GRADING SUMMARY

Name: Niyousha Raeesinejad

ID: 30038699

PROGRAM:		/20
REPORT:		/20
TOTAL		**/36**

PROGRAM

Program builds correctly	Yes	No
Runs: completes with zero run-time errors	Yes	No

IMPORTANT: Source code that does not compile or produces run-time errors will receive 0%

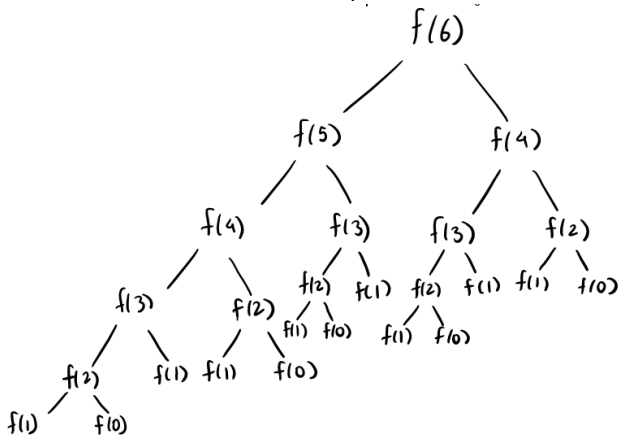
Criteria	TOTAL
Functionality: produces correct output	
Readability: code is clear and easy to read	
Modularity: code is easy to extend, or use	
Generality: easy to extend to more sophisticated applications	
Documented: code classes/functions/inline commented	
TOTAL:	**/20**

REPORT

Criteria	TOTAL
Plots: show important trends in data	/4
Question #1/#2/#3:	/4
Question #4/#5/#6:	/4
Question #7:	/4
Question #8:	/4
TOTAL:	**/20**

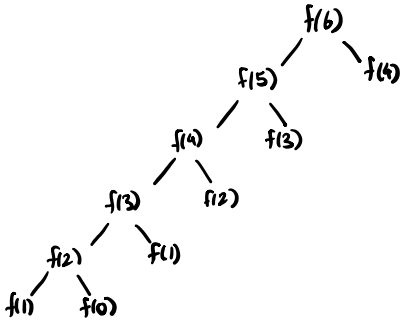
Questions

1. Concerning algorithm 1:
    a. Does it perform redundant calculations (yes or no)?
    Yes, algorithm 1 performs redundant calculations by re-computing intermediate results.
    b. Draw its recursion tree for the computation of F_6



    c. How many times are F_2, F_3, … evaluated (recursion called) in the computation of F_6?
    F_0 is evaluated 5 times, F_1 is evaluated 8 times, F_2 is evaluated 5 times, F_3 is evaluated 3 times, F_4 is evaluated 2 times, and F_5 and F_6 are evaluated once.

2. Concerning algorithm 2:
    a. Does it perform redundant calculations (yes or no)?
    No, algorithm 2 does not perform redundant calculations due to storing intermediate results in an array instead of recomputing them.
    b. Draw its recursion tree for the computation of F_6



    c. How many times are F_2, F_3, … evaluated (recursion called) in the computation of F_6?
    Each Fibonacci number (F_0 – F_6) is evaluated once.

3. Cost-benefit comparison of algorithm 1 and algorithm 2 based on Q1 and Q2.
    Algorithm 1 unnecessarily computes intermediate results of the Fibonacci series regardless of having calculated them in previous recursion calls, while algorithm 2 keeps track of repeated results in an array serving as a cache. Since algorithm 1 requires more steps and memory to calculate and store each result on the stack, it is less efficient than algorithm 2.

4. What is the big-O running time for algorithm 3? Why?
    $f(n) = c_1(n+2) + c_2(1), + c_3(1) + c_4(1) + c_5(n+1) + c_6(n) + c_7(n) + c_8(1)$

$f(n) = (1 + 2 + 5 + 2) \times n + (2 \times 1 + 2 + 2 + 1 + 1 + 2)$
$f(n) = 10n + 10$ → Algorithm 3 is proportional to $10n$ with a linear growth rate.
$f(n) \leq c \times g(n)$
$10n + 10 \leq 20 \times g(n)$
$10n + 10 \leq 20n$
Since this is true for $c = 20$ when $n \geq n_0 = 1$, $f(n) = 10n + n$ is $O(n)$.

