Pascal's Triangle

Pascal's Triangle is a triangular array of numbers where each number is the sum of the two numbers directly above it. The edges of the triangle are always 1.

The triangle starts with a 1 at the top, then each row is constructed by adding the numbers above, treating blank spaces as zeros.

For example:

```
     1
    1 1
   1 2 1
  1 3 3 1
 1 4 6 4 1
1 5 10 10 5 1
```

Problem Types

In an interview, you might encounter three types of problems related to Pascal's Triangle:

- Given a row and column, find the element at that position.
- Given `n`, print the `n`th row of Pascal's Triangle.
- Given `n`, print the entire Pascal's Triangle up to the `n`th row.

Find Element by Row and Column

Given row `r` and column `c`, find the element at that position. For example, if `r = 5` and `c = 3`, the element is 6.

One way is to generate the entire triangle. But, there's an easier way.

Use the formula: ${}^{(r-1)}C_{(c-1)}$

> ${}^{n}C_{r}\ also written as \$NCR\$$ represents the number of combinations of n items taken r at a time. The formula to calculate it is:
>
> ${}^{n}C_{r} = \frac{n!}{r! \cdot (n-r)!}$

where $n!$ is the factorial of n, meaning the product of all positive integers up to n.

So, to find the element at row 5, column 3:

$^{(5-1)}C_{(3-1)} = ^4C_2 = \frac{4!}{2! \cdot (4-2)!} = \frac{4 \cdot 3 \cdot 2 \cdot 1}{(2 \cdot 1) \cdot (2 \cdot 1)} = 6$

Calculating ^nC_r

The most straightforward method is to calculate the factorials separately using loops.

```
def factorial(n):
    fact = 1
    for i in range(1, n + 1):
        fact *= i
    return fact

def nCr(n, r):
    numerator = factorial(n)
    denominator = factorial(r) * factorial(n - r)
    return numerator // denominator
```

However, there's a shortcut to avoid large factorials:

$^7C_2 = \frac{7!}{2! \cdot 5!} = \frac{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}{(2 \cdot 1) \cdot (5 \cdot 4 \cdot 3 \cdot 2 \cdot 1)}$

Notice that $5!$ cancels out. This simplifies to:

$^7C_2 = \frac{7 \cdot 6}{2 \cdot 1}$

From 7, go two places down $because we have 2 in \$C_2\$$.

General Rule: In ^nC_r, both the numerator and denominator will have r terms.

Example:

$^{10}C_3 = \frac{10 \cdot 9 \cdot 8}{3 \cdot 2 \cdot 1}$

Important: When computing this, perform divisions gradually to avoid potential overflow.

Compute in this sense: $\frac{10}{1} \cdot \frac{9}{2} \cdot \frac{8}{3}$

Code Implementation

```python
def nCr(n, r):
    result = 1
    for i in range(r):
        result = result * (n - i) // (i + 1)
    return result
```

In this code:

- We initialize `result` to 1.
- We loop `r` times.
- In each iteration, we multiply `result` by `n - i` and divide by `i + 1`.

Time and Space Complexity

- Time Complexity: $O(r)$ - because we loop `r` times.
- Space Complexity: $O(1)$ - constant space.

Complete Solution

```python
def get_element_pascal(r, c):
    # Adjust for 0-based indexing
    r -= 1
    c -= 1
    return nCr(r, c)
```

Important Note

For such problems, it's safer to use the largest data type available (e.g., `long long` in C++) to prevent overflow.

Printing a Specific Row

Let's explore how to print a specific row from Pascal's Triangle.

Brute Force Approach

- The *n*th row contains *n* elements.
- Each element can be computed using the formula *nCr* $or\$R-1CC-1\$inthiscontext.$
- We can iterate through the columns (from 1 to *n*) and compute/print each element using the *nCr* formula.

```
for each column from 1 to n:
    print NCR(n-1, c-1)
```

- **Time Complexity:** $On*R$, where *n* is the row number and *R* is related to the computation of *nCr*. This is because we loop *n* times and the *nCr* function has a complexity of OR.

Optimization

To optimize, we need to dive deeper into the *nCr* formula and identify patterns.

- Observe how each element relates to the previous one in the row.

Let's look at an example. To get the elements in row 6, we see the following pattern:

- First element: 1
- Second element: 5 $canbewrittenas5/1$
- Third element: 10 $canbewrittenas(5*4/1*2)$
- Fourth element: 10 $canbewrittenas(5*4*3/1*2*3)$

It seems as though each subsequent element is derived from the previous by multiplying and dividing by certain values.

