COSC326 Étude 6: Counting It Up

Introduction

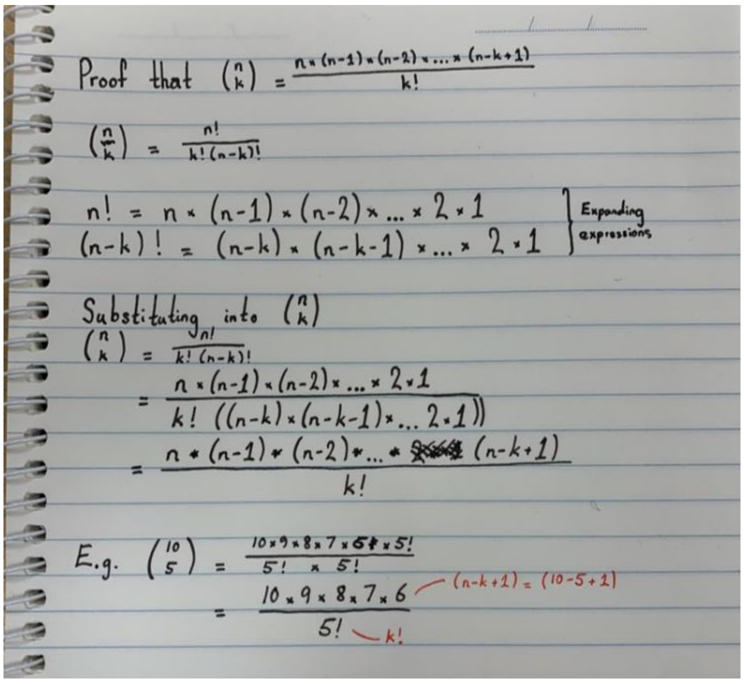
This report concerns testing and benchmarking regarding the binomial coefficient. The binomial coefficient, “n choose k” or $\binom{n}{k}$ denotes the number of ways to choose k objects from a set of n distinct objects without considering their order. The binomial coefficient was given to us in factorial notation which is expressed compactly as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

The binomial coefficient was calculated in the **BinomialCoeffcient.java** class using the multiplicative formula.

$$\binom{n}{k} = \frac{n \times (n-1) \times \cdots \times (n-k+1)}{k \times (k-1) \times \cdots \times 1},$$

Proof for this formula can be seen below



Proof that $\binom{n}{k} = \frac{n \times (n-1) \times (n-2) \times \ldots \times (n-k+1)}{k!}$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

$$\left.\begin{aligned} n! &= n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \\ (n-k)! &= (n-k) \times (n-k-1) \times \ldots \times 2 \times 1 \end{aligned}\right\} \text{Expanding expressions}$$

Substituting into $\binom{n}{k}$

$$\begin{aligned} \binom{n}{k} &= \frac{n!}{k!(n-k)!} \\ &= \frac{n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1}{k!\,((n-k) \times (n-k-1) \times \ldots 2 \times 1))} \\ &= \frac{n \times (n-1) \times (n-2) \times \ldots \times (n-k+1)}{k!} \end{aligned}$$

$$\text{E.g. } \binom{10}{5} = \frac{10 \times 9 \times 8 \times 7 \times 6\cancel{!} \times 5!}{5! \times 5!}$$

$$= \frac{10 \times 9 \times 8 \times 7 \times 6}{5!} \quad \begin{aligned} &(n-k+1) = (10-5+1) \\ &k! \end{aligned}$$

