Data Structures (CS 1520) Homework #4 Due: 10/13/12 (Sat.) at 11:59 PM

Objective: To experiment with recursion, and see the improvement between a recursive divide-and-conquer implementation and an iterative dynamic programming implementation.

The Assignment Overview: (Download hw4.zip at http://www.cs.uni.edu/~fienup/cs1520f12/homework/)

In Discrete Structures (CS 1800) you used (or will use) the binomial coefficient formula:

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

to calculate the number of combinations of "n choose k," i.e., the number of ways to choose k objects from n objects. One problem with using the above formula in most languages is that n! grows very fast and overflows an integer representation before you can do the division to bring the value back to a value that can be represented. (Python does not suffer from this problem, but lets pretend that it does.)

For example, when calculating the number of unique 5-card hands from a standard 52-card deck (e.g., C(52, 5)) we need to calculate 52! / 5! 47!. However, the value of

52! = 80,658,175,170,943,878,571,660,636,856,403,766,975,289,505,440,883,277,824,000,000,000,000

which is much, much bigger than can fit into a 64-bit integer representation. Fortunately, another way to view C(52, 5) is recursively by splitting the problem into two smaller problems by focusing on:

- the hands containing a specific card, say the ace of clubs, and
- the hands that do not contain the ace of clubs.

For those hands that do contain the ace of clubs, we need to choose 4 more cards from the remaining 51 cards, i.e., $C(51, 4)$. For those hands that do not contain the ace of clubs, we need to choose 5 cards from the remaining 51 cards, i.e., $C(51, 5)$. Therefore, $C(52, 5) = C(51, 4) + C(51, 5)$.

In general,

(2) $$C(n,\ k) = C(n-1,\ k-1) + C(n-1,\ k) \quad \text{for } 1 \le k \le (n-1), \text{ and}$$
$$C(n,\ k) = 1 \quad \text{for } k = 0 \text{ and } k = n$$

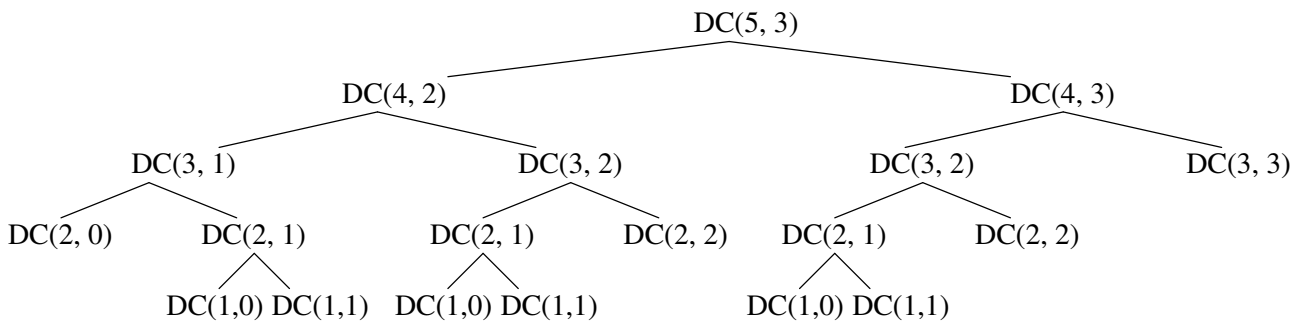
Part A: Implement a recursive factorial function, called `factorial(n)` using the recursive definition:

(3) $$n! = n * (n - 1)! \quad \text{for n} \ge 1, \text{ and}$$
$$0! = 1$$

Utilizing your factorial function and Python's built-in integer type, implement the binomial coefficient function directly using equation (1), call this function `C(n,k)`. You can use this as a check on your results from Parts B and C.

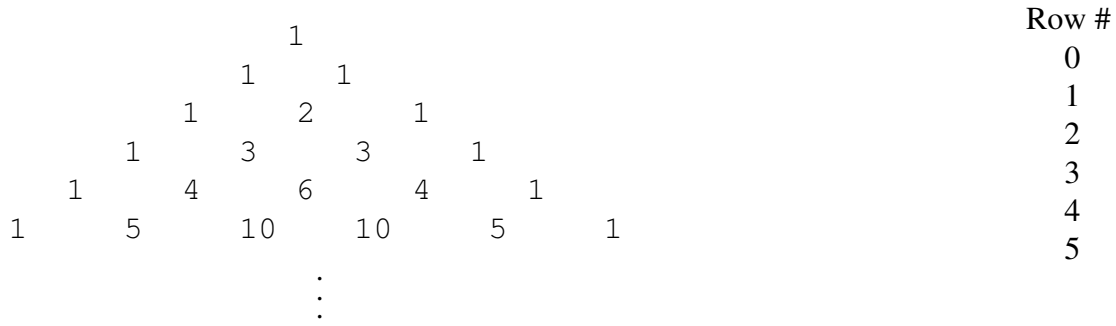
Part B: Implement the recursive "divide-and-conquer" binomial coefficient function using equation (2). Call your function `DC(n,k)` for "divide-and-conquer". Notice the difference in run-time between calculating the binomial coefficient using C(24, 12) vs. DC(24, 12), C(26, 13) vs. DC(26, 13), and C(28, 14) vs. DC(28, 14).

