# Counting it up

# Etude 6

# COSC326

**Problem summary**

The problem for etude 6: Counting it up, was to create a program that can ouput the binomial coefficient of an n k input. This binomial coefficient is a formula that calculates the number of different groups of size k in a set of n. For example, the different number of sets of 5 cards (k) if a 52 card (n) deck. The problem was to only use 64-bit integers (long's in java) and allow for outputs up to the size of a max 64-bit integer which is 9,223,372,036,854,775,807.

**Approach summary**

We had 4 main approaches: factorial approach, digested factorial approach (the count classes), pascals triangle approach (the pascal triangle class), and a formulaic pascal's triangle approach (the formulaC class). The last two approaches were the most successful with one having high efficiency but not 100% accuracy and the other having less efficiency but 100% accuracy (as explained below).

**Approaches**

Approach 1: Factorial approach

Originally, we thought this problem was simple. Simply input k and n and ouput the answer to the formula for k and n. After we realised this failed for inputs of n greater than k we thought the solution was extend the program to get it to work for as high of an input as we could. This approach was based on a misunderstanding of the task. Originally we didn't think that we were trying to get every input that produced an output at max 64-bit integer but instead we were just trying to get the highest output we could. After we realised the solution was to allow for outputs up to max 64-bit int we stopped trying to extend the program to work for larger inputs and tried a new approach altogether.

Approach 2: Digested factorial approach

For our next approach we examined the formula and digested it. The formula for the binomial coefficient is n! / (k!(n-k)!. From this we realised we could simplify the equation by first dividing n! by the larger for k! and (n-k)!. For example, if we had n = 10 and k = 6 then the formula was:

n! = 10*9*8*7*6*5*4*3*2*1

k! = 6*5*4*3*2*1

(n-k)! = 4*3*2*1

Which makes the formula:

(10*9*8*7*6*5*4*3*2*1) / ((6*5*4*3*2*1)*(4*3*2*1))

You can divide n! by k! to get:

(10*9*8*7) / (4*3*2*1).

This was a much simpler equation to work with. We then though, to reduce the chance of the long overflowing we should alternate between multiplying and dividing. This resulted in our first iteration of count (count 1), which worked with 100% accuracy, but only if there was never a division that resulted in a non-whole number.

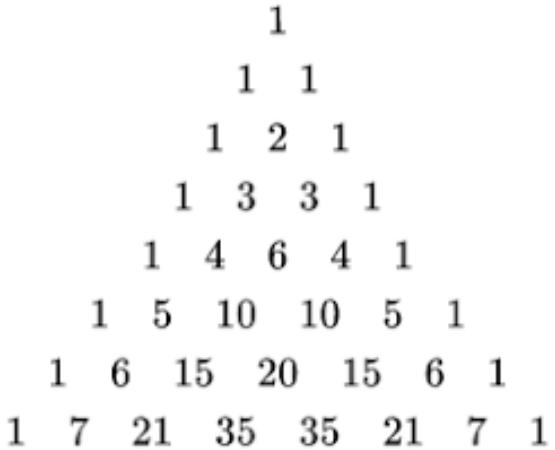
Once we realised this issue we changed all the longs to doubles, as doubles were also of 64-bit size. However, due to doubles allowing decimals, they couldn't fit as many whole numbers as longs could, so we scratched this idea.

Our third implementation required a bit of math. In math you can separate a number, multiply both by x and add them together which would be the same as simply multiplying the original number by x. For example, if you had 2.5 * 2 you could just do 2.5*2 = 5 or you could do 2*2 + 0.5*2 = 5. We used this to separate the decimal part each time and add it to a double called excess we would multiply and divide excess the same as nk but add to excess each time nk resulted in a decimal. Then, at the end, we added excess back to nk. This approach almost worked but eventually excess would get too big in the later iterations

This resulted in our final attempt with this approach. Instead of adding excess to nk at the end of the entire method, we would add the whole part of excess to nk at the end of each iteration and minus the whole part of excess off of excess. This would always keep excess below 2 meaning no overflow could occur. This approach was very efficient as it only looped a max of n/2 times. And it was almost 100% accurate, however, for larger output values the accuracy wasn't exact. This was due to rounding errors. Unfortunately a double can't be 100% accurate with their decimal places. The issue with this was that with larger outputs, the double excess being slightly off caused the output to be slightly off which meant it wasn't a correct solution. We tried a lot of methods to solve this issue, our main idea being to get the remainder of each time and instead set excess equal to the remainder divided by I but unfortunately this didn't work as the remainder couldn't always be predicted at the right part of the loop. Unfortunately, we realised the math was beyond us and we moved on to a different approach.