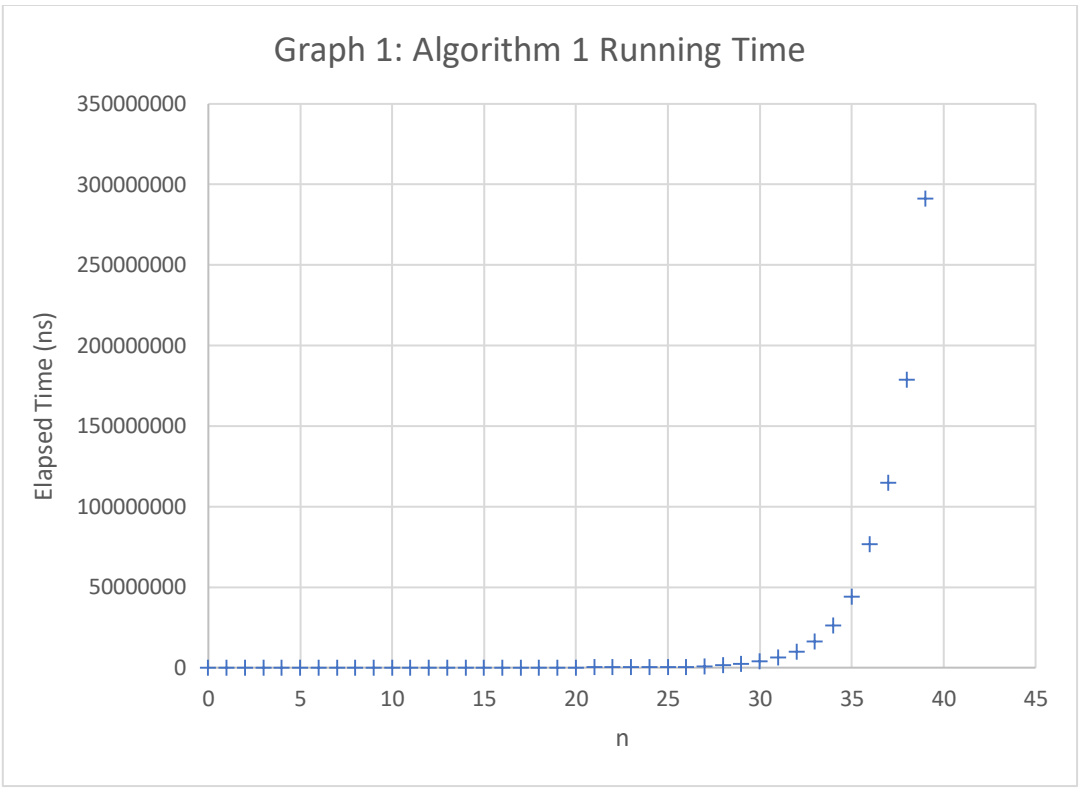
5. What is the big-O running time for algorithm 4? Why?
$f(n) = c_1(1) + c_2(1) + c_3(1) + c_4(1) + c_5(n) + c_6(n-1) + c_7(n-1) + c_8(n-1) + c_9(n-1) + c_{10}(1)$
$f(n) = (1 + 2 + 1 + 1 + 2) \times n + (1 + 1 + 1 + 1 - 2 - 1 - 1 - 2 + 1)$
$f(n) = 7n - 1$ → Algorithm 4 is proportional to $7n$ with a linear growth rate.
$f(n) \leq c \times g(n)$
$7n - 1 \leq 7 \times g(n)$
$7n - 1 \leq 7n$
Since this is true for $c = 7$ when $n \geq n_0 = 1$, $f(n) = 7n - 1$ is $O(n)$.