Part C: Much of the slowness of your "divide-and-conquer" binomial coefficient function, DC(n, k), is due to redundant calculations performed due to the recursive calls. For example, the recursive calls associated with DC(5, 3) = 10 would be:

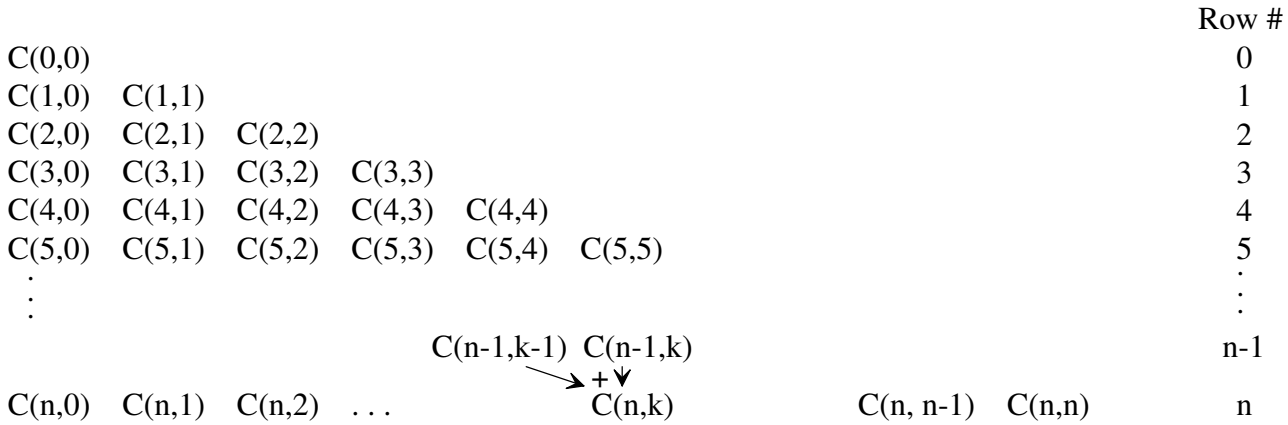


Data Structures (CS 1520) Homework #4 Due: 10/13/12 (Sat.) at 11:59 PM

Pascal's triangle (named for the 17th-century French mathematician Blaise Pascal, and for whom the programming language Pascal was also named) is a "dynamic programming" approach to calculating binomial coefficients.



Recall that dynamic programming solutions eliminate the redundancy of recursive divide-and-conquer algorithms by calculating the solutions to smaller problems first, storing their answers, and looking up their answers if later needed instead of recalculating it. Abstractly, Pascal's triangle relates to the binomial coefficient as in:



For Part C, you job is to implement the "dynamic programming" binomial coefficient function using Python lists and loops (no recursion needed). Call your function `DP(n,k)` for "dynamic programming". Notice the difference in run-time between calculating the binomial coefficient using DC(24, 12) vs. DP(24, 12), DC(26, 13) vs. DP(26, 13), and DC(28, 14) vs. DP(28, 14).

Strong Hints for Part C:

- Review the dynamic programming fibonacci example from Lecture 9. File `hw4/fibonacci.py` contains the recursive divide-and-conquer, and two dynamic programming versions of fibonacci. The first dynamic programming version, `fib_DP`, stores the answers to all of the smaller problems. The second dynamic programming version, `fib_DP2`, stores only the answers to the previous two smaller problems.
- Your function `DP(n,k)` should use the idea of `fib_DP2` to avoid storing all the answers to the smaller problems. Notice that the calculation of the next row in the picture above only needs the previous row and none of the older rows.

Submit all three parts as a single zipped file (called hw4.zip) electronically at

https://www.cs.uni.edu/~schafer/submit/which_course.cgi