Approach 3: Pascals triangle approach

Pascals triangle (shown to the right) is a triangle of numbers where each number of the triangle is the sum of the numbers above. The pascal's triangle approach is very simple, the binomial coefficient of n choose k is the (n+1)th row and (k+1)th column of pascals triangle. For example, n = 6 and k = 2 binomial coefficient is 15, the number in the 7$^{th}$ row and 3$^{rd}$ column of the triangle.

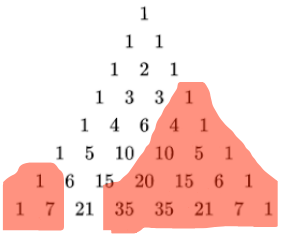



Our original idea was to simply output the whole triangle up to row (n+1) and then output the number in position (k+1). We did this by making a 2d array of size (n+1) with the size of each 1D array inside the 2D array equalling i

(for each iteration). We would then simply output the number at position (k+1) in the final row. The problem with this idea was that this quickly resulted in max memory usage when you started using millions+ rows.

Our next idea was to create a 2D array of size 2, with the 1D arrays all equal to size (n+1), the size required for the last row. This eliminated our max memory use as the amount of memory used was always 2 * (n+1). Our idea was the use the first array in the 2D array to set the numbers for the next row in the second array, and then put the results in the first array and iterate. For example, if array 1 contained {1, 1}, we would use this to input {1, 2, 1} in array 2 and then set array 1 = array 2. This approach worked with 100% accuracy, but it was very inefficient and there was one main issue: it only worked for inputs less than the size of max int as you could only set the size of an array to an int. We decided to put this issue aside and first focus on increasing efficiency.

After evaluating the triangle we realised that you only actually needed part of the triangle to get the answer of n k. For example (on the right), if n = 7 and k = 2 (nk = 21), everything in red isn't needed to calculate the binomial coefficient. Therefore, you would only need to have an array of size 3 that loops (n+1) times. This implementation moderately reduced the efficiency for all inputs less than max int.




Approach 4: Formulaic Approach

Originally, we used the formulaic approach as a way of getting round the max int problems. The only k inputs that would allow for n inputs above max int was k = 1 and k = 2. Therefore, it was simple to solve this. If k = 1 you just want to print out n. If k = 2 however, what do you print out. After evaluating pascal's triangle we realised that all k = 2 inputs were the triangle number for n. This can be seen if you rotate pascal's triangle and look in column 3 (on the right). The triangle number of n is simply n + n-1 + n-2 all the way to 1. There is, however, a simple formula (that I had learnt prior to solving this problem) for the pyramid number of x which is x * (x+1)/2. We used this for the situation where k = 2. However, there were two things we noted for this, one, there was a rounding error with k = 2 that needed to be looked at. And two, if we could figure out a formula that calculates the binomial efficient without having to create pascal's triangle we could completely reduce efficiency issues.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 3 | 6 | 10 | 15 | 21 | 28 |
| 1 | 4 | 10 | 20 | 35 | 56 | 84 |
| 1 | 5 | 15 | 35 | 70 | 126 | 210 |
| 1 | 6 | 21 | 56 | 126 | 252 | 462 |
| 1 | 7 | 28 | 84 | 210 | 462 | 924 |

We first looked at solving the rounding error through using doubles. This was easy after considering how much we had learnt about rounding errors from the first two approaches to this puzzle. We simply calculated the extra amount missed from rounding and added it to the sum. After this we evaluated the table above.

We know if k = 1, nk = n, if k = 2, nk = n * (n+1)/2. With this knowledge, could we create an iterative formulaic approach to calculate nk given k. We first realised again, we don't actually have to loop k times, it would just be the smaller of n-k and k as the binomial coefficient for k = 1 and k = (n-1) is the same (just like k = 2 and k = n-2 is the same and so on). This meant we

would never have to calculate anything more than n/2 which, in turn, meant, given output could only be of max long size, the maximum number of iterations would be 33 (66 33 is the max k).

