for storing the entire triangle.

The thinking process is mainly about optimizing space since we only need the final row, not the entire triangle, and building upon the knowledge of the relationship between the elements in Pascal's triangle.

Solution Approach

The implementation of the solution follows a bottom-up approach in calculating the values of a specific row in the Pascal's triangle. The code makes use of an array to dynamically update and calculate the row's values in-place. Here's a breakdown of the implementation:

1. We initialize a list `f` with `rowIndex + 1` elements, all set to `1`. This represents our row in Pascal's triangle, where the first and last elements are always `1`, and there are `rowIndex + 1` elements in the `rowIndex`'th row.
2. We then iterate from the third row (index `2`) up to `rowIndex` inclusive, because the first two rows are always `[1]` and `[1, 1]` and do not change.
3. For each row `i`, we update the values starting from the end (at position `i - 1`) down to `1`. We do this backward so that each value uses the original value from the preceding step. We update each element `f[j]` by adding to it the value before it, `f[j - 1]` (`f[j] += f[j - 1]`). This effectively calculates the current element based on the two elements above it in the Pascal's triangle.
   - For example, in row `4`, to calculate the value at index `2`, we add the value at `f[1]` and `f[2]` from row `3`. Since we move backward, the value at `f[2]` is still the original one from row `3`.
4. We follow this update process for all rows until we reach the desired `rowIndex`. By then, list `f` contains the correct values for that particular row in the Pascal's triangle.
5. Finally, we return the list `f` as the result, which now contains the values of the `rowIndex`'th row of the Pascal's triangle.

This algorithm is efficient because it uses only a single list, hence the space complexity is O(n), where `n` is the `rowIndex`. The time complexity is also O(n^2), since we must compute the values for each element in the list, and the number of calculations per row grows linearly with the row index.

Example Walkthrough

Let's apply the solution approach to calculate the 4th row (zero-indexed, so it is actually the 5th row in human terms) of Pascal's triangle. Recall that the 4th row index corresponds to `[1, 4, 6, 4, 1]` in Pascal's triangle.

We initialize `f` to `[1, 1, 1, 1, 1]` to represent the 4th row with its length being `rowIndex + 1`, which is `5` in this case.

Now, we iteratively update the list starting from the third row:

- For the third row (index 2), we leave the first and last elements as `1` and update the middle one. We start from the end going backward, so `f[1]` (which is the second element in the array) is updated by adding the value of the first element (`f[0]`):
  - `f[1]` was `1`, and `f[0]` is also `1`, so `f[1]` becomes `1 + 1 = 2`.
- Now the list looks like `[1, 2, 1, 1, 1]`.

Next, we move to the fourth row (index 3):

- Starting from the end again, we first update `f[2]`, which depends on `f[1]` and `f[2]` from the previous step:
  - Initially, both `f[1]` and `f[2]` are `2`, so `f[2]` becomes `2 + 2 = 4`.
- Then we update `f[1]`, which will be the sum of the original `f[0]` and `f[1]` from the prior step:
  - We already know that `f[0]` is `1` and the updated `f[1]` is `2`, so `f[1]` becomes `1 + 2 = 3`.
- Now our list is `[1, 3, 4, 1, 1]`.

Finally, for the row we're interested in (the 4th row, index 4):

- We update `f[3]` similarly, which now becomes the sum of the previous `f[2]` and `f[3]`:
  - With `f[2]` being `4` and `f[3]` being `1`, `f[3]` is updated to `4 + 1 = 5`.
- And then `f[2]` gets updated, this time using the new value of `f[3]` and the previous value of `f[2]`:
  - Now `f[2]` is updated to `3 + 4 = 7`.
- The `f[1]` is updated to be the sum of `f[1]` and `f[2]` from the previous set:
  - We update `f[1]` to `3 + 3 = 6`.
- The list now represents the 4th row and is `[1, 4, 6, 4, 1]`.

We return `[1, 4, 6, 4, 1]` as the solution to the problem for the 4th row index. Each step uses the values from the previous step, and we only need to store one row at a time, ensuring that the space complexity is linear with respect to the row index given.

