Hack 12.0

Computer Science I – Java
Recursion & Memoization

Department of Computer Science & Engineering

University of Nebraska–Lincoln

Introduction

Hack session activities are small weekly programming assignments intended to get you started on full programming assignments. Collaboration is allowed and, in fact, *highly encouraged*. You may start on the activity before your hack session, but during the hack session you must either be actively working on this activity or *helping others* work on the activity. You are graded using the same rubric as assignments so documentation, style, design and correctness are all important.

Problem Statement

A binomial coefficient, "n choose k" is a number that corresponds to the number of ways to *choose* k items from a set of n distinct items. You may be familiar with some the notations, $C(n, k)$ or C_n^k or ${}_nC_k$, but most commonly this is written as

$$\binom{n}{k}$$

and read as "n choose k". There is an easy to compute formula involving factorials:

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

For example, if we have $n = 4$ items, say $\{a, b, c, d\}$ and want to choose $k = 2$ of them, then there are

$$\binom{4}{2} = \frac{4!}{(4-2)!2!} = 6$$

ways of doing this. The six ways are:

$$\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}$$

There are a lot of other interpretations and applications for binomial coefficients, but this hack will focus on computing their value using a different formula, Pascal's Rule[1]:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

which is a recursive formula. The base cases for Pascal's Rule are when $k = 0$ and $n = k$. In both cases, the value is 1. When $k = 0$, we are not choosing any elements and so there is only one way of doing that (i.e. choose nothing). When $n = k$ we are choosing every element, again there is only one way of doing that.

Writing a Naive Recursion

Create a class named `Binomial` and implement and test the following method *using a recursive* solution:

```
public static long choose(int n, int k)
```

which takes n and k and computes $\binom{n}{k}$ using Pascal's Rule. Note that the return type is a `long` which is a 64-bit integer allowing you to compute values up to

$$2^{63} - 1 = 9,223,372,036,854,775,807$$

(a little over 9 quintillion). Write a `main` method that takes n and k as command line arguments and outputs the result to the standard output so you can easily test it.

Benchmarking

Run your program on values of n, k in Table 1 and time (roughly) how long it takes your program to execute. You can check your solutions with an online tool such as https://www.wolframalpha.com/.

n	k
4	2
10	5
32	16
34	17
36	18

Table 1: Test Values

Now formulate an estimate of how long your program would take to execute with larger values. You can make a *rough* estimate how many method calls are made using the

[1] Which can be used to generate Pascal's Triangle, https://en.wikipedia.org/wiki/Pascals_triangle

binomial value itself. That is, to compute $\binom{n}{k}$ using Pascal's Rule would make *about* $\binom{n}{k}$ method calls.

Use the running time of your program from the test values to estimate how long your program would run for the values in Table 2.

$\binom{n}{k}$	value
$\binom{54}{27}$	= 1,946,939,425,648,112
$\binom{56}{28}$	= 7,648,690,600,760,440
$\binom{58}{29}$	= 30,067,266,499,541,040
$\binom{60}{30}$	= 118,264,581,564,861,424
$\binom{62}{31}$	= 465,428,353,255,261,088
$\binom{64}{32}$	= 1,832,624,140,942,590,534
$\binom{66}{33}$	= 7,219,428,434,016,265,740

Table 2: Larger Values

Improving Performance with Memoization

You'll now improve your program's performance using memoization to avoid unnecessary repeated recursive calls.

1. First, change your return type to use Java's `BigInteger` class. This is an arbitrary precision number class meaning that it can represent arbitrarily large integer values. You won't be able to use the normal arithmetic operators however. Instead, you'll need to RTM and use the class's methods to add and perform other operations. See the documentation here: https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/math/BigInteger.html

2. To "cache" values so that you are not continually repeating the same calculations over-and-over you *could* use a table, but let's use a *smart data structure*: a `Map`.

   This map will be used to map a pair of input values, (n, k) to the value of the binomial $\binom{n}{k}$. The problem is that we want to use the combination of two values as a single key. To do so, we've provided a `Pair` class that allows you to pair two objects together to use as a key.

   Create and instantiate a static map of the following type:

   ```
   Map<Pair<Integer, Integer>, BigInteger>
   ```

3. Modify your `binomial` method to use this map to store and use values to avoid unnecessary repeated recursive calls.

   When the method needs to compute $\binom{n}{k}$ it checks the map first: if the value has already been computed (is not `null`) then it returns that value. Otherwise, it

performs the recursive computation. Before returning the value, however, it should store it (*cache* it) in the map so that subsequent computations avoid the recursion.

4. Rerun your program with the values in Tables 1 and 2 to verify they work and note the difference in running time.

Instructions

- All your code should be in the class file, `Binomial.java` along with full documentation.
- You are encouraged to collaborate any number of students before, during, and after your scheduled hack session.
- Include the name(s) of everyone who worked together on this activity in your source file's header.
- Turn in all of your files via webhandin, making sure that it runs and executes correctly in the webgrader. Each individual student will need to hand in their own copy and will receive their own individual grade.