

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 ,” that is, 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 $C(52, 5)$.

A problem with using the above formula to compute binomial coefficients 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 $C(52, 5)$, 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 bigger than can fit into a 64-bit integer representation.

Fortunately, $C(52, 5)$ can also be defined recursively by splitting the problem into two smaller subproblems. Consider card hands containing a specific card, say the ace of clubs, and those that do not. For 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 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, $C(52, 5) = C(51, 4) + C(51, 5)$. This point of view leads to the following recurrence for computing the binomial coefficient $C(n, k)$:

$$\begin{aligned} C(n, 0) &= 1 \\ C(n, k) &= 1, \text{ if } k = n \\ C(n, k) &= C(n-1, k) + C(n-1, k-1), \text{ if } 0 < k < n \end{aligned}$$

Given this recurrence, we can write the following recursive function to compute $C(n, k)$.

```
int C(int n, int k) { // Assume 0 <= k <= n      1
    if (k == 0 || k == n)                        2
        return 1;                               3
    else return C(n-1, k) + C(n-1, k-1);         4
}                                                 5
```

However, this divide-and-conquer approach to computing binomial coefficients is very slow due to redundant calculations performed by the recursive calls, similar to the slow, divide-and-conquer approach to computing Fibonacci numbers.

This programming assignment has two parts. First, write a C program that uses dynamic programming, instead of divide-and-conquer, to compute $C(n, k)$. Second, write a C program that uses memoization, which is a special form of dynamic programming, to compute $C(n, k)$.

The basic dynamic programming approach is “bottom-up.” It solves subproblems in order from smallest to largest until the original problem is solved. Memoization is a dynamic-programming technique that uses the “top-down” recursive definition to solve only the needed smaller problems, and avoids redundant calculations by solving smaller-problems once and storing their solutions.

The code below uses memoization to compute Fibonacci numbers. Before solving a smaller problem, it checks to see if its solution is already stored.

```

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

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;
    }
    return fib_memo(n, fibonacci);
}

```

Using this code as a guide, implement an algorithm that uses memoization to compute binomial coefficients.

What to turn in

Turn in a listing of your code for computing binomial coefficients using both bottom-up dynamic programming and memoization. Your code should be written in C or C++. In addition, turn in a demonstration that your code works correctly, and an experimental comparison of the efficiency of the divide-and-conquer, bottom-up dynamic programming, and memoization algorithms. Does the memoization algorithm solve fewer subproblems than the bottom-up dynamic programming algorithm? Briefly explain.