Python Solution

```python
class Solution:
    def getRow(self, rowIndex: int) -> List[int]:
        row = [1] * (rowIndex + 1)  # Initialize the row with 1s
        # Start from the third row, since the first two rows are trivially [1] and [1, 1]
        for i in range(2, rowIndex + 1):
            # Move backwards to update the current row in place
            # This is required to ensure we use the values from the previous iteration
            # Starting at the second to last item of the row and ending at the second item
            # since the first and the last items of the row are always 1
            for j in range(i - 1, 0, -1):
                # Update the value of the current cell by adding the value of the cell
                # to the left of the current cell, which effectively calculates the binomial coefficient
                row[j] += row[j - 1]
        return row

```

Java Solution

```java
import java.util.ArrayList;
import java.util.List;

class Solution {
    // Function to return the specified row of Pascal's triangle.
    public List<Integer> getRow(int rowIndex) {
        // Initialize the list to hold the row values
        List<Integer> row = new ArrayList<>();

        // Create the first row of Pascal's triangle with all elements initialized to 1
        for (int i = 0; i <= rowIndex; ++i) {
            row.add(1);
        }

        // Generate the values for the rest of the Pascal's triangle row,
        // starting from the third row (since the first two rows are all 1s)
        for (int i = 2; i <= rowIndex; ++i) {
            // Move backwards to modify the row in place,
            // calculating the sum of two elements from the previous row
            for (int j = i - 1; j > 0; --j) {
                // Set the current position with the sum of the two elements above it
                row.set(j, row.get(j) + row.get(j - 1));
            }
        }

        // Return the completed row of Pascal's triangle
        return row;
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to calculate and return the rowIndex-th row of Pascal's Triangle
    std::vector<int> getRow(int rowIndex) {
        // Initialize a vector with size of rowIndex+1 elements, all set to 1
        // This will hold the current state of the row being calculated
        std::vector<int> row(rowIndex + 1, 1);

        // Start iterating from the third row, because the first two rows are [1] and [1, 1]
        for (int i = 2; i < rowIndex + 1; ++i) {
            // Update the row values from the end to the beginning
            // This is done in reverse to avoid overwriting values needed for calculation.
            // Example: for the third iteration the row is [1, 1, 1], and this loop updates it to [1, 2, 1]
            for (int j = i - 1; j > 0; --j) {
                // Current element is the sum of the element above it and the element to the left of the above element
                row[j] += row[j - 1];
            }
        }

        // Return the fully computed row
        return row;
    }
};

```

Typescript Solution

```typescript
function getRow(rowIndex: number): number[] {
    // Initialize an array to hold the coefficients with all elements as 1
    // which represents the first and last element of each row being 1
    const rowCoefficients: number[] = Array(rowIndex + 1).fill(1);

    // Start from the third row since the first two rows are always [1] and [1, 1]
    for (let i = 2; i < rowIndex + 1; ++i) {
        // Update coefficients from the end to avoid overwriting elements
        // that are yet to be updated
        for (let j = i - 1; j > 0; --j) {
            // Pascal's triangle property: each number is the sum of the two numbers above it
            rowCoefficients[j] += rowCoefficients[j - 1];
        }
    }

    // Return the coefficients of the desired row
    return rowCoefficients;
}

```

Time and Space Complexity

The time complexity of the given code is `O(n^2)` where `n` is the `rowIndex`. This quadratic time complexity arises because there are two nested loops: the outer loop runs from 2 to `rowIndex` (exclusive), and for each iteration of the outer loop, the inner loop runs backwards from `i - 1` to 1, where `i` is the current iteration of the outer loop. Since `i` increases linearly, the total amount of work done is the sum of the first `n` integers, which leads to an `O(n^2)` time complexity.

The space complexity of the code is `O(n)` where `n` is the `rowIndex`. This space complexity is due to the list `f` which contains `rowIndex + 1` elements. The space taken by the list is directly proportional to the `rowIndex`, hence the space complexity is linear.