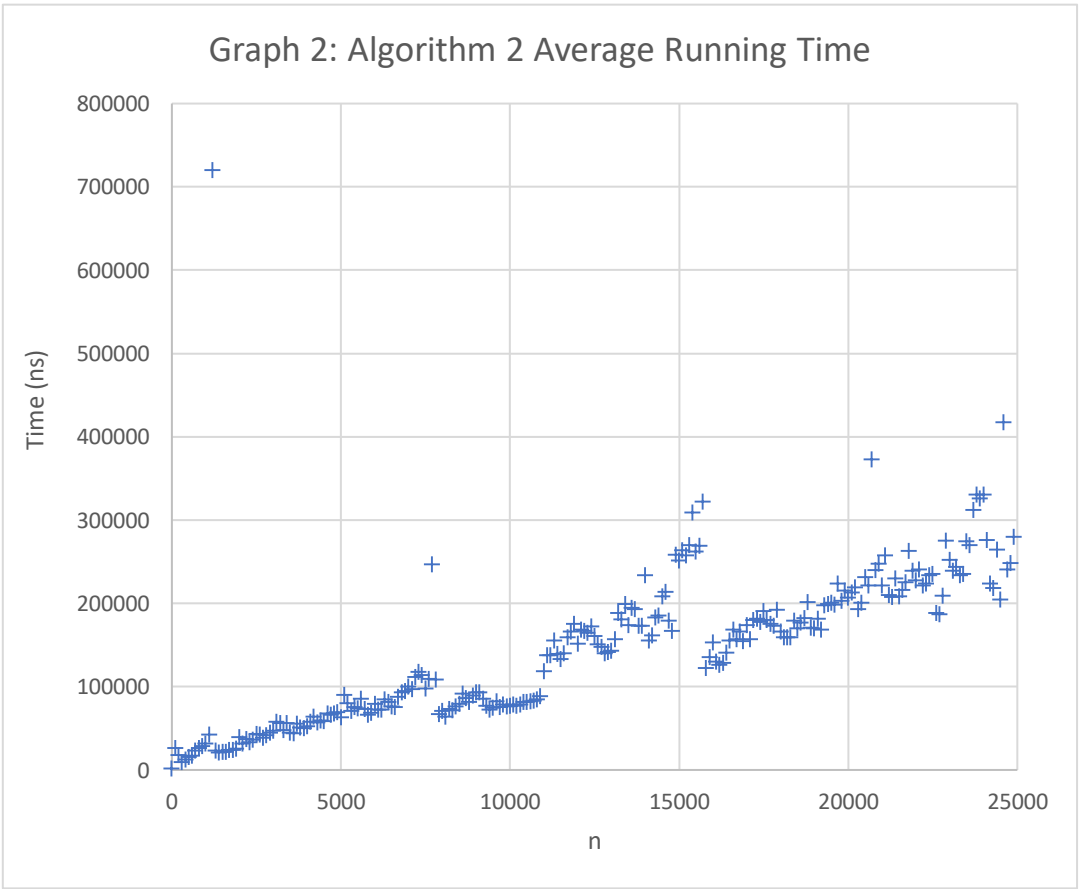
6. Cost-benefit comparison of algorithm 3 and algorithm 4 based on Q4 and Q5.
Algorithm 3 and algorithm 4 are very similar in terms of growth rate, both having a big-O running time of $0(n)$. Since algorithm 3 uses dynamic programming to store all subsequent values up to the nth Fibonacci number, it requires more memory space than algorithm 4, which merely keeps track of the previous two values in an iterative loop and does not need to store all of them in memory. Thus, algorithm 4 is slightly more efficient as it does not go through additional steps of storing data on the heap.