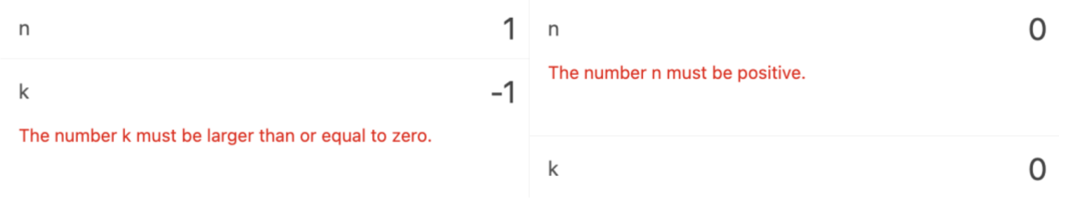
The multiplicative formula was chosen instead of the factorial formula because of the benefits to efficiency. Compared to the multiplicative formula the factorial function grows very quickly and computing it gets more computationally expensive for larger values. In contrast the multiplicative theorem only involves multiplication and division which are generally faster operations for computers to perform. The multiplicative formula still has a factorial in its equation being $k!$ but that's still much less than the factorial formula where $n!$ and $(n-k)!$ on top of $k!$

```
result = 1
for (i = 1; i <= k; i++) {
        result *= n - i + 1;
    result /= i;
}
```

The pseudo code above is how the multiplicative formula was implemented in order to calculate the binomial coefficient. By multiplying and dividing the result after each loop to help not overflow the intermediate result. Rather than for example the pseudo code below which divides the numerator and denominator at the very end which also requires one more loop as well.

```
numerator = 1;
for (i = n; i > n - k; i--) {
        numerator *= i;
}

denominator = 1;
for (i = 2; i <= k; i ++) {
        denominator *= i;
}

result = numerator / denominator;
```

The function only accepts positive values for n so equal or more than 1 and non-negative values for k so equal or more than 0. An IllegalArgumentException was thrown if these cases were hit with the corresponding error messages. These exceptions were established with the help of Maciej Kowalski [1] who is a PHD candidate who created an online binomial coefficient calculator.



n 1

k -1

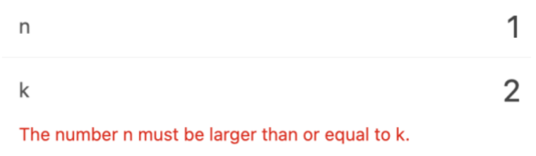
The number k must be larger than or equal to zero.

n 0

The number n must be positive.

k 0

[1] Kowalski, M. "Binomial Coefficient Calculator". Available at: https://www.omnicalculator.com/math/binomial-coefficient.

