Previous Next

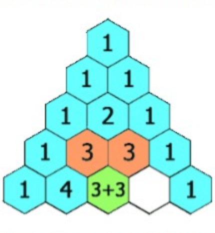
119. Pascal's Triangle II

Feb. 10, 2020 | 12K views

Average Rating: 4.42 (12 votes)

Given a non-negative index k where $k \leq 33$, return the k^{th} index row of the Pascal's triangle.

Note that the row index starts from 0.



In Pascal's triangle, each number is the sum of the two numbers directly above it.

Example:

```
Input: 3
Output: [1,3,3,1]
```

Follow up:

Could you optimize your algorithm to use only $O(k)$ extra space?

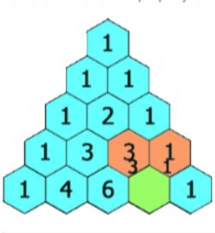
Solution

> If you haven't attempted 118. Pascal's Triangle, I would strongly recommend that you try that first.

Approach 1: Brute Force Recursion

Intuition

We'll utilize a nice little property of Pascal's Triangle (given in the problem description):



> In Pascal's triangle, each number is the sum of the two numbers directly above it.

Approach 4 will expand more on why it is so.

Algorithm

Let's say we had a function `getNum(rowIndex, colIndex)`, which gave us the `colIndex` th number in the `rowIndex` th row, we could simply build the k^{th} row by repeatedly calling `getNum(...)` for columns 0 to k.

We can formulate our intuition into the following recursion:

$$\text{getNum}(\text{rowIndex}, \text{colIndex}) = \text{getNum}(\text{rowIndex-1}, \text{colIndex-1}) + \text{getNum}(\text{rowIndex-1}, \text{colIndex})$$

The recursion ends in some known base cases: 1. The first row is just a single 1, i.e. $\text{getNum}(0, ...) = 1$

1. The first and last number of each row is 1, i.e. $\text{getNum}(\text{k}, 0) = \text{getNum}(\text{k}, \text{k}) = 1$

C++ Java Copy

```cpp
class Solution {
public:
  int getNum(int row, int col) {
    if (row == 0 || col == 0 || row == col)
      return 1;

    return getNum(row - 1, col - 1) + getNum(row - 1, col);
  }

  vector<int> getRow(int rowIndex) {
    vector<int> ans;

    for (int i = 0; i <= rowIndex; i++)
      ans.push_back(getNum(rowIndex, i));

    return ans;
  }
};
```

Complexity Analysis

- Time complexity : $O(2^k)$. The time complexity recurrence is straightforward:

$$T(k, i) = T(k-1, i) + T(k-1, i-1) + O(1) \quad \ni \quad T(k, k) = T(k, 0) = O(1)$$

  Thus, $\text{T}(\text{k}, \text{m})$ takes $\binom{k}{m}$ units of constant time. [1]

  For the k^{th} row, total time required is:

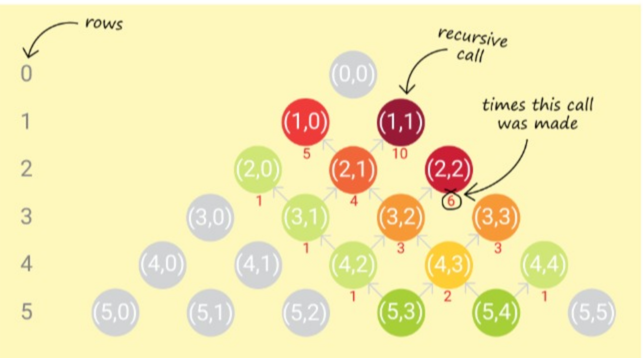
$$\begin{aligned} T(k,0) + T(k,1) + \ldots + T(k,k-1) + T(k,k) &= \sum_{m=0}^{k} T(k,m) \\ &\simeq \sum_{m=0}^{k} O(\binom{k}{m}) \\ &\simeq O(\sum_{m=0}^{k} \binom{k}{m}) \\ &= O(2^k) \end{aligned}$$

- Space complexity : $O(k) + O(k) \simeq O(k)$.
  - We need $O(k)$ space to store the output of the k^{th} row.
  - At worst, the recursive call stack has a maximum of k calls in memory, each call taking constant space. That's $O(k)$ worst case recursive call stack space.

Approach 2: Dynamic Programming

Intuition

In the previous approach, we end up making the same recursive calls repeatedly.



For example, to calculate `getNum(5, 3)` and `getNum(5, 4)`, we end up calling `getNum(3, 2)` thrice. To generate, the entire fifth row (`0`-based row indexing), we'd have to call `getNum(3, 2)` four times.