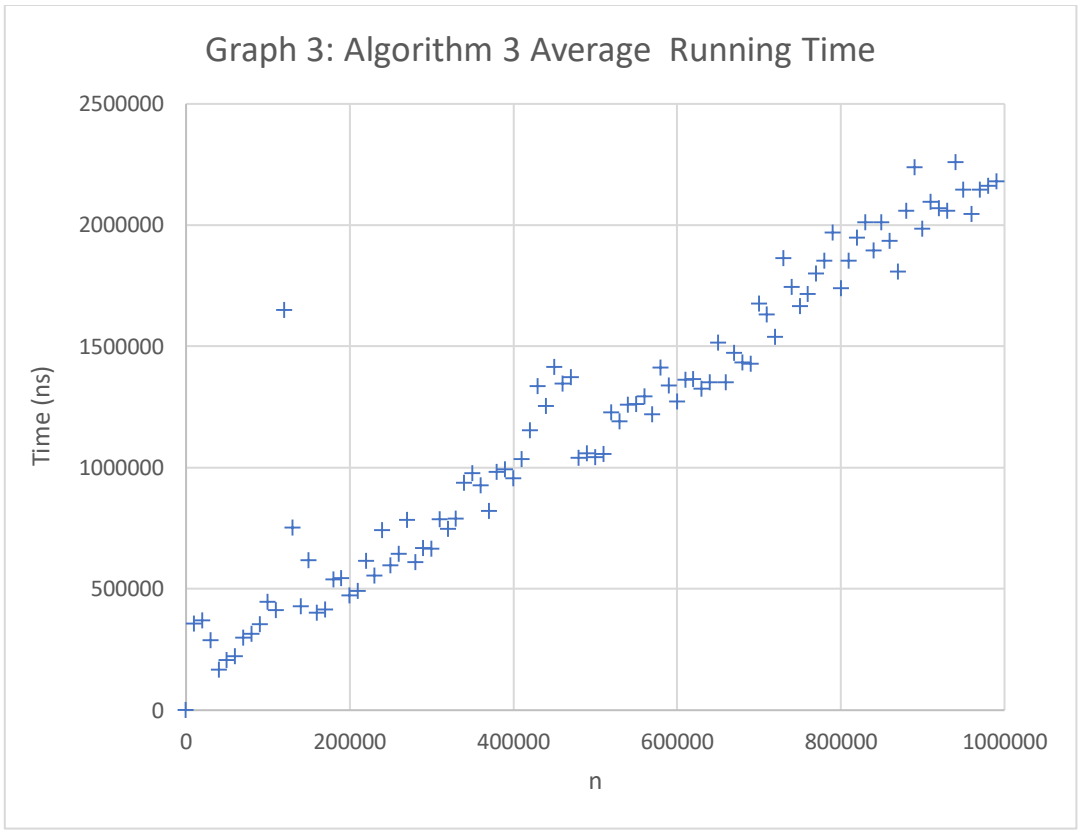
7. What is the big-O running time for algorithm 5? Why?
$T(n) = 108 + f\left(\frac{n}{2}\right)$
$f(n) = 108 + f\left(\frac{n}{2}\right) \rightarrow f(n) = 216 + f\left(\frac{n}{4}\right) \rightarrow\ f(n) = 324 + f\left(\frac{n}{8}\right) \rightarrow f(n) = 432 + f\left(\frac{n}{16}\right) \ldots$
$f(n) = 108k + f\left(\frac{n}{2^k}\right)$
$Base\ case$: $\frac{n}{2^k} = 1 \rightarrow k = \log_2 n$
$f(n) = 108 \log_2 n + 1$ → Algorithm 5 is proportional to $108\log_2 n$ with a logarithmic growth rate.
$f(n) \leq c \times g(n)$ → $108 \log_2 n + 1 \leq 109 \log_2 n$
Since this is true for $c = 108$ when $n \geq n_0 = 2$, $f(n) = 108 \log_2 n + 1$ is $O(\log_2 n)$.