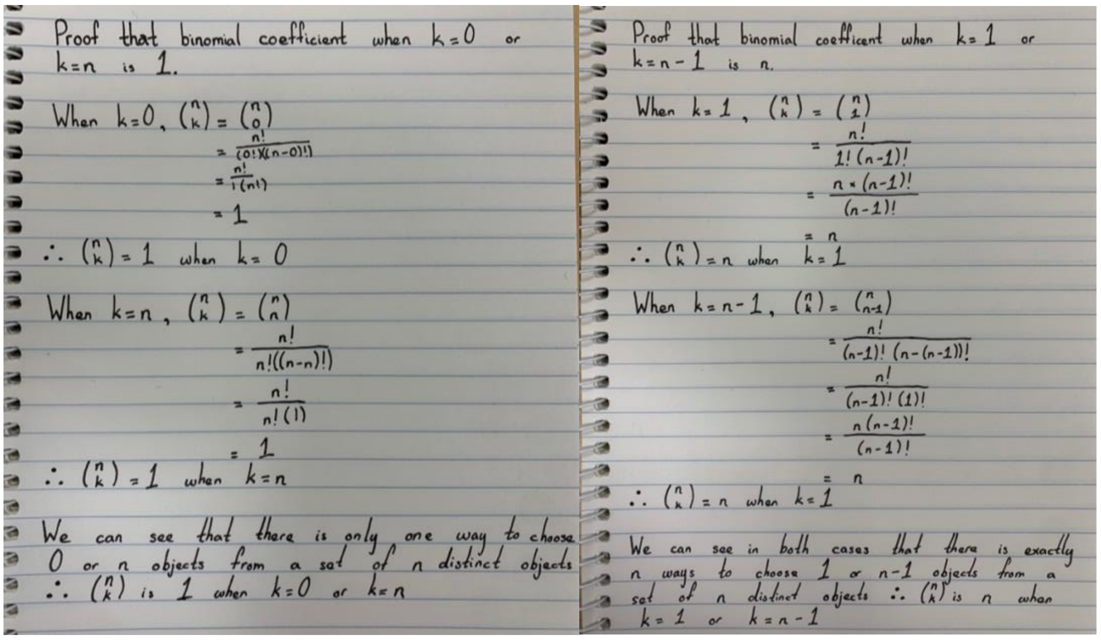


n	1
k	2

The number n must be larger than or equal to k.

His program was also used to verify the results of our calculation of the binomial coefficient.

Two cases for the binomial coefficient were established which didn't have to be calculated and could just be derived from the values of n and k. That being when $k = 0$ or when $k = n$ the result is 1 and when $k = 1$ and $k = n - 1$



Proof that binomial coefficient when $k=0$ or $k=n$ is 1.

When $k=0$, $\binom{n}{k} = \binom{n}{0}$

$$= \frac{n!}{(0!)((n-0)!)}$$

$$= \frac{n!}{1(n!)}$$

$$= 1$$

$\therefore \binom{n}{k} = 1$ when $k = 0$

When $k=n$, $\binom{n}{k} = \binom{n}{n}$

$$= \frac{n!}{n!((n-n)!)}$$

$$= \frac{n!}{n!(1)}$$

$$= 1$$

$\therefore \binom{n}{k} = 1$ when $k=n$

We can see that there is only one way to choose 0 or n objects from a set of n distinct objects $\therefore \binom{n}{k}$ is 1 when $k=0$ or $k=n$

Proof that binomial coefficent when $k=1$ or $k=n-1$ is n.

When $k=1$, $\binom{n}{k} = \binom{n}{1}$

$$= \frac{n!}{1!(n-1)!}$$

$$= \frac{n \times (n-1)!}{(n-1)!}$$

$$= n$$

$\therefore \binom{n}{k} = n$ when $k=1$

When $k=n-1$, $\binom{n}{k} = \binom{n}{n-1}$

$$= \frac{n!}{(n-1)!(n-(n-1))!}$$

$$= \frac{n!}{(n-1)!(1)!}$$

$$= \frac{n(n-1)!}{(n-1)!}$$

$$= n$$

$\therefore \binom{n}{k} = n$ when $k=1$

We can see in both cases that there is exactly n ways to choose 1 or n-1 objects from a set of n distinct objects $\therefore \binom{n}{k}$ is n when $k=1$ or $k=n-1$

The value for $k = 0$ and $k = n$ or $k = 1$ and $k = n - 1$ being the same makes sense because of the symmetrical nature of a binomial. The value for the output when k is 0 is the exact same when it's the largest possible value being n. This holds true for the next pair of values when k is 1 and $k = n - 1$ and etc. the value of the coefficient is symmetric around the middle point of n.

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

One way we optimized the binomial coefficient algorithm was by taking advantage of the symmetric property of the coefficient formula by always calculating the coefficient for the smaller value of k. This reduces the number of calculations and iterations required in the for loop, resulting in improved efficiency of the algorithm. This is done by taking the min amount between k and n – k.