- Start with `answer = 1`.
- In the second column, add a factor of `5/1`.
- In the third column, add a factor of `4/2`.
- In the fourth column, add a factor of `3/3`.

Formula

Using zero-based indexing to simplify the formula:

- Multiply by `(row - column)`
- Divide by `column`

```
answer = 1
for i from 1 to n:
    answer = answer * (n - i)
    answer = answer / i
    print answer
```

- Time Complexity: $O(n)$, where n is the number of rows, as we iterate through each column once.
- Space Complexity: $O(1)$, as we use a constant amount of extra space.

Printing the Entire Pascal's Triangle

Now, let's tackle the problem of printing the entire Pascal's Triangle up to a given row n.

Brute Force Approach

```
The brute force approach involves using the formula R-1 C C-1 to generate eac
```

- The rows will range from 1 to n.
- The columns will range from 1 to the current row number.
- Create a temporary list to store the elements of the current row.
- Add this temporary list to the answer list.

```
answer = []
for row from 1 to n:
    temp_list = []
    for col from 1 to row:
        temp_list.add(NCR(row-1, col-1))
    answer.add(temp_list)
return answer
```

- Time Complexity: $O(n^3)$, which comes from the nested loops $one for rows$, $one for columns$ and the nCr calculation. This is approximate.
- Space Complexity: Not discussed, as it's primarily for storing the answer to be returned.

We will need to use dynamic programming or other optimization techniques to calculate the value of NCR.

Optimal Solution for Type 3 Pascal's Triangle

The optimal solution for Type 3 Pascal's Triangle generation involves creating each row individually and appending it to the result. For `n` rows, this approach takes `O(n)` time for each row, leading to a total complexity of `O(n^2)`.

Generate Row Function

The `generateRow` function takes the row number as input and returns a list containing all the elements in that row.

Here's a step-by-step breakdown:

1. Initialize the **answer** list, starting with `1`.
2. Use **zero-based indexing**, where the first row is considered row `0`.
3. Iterate through each column from `1` up to the row number.
4. Update the answer using the formula:

$$answer = answer * (row - column + 1)/column$$

Here's a Python example of the `generateRow` function:

```python
def generateRow(row_number):
    answer = [1]
    for column in range(1, row_number + 1):
        answer.append(answer[-1] * (row_number - column + 1) // column)
    return answer
```

Generate Triangle Function

The `generateTriangle` function uses the `generateRow` function to create each row of the triangle.

1. Initialize an empty list of lists called `answer`.
2. Iterate through each row number from `0` to `n - 1`.
3. Call the `generateRow` function for each row number and append the resulting list to the `answer`.
4. Return the `answer`.

Here's a Python example of the `generateTriangle` function:

```python
def generateTriangle(num_rows):
    answer = []
    for i in range(num_rows):
        answer.append(generateRow(i))
    return answer
```

Code Quality Matters

Maintaining code quality is vital, especially during interviews. Here are some points to consider:

- Readability: Ensure your code is easy to understand.
- Modularity: Break down complex tasks into smaller, manageable functions.
- Function Usage: Use functions to encapsulate specific tasks, making the code more organized.

  Code quality is important because interviewers will focus on your code quality in addition to the correctness of your solution.

Instead of writing lengthy, unorganized code, structure it into logical blocks. For example, instead of copy-pasting the row generation logic directly into the main function, encapsulate it within the `generateRow` function.

Good Code Example:

```python
def generateRow(row_number):
    answer_row = [1]
    for column in range(1, row_number + 1):
        answer_row.append(answer_row[-1] * (row_number - column + 1) // co
    return answer_row

def generateTriangle(num_rows):
    answer = []
    for i in range(num_rows):
        answer.append(generateRow(i))
    return answer
```

Bad Code Example:

```python
def generateTriangle(num_rows):
    answer = []
    for i in range(num_rows):
        answer_row = [1]
        for column in range(1, i + 1):
            answer_row.append(answer_row[-1] * (i - column + 1) // column)
        answer.append(answer_row)
    return answer
```

Key Takeaways

- Optimal solution for Type 3 Pascal's Triangle has a time complexity of `O(n^2)`.
- Break down problems into smaller, modular functions for better code quality.
- Focus on code readability to impress interviewers.