Data Structures (CS 1520) Lab 6 Name:__________________

Objective: To gain experience with a simple, recursive, divide-and-conquer implementation and how to develop faster solutions using an iterative dynamic programming implementation

To start the lab: Download and unzip the lab6.zip file from eLearning.

Part A: We've seen several recursive "divide-and-conquer" algorithms: fibonacci and coin-change problem. The general idea of divide-and-conquer algorithms is:

- dividing the original problem it into small problem(s) (e.g., fib(n-1) and fib(n-2))
- solving the smaller problem(s) recursively
- combine the solution(s) to smaller problem(s) to solve the original problem (e.g., return fib(n-1) + fib(n-2))

Mathematics has several simple recursively defined functions. For example, the *factorial* function can be recursively defined as:

(1) $n! = n * (n - 1)!$ for $n \geq 1$, and
$0! = 1$

Implement a recursive `factorial(n)` function using this recursive definition and test it with several small examples (e.g., 3! = 3*2! = 3*2*1! = 3*2*1*0! = 3*2*1*1 = 6, and 5! = 5*4*3*2*1*1 = 120).

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

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

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, 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! = 2,598,960.

Implement the function C(n, k) using formula (2) above and your recursive factorial function.

Part B: One problem with using formula (2) in most languages is that n! grows very fast and overflows the integer representation before you can do the division to bring the value back to a value that can be represented. (NOTE: Python does not suffer from this problem, but lets pretend that it does.)
For example, when calculating 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

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.

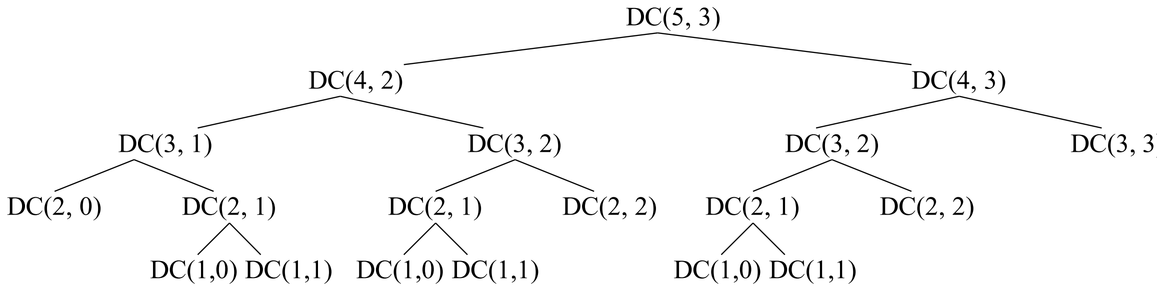
For those hands that do contain the ace of clubs, we need to choose 4 more cards from the remaining 51 cards, i.e., $C(51, 4)$. For those hands that do not contain the ace of clubs, we need to choose 5 cards from the remaining 51 cards, i.e., $C(51, 5)$. Therefore, $C(52, 5) = C(51, 4) + C(51, 5)$.

In general, **(NOTE: When implementing your recursive code, be sure to use DC for the recursive calls)**

(3) $C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$ for $1 \leq k \leq (n - 1)$, and
$C(n, k) = 1$ for $k = 0$ or $k = n$

Implement the recursive "divide-and-conquer" binomial coefficient function using equation (3). 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:



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.

```
                    1                                   Row #
                 1     1                                  0
              1     2     1                               1
           1     3     3     1                            2
        1     4     6     4     1                         3
     1     5    10    10     5     1                      4
                    .                                     5
                    .
```

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:

									Row #
C(0,0)									0
C(1,0)	C(1,1)								1
C(2,0)	C(2,1)	C(2,2)							2
C(3,0)	C(3,1)	C(3,2)	C(3,3)						3
C(4,0)	C(4,1)	C(4,2)	C(4,3)	C(4,4)					4
C(5,0)	C(5,1)	C(5,2)	C(5,3)	C(5,4)	C(5,5)				5
⋮									⋮
			C(n-1,k-1)	C(n-1,k)					n-1
				+					
C(n,0)	C(n,1)	C(n,2)	. . .	C(n,k)			C(n, n-1)	C(n,n)	n

For Part C, your job is to implement the "dynamic programming" binomial coefficient function using a Python list of lists to store Pascal's triangle (e.g., `pascalsTriangle[row #][col #]`) and loops (no recursion needed). Call your function `DP(n,k)` for "dynamic programming". **Hints for Part C:**

- Review the dynamic programming fibonacci example from Lecture 9. File `lab6/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`, reduces the amount of memory used by only storing the answers for the previous two smaller problems.

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).

<u>**Part D**</u>**: EXTRA CREDIT** - Your function `DP_Extra_Credit(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.

After you have correct code for all parts of the lab, submit a lab6.zip containing your code on eLearning.

If you do not get done today, then submit it by next week's lab period.