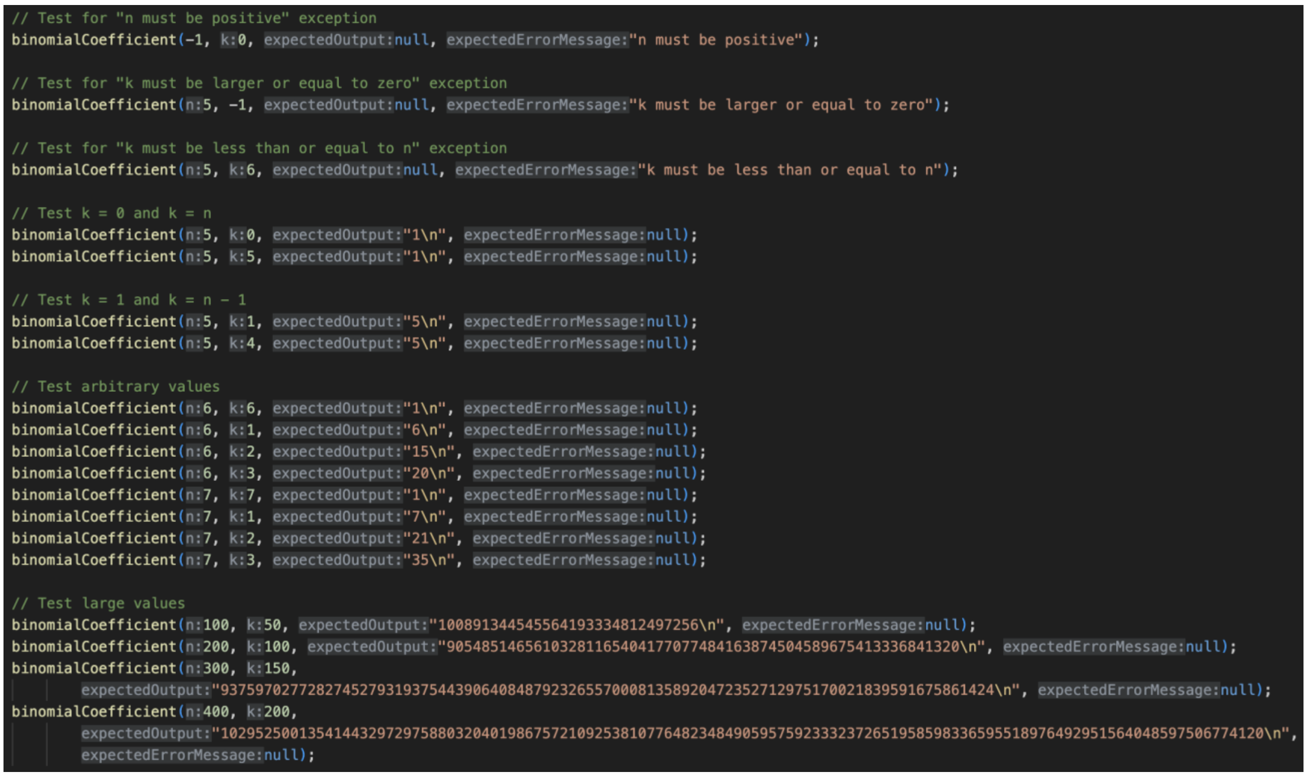
```
Long.min(k, n - k)
```

Testing

The testing of the application was done in the **BinomialCoeffcient.java** class was done with the help of JUnit tests.

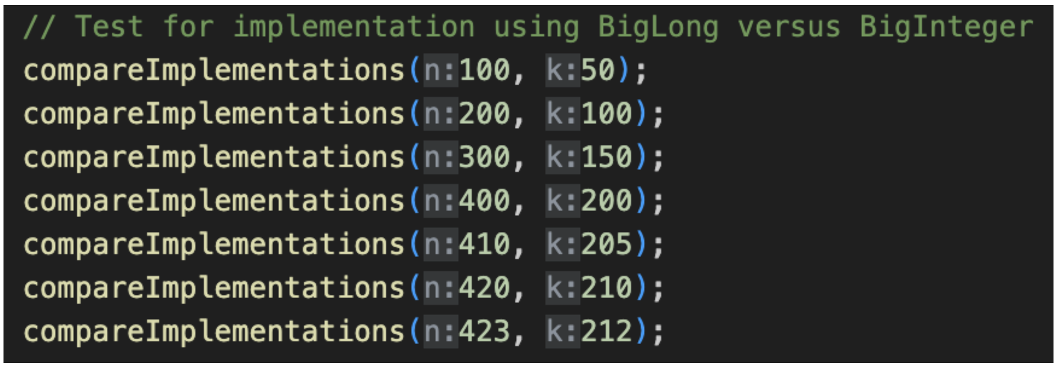
The **binomialCoefficient** function takes parameters n and k and compares $\binom{n}{k}$ to the expected output. The following cases were accounted for:

- Cases which returned exceptions were tested by matching the expected error message to the actual error message. i.e. non-positive values for n so equal or less than 0 and negative values for k so equal or than -1.
- Cases which didn't have to be calculated like $k = 0$ and $k = n$ or $k = 1$ and $k = n - 1$.
- Cases for arbitrary values.
- Cases for large values.



```
// Test for "n must be positive" exception
binomialCoefficient(-1, k:0, expectedOutput:null, expectedErrorMessage:"n must be positive");

// Test for "k must be larger or equal to zero" exception
binomialCoefficient(n:5, -1, expectedOutput:null, expectedErrorMessage:"k must be larger or equal to zero");

// Test for "k must be less than or equal to n" exception
binomialCoefficient(n:5, k:6, expectedOutput:null, expectedErrorMessage:"k must be less than or equal to n");

// Test k = 0 and k = n
binomialCoefficient(n:5, k:0, expectedOutput:"1\n", expectedErrorMessage:null);
binomialCoefficient(n:5, k:5, expectedOutput:"1\n", expectedErrorMessage:null);

// Test k = 1 and k = n - 1
binomialCoefficient(n:5, k:1, expectedOutput:"5\n", expectedErrorMessage:null);
binomialCoefficient(n:5, k:4, expectedOutput:"5\n", expectedErrorMessage:null);

// Test arbitrary values
binomialCoefficient(n:6, k:6, expectedOutput:"1\n", expectedErrorMessage:null);
binomialCoefficient(n:6, k:1, expectedOutput:"6\n", expectedErrorMessage:null);
binomialCoefficient(n:6, k:2, expectedOutput:"15\n", expectedErrorMessage:null);
binomialCoefficient(n:6, k:3, expectedOutput:"20\n", expectedErrorMessage:null);
binomialCoefficient(n:7, k:7, expectedOutput:"1\n", expectedErrorMessage:null);
binomialCoefficient(n:7, k:1, expectedOutput:"7\n", expectedErrorMessage:null);
binomialCoefficient(n:7, k:2, expectedOutput:"21\n", expectedErrorMessage:null);
binomialCoefficient(n:7, k:3, expectedOutput:"35\n", expectedErrorMessage:null);

// Test large values
binomialCoefficient(n:100, k:50, expectedOutput:"100891344545564193334812497256\n", expectedErrorMessage:null);
binomialCoefficient(n:200, k:100, expectedOutput:"90548514656103281165404177077484163874504589675413336841320\n", expectedErrorMessage:null);
binomialCoefficient(n:300, k:150,
        expectedOutput:"93759702772827452793193754439064084879232655700081358920472352712975170021839591675861424\n", expectedErrorMessage:null);
binomialCoefficient(n:400, k:200,
        expectedOutput:"102952500135414432972975880320401986757210925381077648234849059575923332372651958598336595518976492951564048597506774120\n",
        expectedErrorMessage:null);
```

The **compareImplementations** function takes parameters n and k and compares $\binom{n}{k}$ using custom BigLong implementation versus using BigInterger. This was mainly used to test values that outputs exceeded the max long value.



