Design & Analysis of Algorithms Name:_____

In Mathematics the factorial function is usually written as n!. For example, 5! = 5 x 4 x 3 x 2 x 1. You've probably seen it implemented as a recursive function, called factorial(n) using the recursive definition:

$n! = n * (n - 1)!$ for n ≥ 1, and

0! = 1 for n = 0

In Discrete Structures you used the binomial coefficient formula:

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

to calculate the number of combinations of "n choose k," i.e., the number of ways to choose k objects from n objects. For example, the number of unique 5-card hands from a standard 52-card deck is C(52, 5).

One problem with using the above binomial coefficient formula directly 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. When calculating the number of unique 5-card hands from a standard 52-card deck (e.g., C(52, 5)) for example, 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

is much, much bigger than can fit into a 64-bit integer representation.

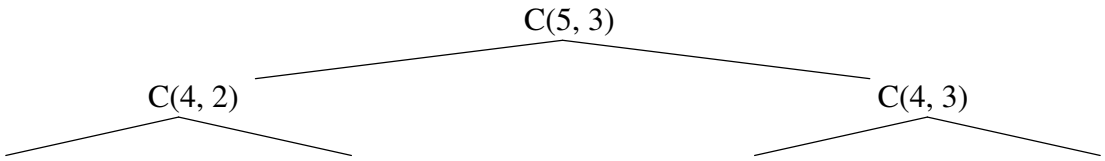
Fortunately, another way to view C(52, 5) is recursively by splitting the problem into two smaller problem by focusing on the hands containing a specific card, say the ace of clubs, and those that do not. For those hands that do contain the ace of clubs, we need to choose 4 more cards from the remaining 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, $C(52, 5) = C(51, 4) + C(51, 5)$.

1. Write the recursive definition for the binomial coefficient.

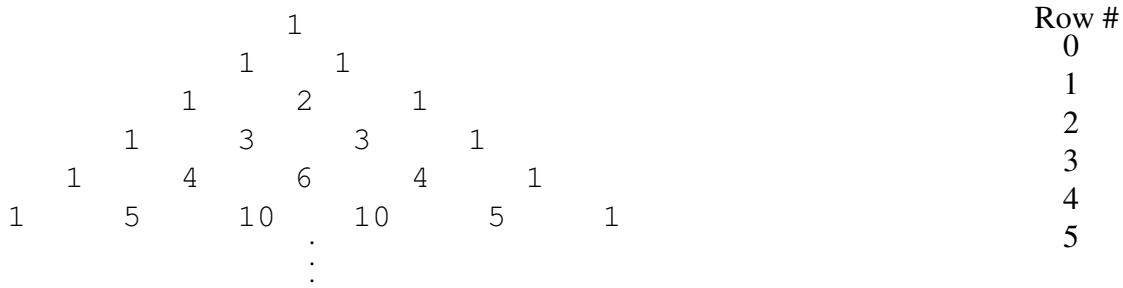
$C(n, k) =$ for $1 \le k \le (n - 1)$, and

$C(n, k) =$ for $k = 0$ or $k = n$

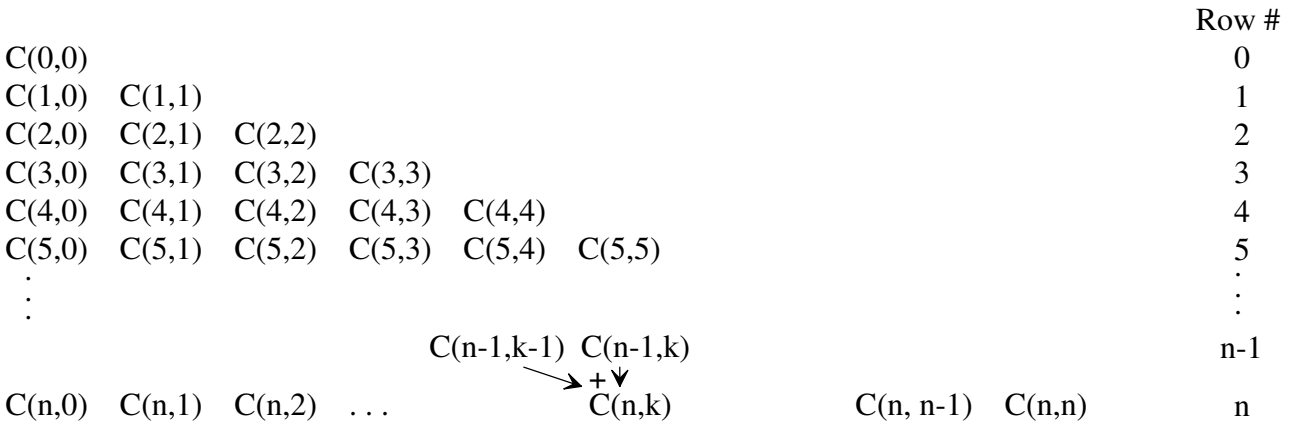
2. As you might guess, implementing the recursive "divide-and-conquer" binomial coefficient function using its recursive definition is slow due to redundant calculations performed due to the recursive calls. Complete the call tree for C(5, 3) = 10 is:



