Problem Description:

The Fibonacci sequence can be thought of as a series of numbers in which each number F_n is the sum of its two predecessors, for all $n > 2$.

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & n > 2 \\ 1 & otherwise \end{cases}$$

From a computational standpoint, the recurrence relation $F_{n-1} + F_{n-2}$ serves as the recursive case , while the initial condition $n > 2$ serves as the base case.

Dynamic Programming Implementation:

Dynamic Programming is a programming paradigm whose namesake is given by the varying time complexity of its algorithms, typically starting out inefficient at speeds of O(n^2) or worse, but saving time as it handles inputs it has previously calculated. Our program is initially no faster than the iterative or recursive approach, until it uses values from the fibonacci sequence it has already documented, at which point it takes constant time.

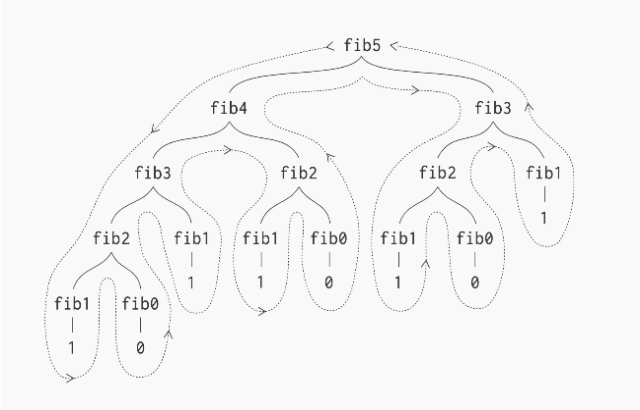
Recursive Implementation:

The recursive implementation of the Fibonacci sequence, without optimization, is quite expensive. For any Fibonacci number F_n, there are approximately 2^n recursive calls. [1]

1. Base Case ($n \leq 2$): There are no recursive calls.
2. Recursive Case ($n > 2$): The algorithm will make two recursive calls.
    a. $F(n-1)$
    b. $F(n-2)$

It will follow this pattern all the way to the base case, performing redundant calculations along the way. To visualize, we can use a recursive tree, where at each level k, there are 2^k nodes, each representing a recursive call.

[1] The total number of recursive calls being $2n-1$, where we account for the initial call.



So, as our input n increases, the time the algorithm takes to execute increases exponentially, resulting in a time-complexity of $O(2^n)$.

Space-Time Trade-off:

The concept of space-time trade-off refers to a situation in which an algorithm may increase storage space to decrease its time-complexity. An example of this tradeoff would be our implementation of the fibonacci sequence that utilizes memoization. While the extra storage of previous values increases the space complexity, it results in a much faster run time. This is useful when extra storage is available. However, applications that require efficient memory management may opt for slower runtimes to decrease their space usage.

Performance Analysis:

See pictures below for visuals detailing the speed of each of our various implementations.

Conclusion:

As expected, the optimized recursive approach significantly improves time complexity by sacrificing O(n) space. With a growing amount of required function calls as N increases, the naive recursive algorithm recalculates values of fibonacci thousands of times over and slows down before finally reaching a solution. However, this approach still has its applications, as in embedded systems like satellites, storage space is a crucial resource so the dynamic approach may not be an acceptable solution. Naturally though, these optimized approaches are preferred whenever the space tradeoff is negligible to improve overall time complexity.

Run Time Visualizations

```
Recursive (n = 1): 90ns
Recursive (n = 2): 60ns
Recursive (n = 3): 90ns
Recursive (n = 4): 80ns
Recursive (n = 5): 90ns
Recursive (n = 6): 120ns
Recursive (n = 7): 220ns
Recursive (n = 8): 230ns
Recursive (n = 9): 570ns
Recursive (n = 10): 770ns
Recursive (n = 11): 1060ns
Recursive (n = 12): 1310ns
Recursive (n = 13): 3930ns
Recursive (n = 14): 2960ns
Recursive (n = 15): 6060ns
Recursive (n = 16): 3430ns
Recursive (n = 17): 5400ns
Recursive (n = 18): 8640ns
Recursive (n = 19): 13960ns
PS C:\Users\corey\Desktop\group-fibonacci>
```

```
Iterative (n = 5): 180ns
Iterative (n = 10): 490ns
Iterative (n = 15): 310ns
Iterative (n = 20): 525ns
Iterative (n = 25): 770ns
Iterative (n = 30): 905ns
Iterative (n = 35): 1000ns
Iterative (n = 40): 1185ns
Iterative (n = 45): 1330ns
Iterative (n = 50): 3300ns
Iterative (n = 55): 1445ns
Iterative (n = 60): 1750ns
Iterative (n = 65): 1695ns
Iterative (n = 70): 1690ns
Iterative (n = 75): 1995ns
Iterative (n = 80): 2195ns
Iterative (n = 85): 2240ns
Iterative (n = 90): 2610ns
Iterative (n = 95): 4250ns
```

```
Optimized Recursion (n = 1): 170ns
Optimized Recursion (n = 2): 430ns
Optimized Recursion (n = 3): 330ns
Optimized Recursion (n = 4): 380ns
Optimized Recursion (n = 5): 350ns
Optimized Recursion (n = 6): 320ns
Optimized Recursion (n = 7): 360ns
Optimized Recursion (n = 8): 350ns
Optimized Recursion (n = 9): 480ns
Optimized Recursion (n = 10): 780ns
Optimized Recursion (n = 11): 570ns
Optimized Recursion (n = 12): 840ns
Optimized Recursion (n = 13): 810ns
Optimized Recursion (n = 14): 1040ns
Optimized Recursion (n = 15): 890ns
Optimized Recursion (n = 16): 1060ns
Optimized Recursion (n = 17): 920ns
Optimized Recursion (n = 18): 1050ns
Optimized Recursion (n = 19): 1420ns
```