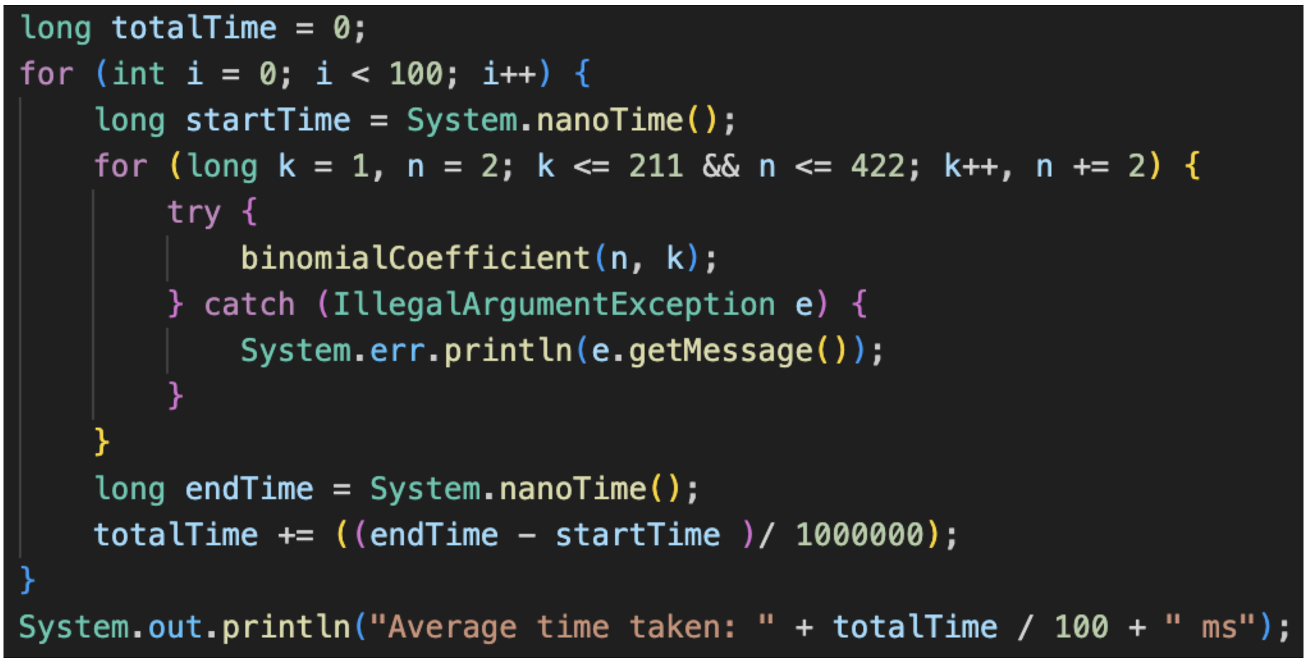
```
// Test for implementation using BigLong versus BigInteger
compareImplementations(n:100, k:50);
compareImplementations(n:200, k:100);
compareImplementations(n:300, k:150);
compareImplementations(n:400, k:200);
compareImplementations(n:410, k:205);
compareImplementations(n:420, k:210);
compareImplementations(n:423, k:212);
```

This was done by comparing the output of **Binomial.binomialCoefficient** to the test specific function **binomialCoefficientUsingBigInteger** which are identical functions apart from the fact that one uses the custom **BigLong.java** class while the over just uses BigInteger. These two implementations were compared because the purpose of the étude was to deal with overflow without using high precision classes like BigInteger. The **BigLong.java** class has implemented its own multiply and divide functions using column multiplication and long division. That way it could handle some overflow if the intermediate value went beyond 9223372036854775807 which is the value of the max long in Java. The value for k that was used in the **compareImplementations** was usually n / 2 because if we know the largest value of $\binom{n}{k}$ was computed correctly we can infer that the values below that were all computed correctly.

Benchmarking

The **BinomialCoefficient** is accurate up to $\binom{424}{212}$ where beyond that the multiply and divide function can't handle the overflow and many "-" symbols are inserted into the output. The function works values up to:

- 838,859,504,238,807,470,085,324,211,200,052,525,323,190,709,446,713,913,854,783,565,098,627,679,224,848,339,589,550,763,470,754,867,947,685,242,892,146,761,968,759,160.