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. It is general written with numeric values in the form:



Recall that dynamic programming solutions eliminate the redundancy of 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:



3. Write psuedo-code for the "dynamic programming" binomial coefficient function using a two-dimensional array and loops (no recursion needed).

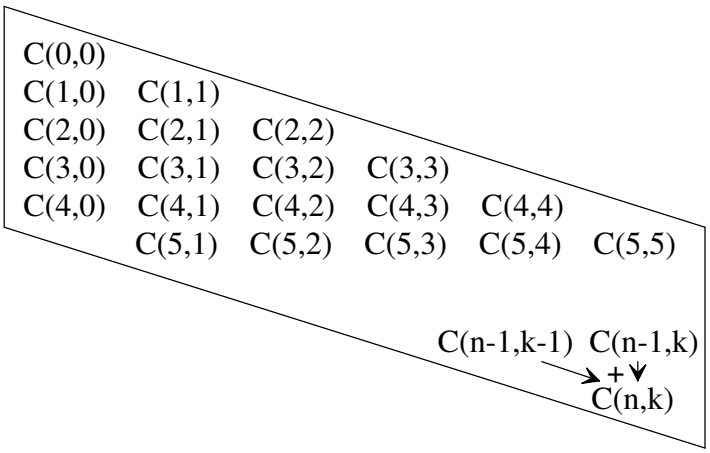
4. Dynamic-programming solutions trade-off space for storing smaller-problem solutions vs. improved execution time. What is the space-complexity of our above algorithm?

5. Often we can optimize (reduce) the amount of storage needed in dynamic programming by not storing solutions to all smaller problems, but just the ones we need again.
a) In binomial coefficient, to calculate the next row of solutions what smaller solutions do we need?

b) How would you modify the previous algorithm to reduce the space-complexity?

c) What is the space-complexity of the modified algorithm?

6. Some dynamic-programming algorithms do extra work calculating solutions to smaller-size problems which are not needed. For example, the "shape" of smaller-size problems needed for solving C(n, k) is:



Row #
0
1
2
3
4
5
.
.
.
n-1
n

Memoization is a dynamic-programming technique which uses the "top-down" recursive definition to discover only the needed smaller problems, but avoids redundant calculations by solving smaller-problems once and storing their solutions. Before recalculating a smaller problem, it checks to see if its solution is already stored. Recall the recursive definition of the Fibonacci sequence (0, 1, 1, 2, 3, 5, 8, 13, ...) is:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \quad \text{for } n \geq 2$$

```
int fib( int n ) {
    // array to store solutions
    int fibonacci[n+1];

    // fill base cases
    fibonacci[0] = 0;
    fibonacci[1] = 1;
    // Use -1 for unknown solutions
    for (i=2; i <= n; i++) {
       fibonacci[i] = -1;
    } // end for

    return fib_memo(n, fibonacci);

} // end fib
```

```
int fib_memo( int n, int fibonacci[] ) {

    if (fibonacci[n] == -1) {
       fibonacci[n] = fib_memo(n-1, fibonacci)
                    + fib_memo(n-2, fibonacci);
    } // end if

    return fibonacci[n];

} // end fib_memo
```

Using the memoization solution to fibonacci as a guide, write the memoization algorithm for binomial coefficient.