**Directions for Fibonacci Calculator:**

1. Select the method you would like to use.

2. Enter the number you would like to calculate.

3. Click the calculate button. (Enter will not work.)

**Printing the Results to the screen, local printer, and/or e-mailing the results:**

1. After following the initial directions, you can click "Print" on the menu at the top left corner of the program.

2. Select "Print Report".

3. After the report is printed to the screen, you have the option to print the results to a printer, e-mail the results, or exit the report screen.

**Exiting the program:**

If you would like to exit the program, you can click the red "X" at the top-right corner of the screen **OR** click "File" and select "Exit".

**GENERAL NOTES:**

For all methods used, I continually clicked the "Calculate" button until I saw the lowest possible number that was reported. I did this for each number and each method in order to insure that I was able to get the lowest ***nanoTime*** result. For the recursive method, I did not use a number higher than 41 on my results, as anything higher would cause the system to hang for longer periods of time as the input increased. Although I could have converted the ***nanoTime*** results to milliseconds or seconds, I decided against it in order to show the magnitude of using large numbers using the recursive method. I used an online graphing tool that gave me an estimated equation for each method.  
Link:

**RESULTS:**

***Method: Recursion - r(N) – Exponential Complexity***

|  |  |
| --- | --- |
| **RESULTS:** | |
| Fib(10) | 2,913 ns |
| Fib(20) | 56,506 ns |
| Fib(30) | 6,642,740 ns |
| Fib(40) | 822,664,832 ns |
| Fib(41) | 1,323,286,114 ns |

Overall, this method is the least efficient of the three methods used in order to calculate the results. Even when small numbers were used as inputs, the minimum time calculated by nanoTime was 2,913 ns. This time grew exponentially as larger numbers were used as inputs. This is due to the fact that the CPU must unfold each number and develop a larger tree each time a larger number is used as the input.

***Method: Loop - l(N) - Linear Complexity***

|  |  |
| --- | --- |
| **RESULTS:** | |
| Fib(10) | 283 ns |
| Fib(20) | 582 ns |
| Fib(30) | 583 ns |
| Fib(40) | 583 ns |
| Fib(1000) | 1,457 ns |
| Fib(2000) | 46,312 ns |

Initially, calculating the results was much more efficient than any other method. However, as larger numbers were used as inputs, this method became less efficient. However, this method was still much more efficient than the recursive method.

***Method: Binet's Formula – b(N) – Constant Complexity***

|  |  |
| --- | --- |
| **RESULTS:** | |
| Fib(10) | 1,165 ns |
| Fib(20) | 1,165 ns |
| Fib(30) | 1,165 ns |
| Fib(40) | 1,165 ns |
| Fib(1000) | 1,165 ns |
| Fib(2000) | 4,951 ns |

Using smaller inputs, Binet’s Formula was not as efficient as the loop method. As larger numbers were used, it was clear that this was the most efficient methods when dealing with very large numbers.

**Order of methods from most efficient to least efficient (given large inputs):**

1. Binet's Formula - b(N)
2. Loop - l(N)
3. Recursive - r(N)

***BIG O:***

O(l(N)) = Ob((N)) = O(r(N))

**FINAL THOUGHTS:**

Initially, I had assumed that the loop method was the best method. However, I decided to use a much larger number in order to finalize my conclusion. I was incorrect in my initial assumptions. Luckily, I decided to check my work a third time as I did not used Fib(1000) nor Fib(2000) the first two times.