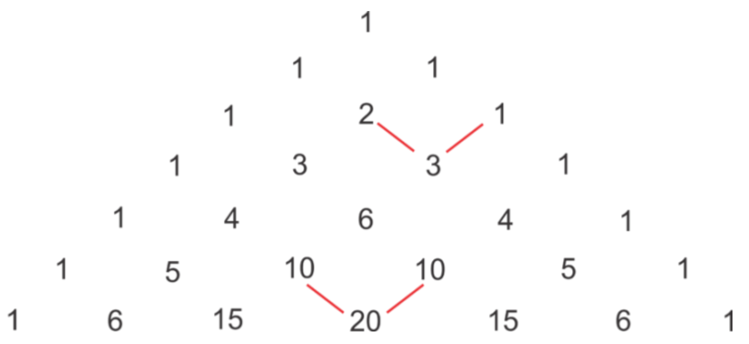


```
long totalTime = 0;
for (int i = 0; i < 100; i++) {
    long startTime = System.nanoTime();
    for (long k = 1, n = 2; k <= 211 && n <= 422; k++, n += 2) {
        try {
            binomialCoefficient(n, k);
        } catch (IllegalArgumentException e) {
            System.err.println(e.getMessage());
        }
    }
    long endTime = System.nanoTime();
    totalTime += ((endTime - startTime )/ 1000000);
}
System.out.println("Average time taken: " + totalTime / 100 + " ms");
```

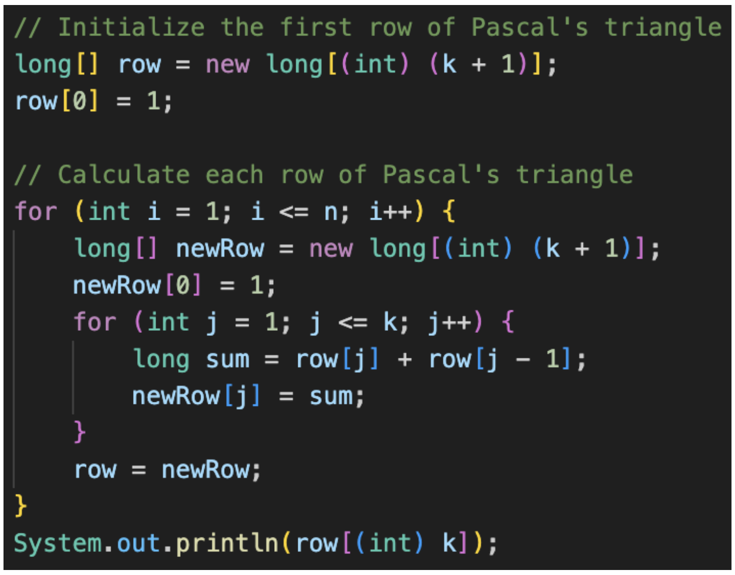
This is the code I used to calculate the computing times of the function. On average the **binomialCoefficient** function took around 15 ms while the **binomialCoefficientUsingBigInteger** function took 5 ms. This is to be expected because BigInterger will use much more efficient algorithms to multiply and divide compared to what we implemented.