It makes sense to store the results of intermediate recursive calls for later use.

Algorithm

Simple memoization caches results of deep recursive calls and provides significant savings on runtime.

C++ Java Copy

```cpp
class Solution {
  unordered_map<size_t, int> cache;

  // use a better hashing function like `boost::hash_combine` in the real world.
  int key(int i, int j) const {
    size_t hash_i = hash<int>{}(i), hash_j = hash<int>{}(j);
    int hashed = (int)(hash_i ^ (hash_i >> 32));
    return (hashed << 5) - 1 + (int)(hash_j ^ (hash_j >> 32));
  }

 public:
  int getNum(int row, int col) {
    auto rowCol = key(row, col);

    if (cache.count(rowCol) > 0)
      return cache[rowCol];

    if (row == 0 || col == 0 || row == col)
      return (cache[rowCol] = 1);

    return (cache[rowCol] = getNum(row - 1, col - 1) + getNum(row - 1, col));
  }

  vector<int> getRow(int rowIndex) {
    vector<int> ans;

    for (int i = 0; i <= rowIndex; i++)
```

But, it is worth noting that generating a number for a particular row requires only two numbers from the previous row. Consequently, generating a row only requires numbers from the previous row.

Thus, we could reduce our memory footprint by only keeping the latest row generated, and use that to generate a new row.

C++ Java Copy

```cpp
class Solution {
 public:
  vector<int> getRow(int rowIndex) {
    vector<int> curr, prev = {1};

    for (int i = 1; i <= rowIndex; i++) {
      curr.assign(i + 1, 1);

      for (int j = 1; j < i; j++)
        curr[j] = prev[j - 1] + prev[j];

      prev = move(curr);  // This is O(1)
    }

    return prev;
  }
};
```

> The `std::move()` operator on vectors in C++ is an $O(1)$ operation. [2]

Complexity Analysis

- Time complexity : $O(k^2)$.
  - Simple memoization would make sure that a particular element in a row is only calculated once. Assuming that our memoization cache allows constant time lookup and updation (like a hash-map), it takes constant time to calculate each element in Pascal's triangle.
  - Since calculating a row requires calculating all the previous rows as well, we end up calculating $1 + 2 + 3 + ... + (k+1) = \frac{(k+1)(k+2)}{2} \simeq k^2$ elements for the k^{th} row.
- Space complexity : $O(k) + O(k) \simeq O(k)$.
  - Simple memoization would need to hold all $1 + 2 + 3 + ... + (k+1) = \frac{(k+1)(k+2)}{2}$ elements in the worst case. That would require $O(k^2)$ space.
  - Saving space by keeping only the latest generated row, we need only $O(k)$ extra space, other than the $O(k)$ space required to store the output.

Approach 3: Memory-efficient Dynamic Programming

Intuition

Notice that in the previous approach, we have maintained the previous row in memory on the premise that we need terms from it to build the current row. This is true, but not *wholly*.

What we actually need, to generate a term in the current row, is *just* the two terms above it (present in the previous row).

Formally, in memory,

```
pascal[i][j] = pascal[i-1][j-1] + pascal[i-1][j]
```

where `pascal[i][j]` is the number in i^{th} row and j^{th} column of Pascal's triangle.

So, trying to compute `pascal[i][j]`, only the memory regions of `pascal[i-1][j-1]` and `pascal[i-1][j]` are required to be accessed.

Algorithm

Let's take a step back and analyze the circumstances under which `pascal[i][j]` might be accessed. Given that we have already employed DP to save us valuable run-time, the access pattern for `pascal[i][j]` looks a bit like this:

- `WRITE` `pascal[i][j]` (after generating it from `pascal[i-1][j-1]` and `pascal[i-1][j]`)
- `READ` `pascal[i][j]` to generate `pascal[i+1][j]`
- `READ` `pascal[i][j]` to generate `pascal[i+1][j+1]`

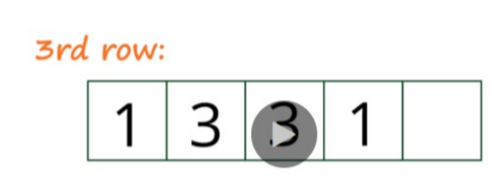
That's it! Once we've written out `pascal[i][j]`: 1. We don't ever need to modify it. 2. It's only read a *fixed* number of times, i.e. **twice** (thanks to DP).

Hypothetically, if we kept the the current row (in the process of being generated) and the previous row, in the same memory block, what kind of access patterns would we see (assume `pascal[j]` means the j^{th} number in a row)?