Now knowing this would be an efficient approach, we looked at the formula. We first tried (for k = 3) n * (n+1)/2 * (n+2)/2 which wasn't correct. We then tried n * (n+1)/2 * (n+2)/3 which did work. This worked for all k which meant we had found an efficient formula that solved the nk problem with little code and 100% accuracy. The final step was to figure out what the row and columns should be given n.

To work this out we realised we wanted to skip the first row and column (row of 1's and column of 1's) in our equation, therefore, k and n should always add 1. We also realised, because we had rotated pascals triangle, the row number we wanted was actually n – k, not n. For example (on the right), for 6 3 we wanted 20, but instead we got 56. To account for this we simply subtracted k from the row size which worked at calculating the rows and columns we needed.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 3 | 6 | 10 | 15 | 21 | 28 |
| 1 | 4 | 10 | 20 | 35 | 56 | 84 |
| 1 | 5 | 15 | 35 | 70 | 126 | 210 |
| 1 | 6 | 21 | 56 | 126 | 252 | 462 |
| 1 | 7 | 28 | 84 | 210 | 462 | 924 |

**Testing**

After testing the base cases we realised there was an issue. Rounding. To calculate the excess, we had to use a double. The issue with doubles is they can only have a max number of decimal places equal to 15. This meant our answers, when the output was 19 digits long (the max long length), could be off by 9999 which is what occurred in a few of our outputs.

**Efficiency and accuracy - Formulaic approach**

Efficiency:

In terms of efficiency, our final implementation has O(k) efficiency or O(n-k) efficiency. This was implemented through the knowledge that it didn't matter if you iterated through the rows and calculated using columns or iterated through columns and calculated using rows (e.g., row = 6, col = 2 has nk of 6, row = 2, col = 6 has nk of 6). This meant our max amount of loops was 33 as stated earlier. This means our time complexity is technically linear time O(n) (or O(log n) depending on how you analyse it). However, Because our max number of iterations is 33, the time complexity could be considered linear time O(33). Making it a very efficient implementation, much more so than our pascal's triangle approach.

Accuracy:

From our testing at limit cases (max cases for each k given output < MAX_LONG). We found all cases where overflow couldn't occur had 100% accuracy, however, cases with overflow (which tended to be cases with a higher k value) would be inaccurate.

Overall:

This approach is very efficient but there is an accuracy issue at higher k values as to get the excess you have to do excess*sum (where excess <= k) which can result in overflow. There

are ways to reduce this likelihood of overflow such as dividing excess and I by their highest common factor. However, this method results in increased efficiency and doesn't 100% remove the overflow option.

**Efficiency and accuracy – Pascal's triangle approach**

Efficiency:

Increases to O(n * (k-n)) or O(n * k) if k < (n-k). This is quadratic time complexity (constant * constant). However, as k (or n-k) is max 33 it could be considered constant time complexity of max O(33n). This is much less efficient than the formulaic approach but allows for increased accuracy.

Accuracy:

Because this approach doesn't rely on calculations with non-whole numbers overflow can't occur with numbers where output < MAX_LONG. This is because you are only every adding to get to the output meaning you will never use a number > output that's relevant to calculating output. Therefore, this approach gives 100% accuracy for all cases where output < MAX_LONG.

Overall:

While less efficient, this approach is 100% accurate making it our best approach.

**Summary**

To make the most efficient yet 100% accurate program you should mix the last 2 results so that you use the formulaic approach for low k values and the pascal's triangle approach for high values. To determine the value of k that you need to start using pascals triangle you would have to first to an evaluation of all possible inputs that result in overflow and calculate the input with the lowest k resulting in overflow. You should then set this as the max k for using the formulaic approach. Or, if you wanted to have max efficiency, you would create a hashmap that determines all the inputs with overflow and then, for these inputs, use the pascals tringle approach else use the formulaic approach.

**Submission**

For our submission we gave all pieces of code we worked on as it shows our process. However, because the pascal's triangle approach is the only one with 100% accuracy, that is the one we would like to submit.