Comparing my approach to Pascal's Triangle

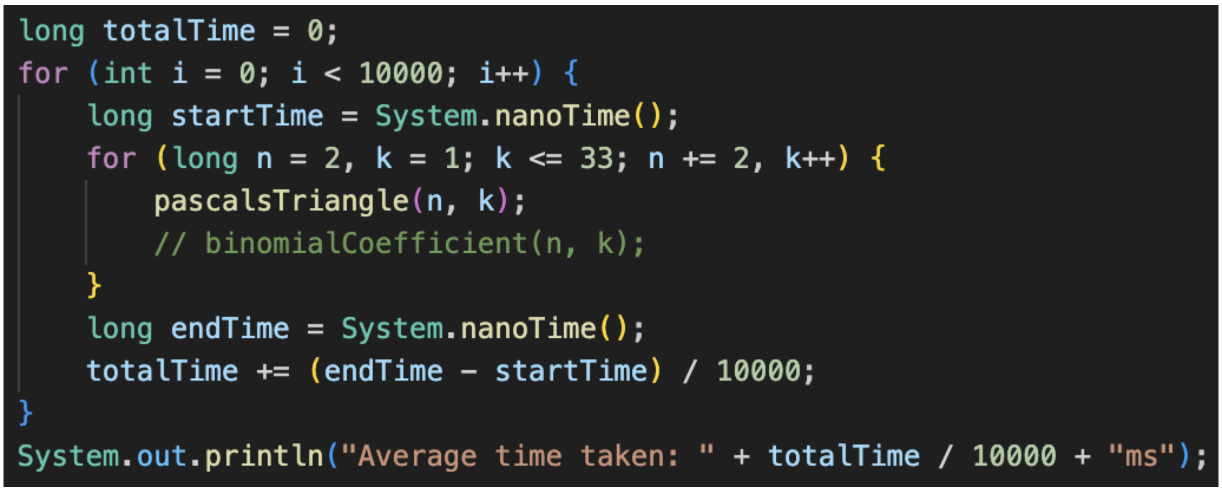
Another way the binomial coefficient can be calculated is by using Pascal's triangle. The formula states that for $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k+1}$. Basically for each element in the current row, calculate its value by adding the corresponding elements from the previous row. Where the corresponding elements are the element at the same index and the element at the index one less than the current index. This… implementation doesn't worry about overflow because the only values that are overflowing are those whose output is greater than the max Long value in Java since only addition is occurring.



This can be easily translated in to code by creating an array which values are calculated from that of the previous row.


```java
// Initialize the first row of Pascal's triangle
long[] row = new long[(int) (k + 1)];
row[0] = 1;

// Calculate each row of Pascal's triangle
for (int i = 1; i <= n; i++) {
    long[] newRow = new long[(int) (k + 1)];
    newRow[0] = 1;
    for (int j = 1; j <= k; j++) {
        long sum = row[j] + row[j - 1];
        newRow[j] = sum;
    }
    row = newRow;
}
System.out.println(row[(int) k]);
```

How does this compare my previous implementation?



```
long totalTime = 0;
for (int i = 0; i < 10000; i++) {
    long startTime = System.nanoTime();
    for (long n = 2, k = 1; k <= 33; n += 2, k++) {
        pascalsTriangle(n, k);
        // binomialCoefficient(n, k);
    }
    long endTime = System.nanoTime();
    totalTime += (endTime - startTime) / 10000;
}
System.out.println("Average time taken: " + totalTime / 10000 + "ms");
```

Figure 1: Code used to benchmark

On average using Pascal's triangle the binomial coefficient takes on average 15 milliseconds to calculates $\binom{n}{k}$ from $\binom{2}{1}$ to $\binom{66}{33}$ where n is twice that of k, while my previous implementation took on average 24 milliseconds. This is an increase of around 60% versus the previous implementation. Multiplication and division operations are generally computationally more expensive compared to addition operations. This is because multiplication and division involve more intricate calculations and usually require greater computational resources. In contrast, addition is a simpler operation that can be efficiently executed in hardware, requiring fewer computational resources and yielding faster results than multiplication and division.

Does this mean that using Pascal's triangle is definitively more efficient than implementing a high-precision class? No because the computation of the binomial coefficient using Pascal's triangle can become computationally expensive for large values of n and k. The size of the triangle can grow significantly, leading to memory and performance issues. This is demonstrated for the case $\binom{4294967296}{2}$ which takes an extended period of time which is almost impossible to calculate because it will have to loop 4,294,967,296 times plus compute the current values from the previous rows. As soon as $n \gg k$ it's better to use an approach like my previous implementation which would just have to calculate 4294967296 / 1 x 4294967295 / 2 which is only three operations versus upwards of a billion calculations. A new base case was created for this specifically for when k = 2 or k = n – 2 that will just compute $\frac{n}{2}(n-1)$. $\frac{n}{2}$ is calculated first to decrease the value before multiplying it again to prevent any overflow that might occur. This demonstrates some of the limitations of integer arithmetic for when $n \gg k$.