Objective: To observe that a faster algorithm is a better solution than tweaking a slow one. To gain experience with a simple, recursive, divide-and-conquer implementation and how to develop faster solutions using an iterative dynamic programming implementation and a recursive memoization implementation.

We’ve seen several recursive “divide-and-conquer” algorithms: binary search, merge sort, quick sort. The general idea of divide-and-conquer algorithms is:

- dividing the original problem into small problem(s) (e.g., merge sort - split into left half and right half)
- solving the smaller problem(s) (e.g., merge sort recursively solved each half)
- combine the solution(s) to smaller problem(s) to solve the original problem (e.g., merge sort - merged the two half back together to solve the original sorting problem)

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

\[ n! = n \times (n - 1)! \quad \text{for } n \geq 1, \text{ and } 0! = 1 \]

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). One problem with using the above formula in most languages is that n! grows very fast and overflows an integer representation. For example, 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, Python does not suffer from this problem, since it automatically switches to unlimited Long integers for really big values like this. Use your function to calculate 52!.

Another simple recursive definition is the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, .... Where each number in the sequence is the sum of the previous two numbers. A recursive definition to calculate the \(n\)th number in the Fibonacci sequence is: (Notice that we start numbering the position in the Fibonacci sequence from 0)

\[
\begin{align*}
\text{Fib}(n) &= \text{Fib}(n-1) + \text{Fib}(n-2) \quad \text{for } n > 1, \text{ and } \\
\text{Fib}(1) &= 1 \quad \\
\text{Fib}(0) &= 0
\end{align*}
\]

a) Implement a recursive function \( \text{Fib}(n) \) to calculate the \( n \)th number in the Fibonacci sequence. Test is with small values like \( \text{Fib}(8) \) which is 21, and \( \text{Fib}(10) \) which is 55.

Use a simple main function as shown below to print the first 50 values of the fibonacci sequence.

```python
def main():
    for n in range(50):
        print "The", n, "th fibonacci value is", Fib(n)
```

b) What do you observe about the time it takes for each \( \text{Fib}(\ ) \) call to complete?

c) Why do you thing that the recursive fibonacci is so slow?
Part B. Complete the following recursion "calling tree" for Fib(6).

\[
\text{Fib(6)} \quad \rightarrow \quad \text{Fib(5)} \quad \text{Fib(4)}
\]

a) Approximately how much more work is performed by calculating Fib(n+1) than Fib(n) for some large n?

b) What would you guess would be the “big-oh” notation for your recursive Fib(n) function? $O(\quad )$

c) Much of the slowness of your “divide-and-conquer” Fibonacci function, Fib(n), is due to redundant calculations performed due to the recursive calls. How else might we calculate Fibonacci without constantly recalculating the result of smaller problem instances?
Part C. What you hopefully figured out from Part B is a VERY POWERFUL concept in Computer Science --
dynamic programming. 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 them. In Python we could use a list to store the answers to smaller
problems of the Fibonacci sequence (in most programming languages you would use an array).

To use dynamic programming when you have a recursive statement of the problem. You can do the following
steps:
1) Store the solution to smallest problems, i.e., the base cases
2) Loop (no recursion needed) from the base cases up to the biggest problem of interest. On each iteration of the
loop we:
   • solve the next bigger problem
   • store its result for later use so we never have to recalculate it

We’ll reimplement Fib(n) that calculates the n\textsuperscript{th} number in the Fibonacci sequence use dynamic programming.
Recall the recursive definition of the Fibonacci sequence:

\[
\begin{align*}
Fib(n) &= Fib(n-1) + Fib(n-2) \quad \text{for } n > 1, \text{ and} \\
Fib(1) &= 1 \\
Fib(0) &= 0
\end{align*}
\]

a) We’ll use a list called, fibonacci, to store the solution to the “smaller” problems. Complete the dynamic
programming code:

```python
# Dynamic programming solution to determine the nth number in the
# Fibonacci sequence.
def Fib(n):
    # Step 1: Store base case solutions
    fibonacci = [
        # Step 2: Loop from base case to biggest problem of interest
        for next in range(n):
            # return nth number in the Fibonacci sequence
            return

b) Rerun the simple main function to print the first 50 values of the fibonacci sequence using your dynamic
programming solution.

```
c) One tradeoff of simple dynamic programming implementations is that they can require more memory since we store solutions to all smaller problems. Often, we can reduce the amount of storage needed if the next larger problem (and all the larger problems) don’t really need the solution to the really small problems, but just the larger of the smaller problems. In fibonacci for example, when calculating the next value in the sequence we only need the previous two solution. Reimplement the Fib(n) function a third time to eliminate the list fibonacci and replace it with three variables.

**Part D.** Sometimes a solution to the (“largest”) problem of interest does not require the solution to all of the smaller problems. We’d like to calculate only the solutions to smaller problems that we actually need for the problem of interest. One problem is predicting which smaller problems will be needed. One solution to this is memoization where you write a recursive solution like divide-and-conquer because the recursive solution starts with the largest problem and only calls smaller problems that are actually needed to solve it. To avoid the divide-and-conquer problem of recalculating smaller problems multiple times, we’ll check if we previously calculated a solution to a smaller problem before doing the recursive call.

The general steps for using memoization are:
1) initialize a list (array) of all smaller problem solutions to some dummy value like -1 to indicate that these problems have not yet been solved.
2) in this list set the base case solutions
3) write a recursive version like divide-and-conquer, except before doing the recursive call(s) check if your list already has an answer. If it does, use the list answer instead.

```python
def fib(n):
    # Step 1 and 2: initialize solution list and set base cases
    fibonacci = []
    fibonacci.append(0)
    fibonacci.append(1)
    for i in range(2, n+1):
        fibonacci.append(-1)

    # Step 3:
    fib_helper(n, fibonacci)

    return fibonacci[n]

def fib_helper(n, fibonacci):
    Write fib_helper recursive function.
```