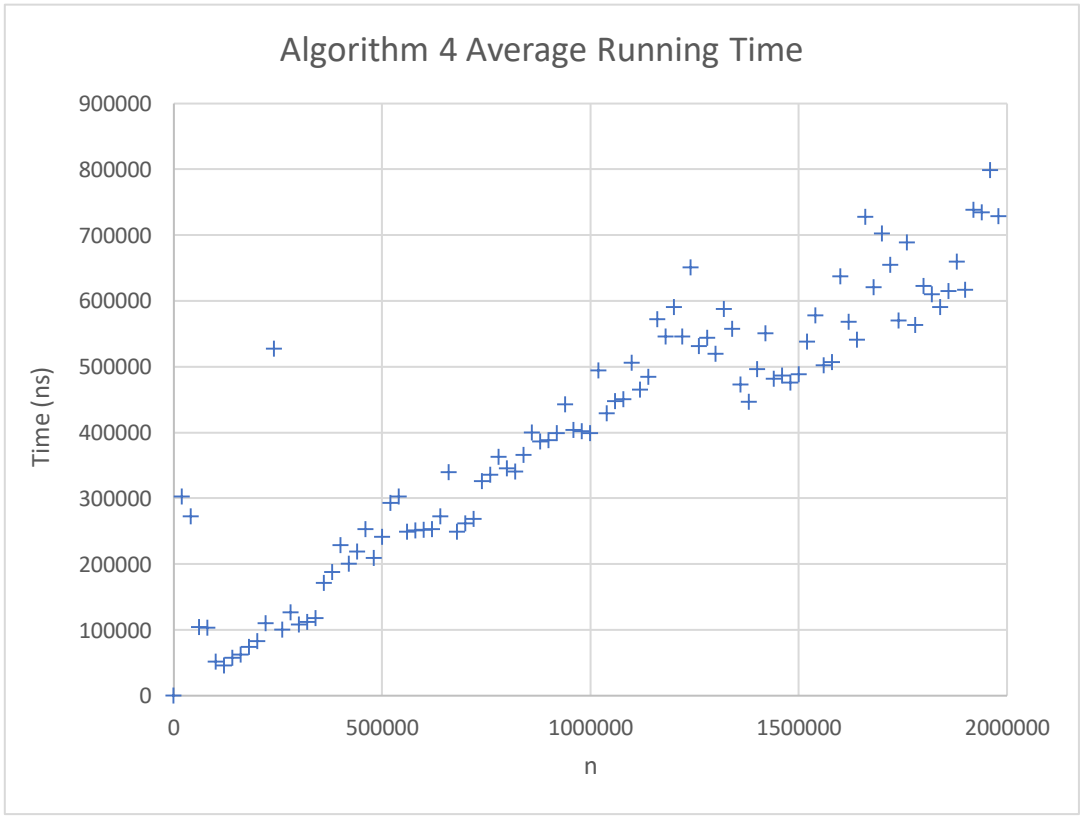
8. Under what situations would you use each of the algorithms? Answer your question according to the experimental studies results (i.e., after computing the timing execution). You don't have to repeat verbatim answers to Q3/Q6 here but should likely reference them in making your determinations.