- `pascal[j]` was somehow generated in a previous instance. Currently, it holds the previous row value.
- `pascal[j]` (which holds the j^{th} number of the previous row) must be read when writing out `pascal[j]` (the j^{th} number of the current row).
  - Obviously they are the same memory location, so a conflict exists: the previous row value of `pascal[j]` will be lost after the write-out.
  - Is that ok? If we don't need to read the previous row value of `pascal[j]` anymore, is there any harm in writing out the current row value in its place?
- `pascal[j]` (which holds the j^{th} number of the previous row) must be read when writing out `pascal[j+1]` (the $j+1^{th}$ number of the current row). These are two different memory locations, so there is no conflict.

If we managed to keep all read accesses on the previous row value of `pascal[j]`, **before** any write access to `pascal[j]` for the current row value, we should be good! That's possible by evaluating each row from the end, instead of the beginning. Thus, a new row value of `pascal[j+1]` must be generated *before* doing so for `pascal[j]`.

The following animation demonstrates the above algorithm, used to generate the 4^{th} row of Pascal's Triangle, from an existing 3^{rd} row:



C++ Java Copy

```cpp
class Solution {
 public:
  vector<int> getRow(int rowIndex) {
    vector<int> ans = vector<int>(rowIndex + 1, 1);

    for (int i = 1; i < rowIndex; i++)
      for (int j = i; j > 0; j--)
        ans[j] += ans[j - 1];  // ans[j] = ans[j-1] + ans[j]

    return ans;
  }
};
```

Complexity Analysis

- Time complexity : $O(k^2)$. Same as the previous dynamic programming approach.
- Space complexity : $O(k)$. No extra space is used other than that required to hold the output.
- Although there is no savings in theoretical computational complexity, in practice there are some minor wins:
  - We have one vector/array instead of two. So memory consumption is roughly half.
  - No time wasted in swapping references to vectors for previous and current row.
  - Locality of reference shines through here. Since every read is for consecutive memory locations in the array/vector, we get a performance boost.

Approach 4: Math! (specifically, Combinatorics)

Intuition

Let's go back to the definition of a Pascal's Triangle:

> In mathematics, Pascal's triangle is a triangular array of the binomial coefficients.
> ... The entry in the n^{th} row and r^{th} column of Pascal's triangle is denoted $\binom{n}{r}$.

As a refresher, $\binom{n}{r} = \frac{n!}{r!(n-r)!}$.

Binomial coefficients have an additive property, known as Pascal's rule:

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r} \quad \forall \quad r, n \in \mathbb{N}^0,\ 0 \leq r \leq n$$

If you notice carefully, the terms $\binom{n-1}{r-1}$ and $\binom{n-1}{r}$, are the two numbers directly above the number $\binom{n}{r}$ in Pascal's triangle. This recurrence is same as the intuition for Approach 1.

Algorithm

While knowing Pascal's rule does not give us any benefits over previous approaches, knowing that the numbers in Pascal's triangle are just binomial coefficients will come in handy.

Successive binomial coefficients $\binom{n}{r-1}$ and $\binom{n}{r}$ differ by a factor of:

$$\frac{\binom{n}{r}}{\binom{n}{r-1}} = \frac{\frac{n!}{r! \cdot (n-r)!}}{\frac{n!}{(r-1)! \cdot (n-r+1)!}} = \frac{n-r+1}{r}$$

Thus, we can derive the next term in a row in Pascal's triangle, from a preceding term. Running a loop should give us the required row.

- We know that each row starts with a 1, so we have a starting point.
- We also know that the k^{th} row has exactly $k+1$ terms, so we know how long we need to run the loop.

C++ Java Copy

```cpp
class Solution {
 public:
  vector<int> getRow(int n) {
    vector<int> ans = {1};

    for (int k = 1; k <= n; k++)
      ans.push_back((int)((ans.back() * (long long)(n - k + 1)) / k));

    return ans;
  }
};
```

Complexity Analysis

- Time complexity : $O(k)$. Each term is calculated once, in constant time.
- Space complexity : $O(k)$. No extra space required other than that required to hold the output.

Further Thoughts

- The symmetry of a row in Pascal's triangle allows us to get away with computing just half of each row.

> Pop Quiz: Are there any computational complexity benefits of doing this?
>
> Pop Quiz: Can you prove *why* these rows are symmetrical?

---

1. This Stack Overflow answer has a good explanation. See the parallel between the time complexity recurrence and Pascal's rule.
2. Starting C++11, `std:move()` can be used to move resources across arguments or references. Since underlying representations are simply moved, and *not* copied, this can be a very efficient operation to transfer elements across collections or containers. See this Stack Overflow answer for more.

Rate this article:

Previous Next