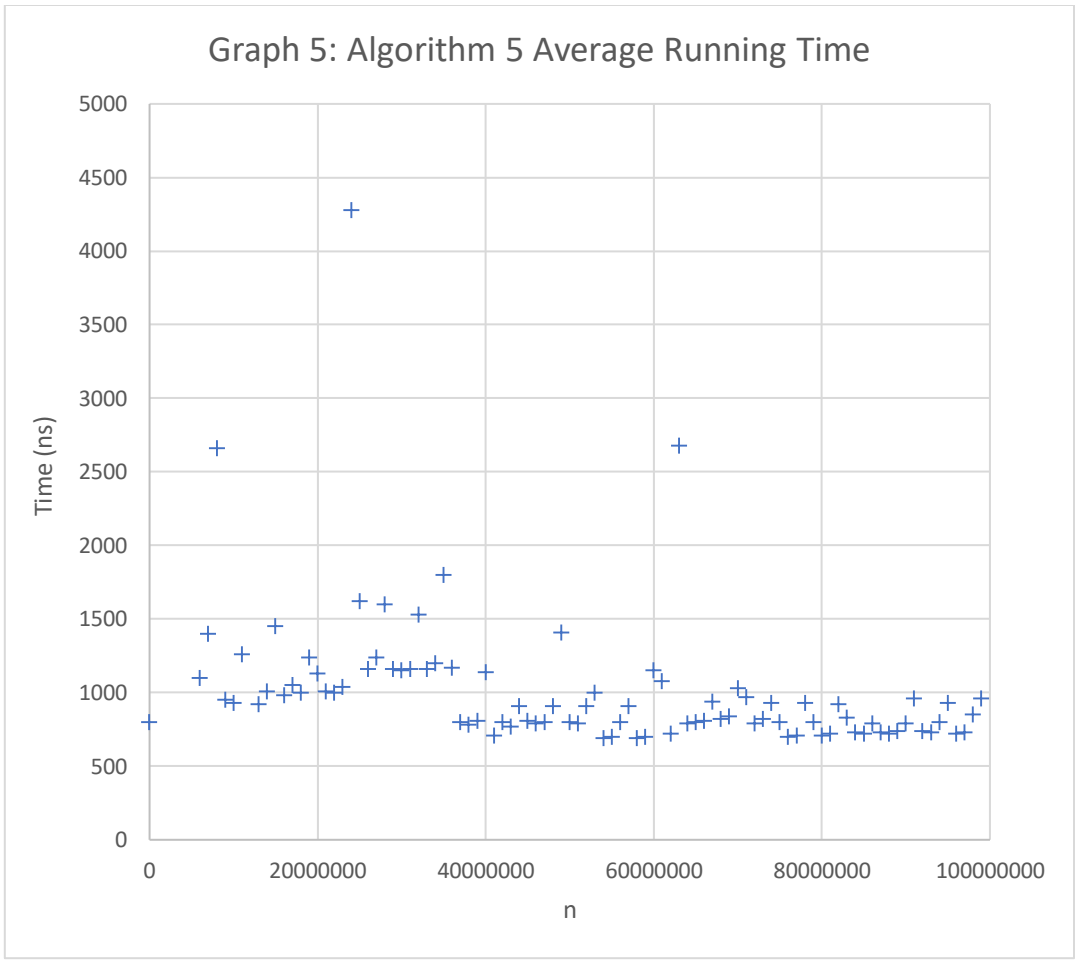
Algorithm 1 may be utilized for very simple computations due to its tendency to quickly take up unnecessary memory space and time, as shown by its exponential growth rate in Graph 1. Alternatively, algorithm 2 requires less time and intermediate computations to yield the same and even further results compared to algorithm 1, thus it may be utilized more in problems containing larger input sizes. Computing with a similar growth rate as algorithm 2 are algorithms 3 and 4 which also have big-O running times of $O(n)$. However, they are different in terms of storing subsequent values; algorithm 3 may be utilized when one might need to keep track of all previously computed values leading up to the result, such as in a dynamic array, while utilizing algorithm 4 entails only one output with disregard for subsequent computations whose results are replaced in the iterative loop. Finally, algorithm 5 demonstrates the best time execution with a big-O running time of $O(\log_2 n)$ as shown in Graph 5. Thus, the last algorithm may be used in situations involving very large input sizes.


Graph 1: Algorithm 1 Running Time
Elapsed Time (ns)
350000000
300000000
250000000
200000000
150000000
100000000
50000000
0
0
5
10
15
20
25
30
35
40
45
n


Graph 2: Algorithm 2 Average Running Time
Time (ns)
800000
700000
600000
500000
400000
300000
200000
100000
0
0
5000
10000
15000
20000
25000
n


Graph 3: Algorithm 3 Average Running Time
Time (ns)
2500000
2000000
1500000
1000000
500000
0
0
200000
400000
600000
800000
1000000
n


Algorithm 4 Average Running Time
Time (ns)
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
0
500000
1000000
1500000
2000000
n


Graph 5: Algorithm 5 Average Running Time
Time (ns)
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
20000000
40000000
60000000
80000000
100000000
n