## **GRADING SUMMARY**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Name:** | Moiz Abdullah | |  |  | | **ID:** | 30066638 | |  | |  |  |  | | --- | --- | --- | | **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** |

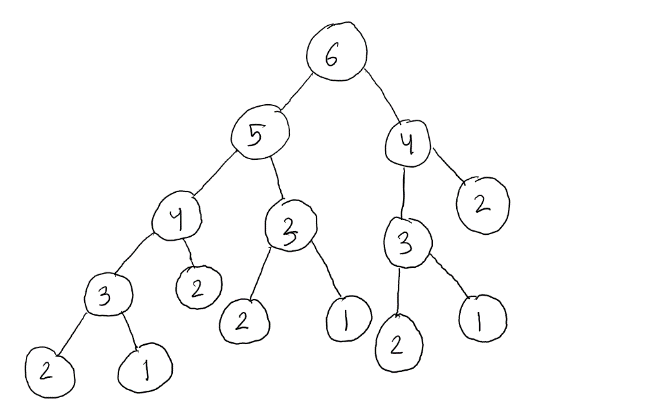
**REPORT**

|  |  |
| --- | --- |
| **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(a). Yes, there are many redundant calculations in the recursive method because same calculations are preformed multiple times.

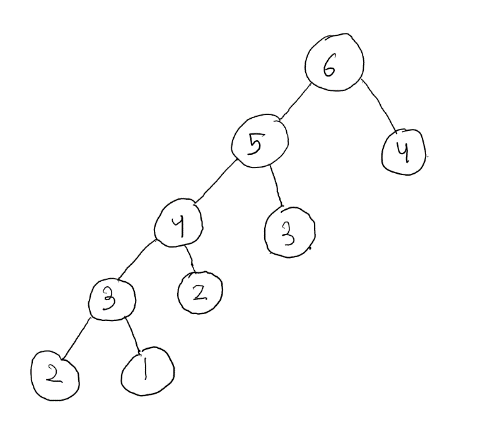
1(b). Recursion tree for the computation of F6:



1(c). F2 is evaluated 5 times and F3 evaluated 3 times in the computation of F6.

2(a). No redundant calculations are preformed in the memorization method because calculated values are stored in an array for later use.

2(b). Recursion tree for the computation of F6:



2(c). F2 is evaluated 2 times and F3 evaluated 2 times in the computation of F6.

3. Algorithm 2 calls less functions than algorithm 1, so algorithm uses less memory. Moreover, algorithm 2 preforms less computations than algorithm 1. Thus algorithm 2 is much better in terms of cost-benefit than algorithm 1.

4. According to the growth rate analysis of fibDyn (line 10 -10), the algorithm is linear and requires 8n + 9 times unit to solve a problem of size n. So, O(g(n)) = O(n), then f(n) <= c\*g(n) becomes 8n + 9 <= 17n, for c = 17, when n >= no = 1.

5. According to the growth rate analysis of fibIter (line 10 -10), the algorithm is linear and requires 7n + 7 times unit to solve a problem of size n. So, O(g(n)) = O(n), then f(n) <= c\*g(n) becomes 7n + 7 <= 14n, for c = 14, when n >= no = 1.

6. For small values of n, the growth rate of algorithm 3 (8n + 9) and algorithm 4 (7n + 7) intersect at -2 so algorithm 4 is faster than algorithm 3 for all positive values on n. Thus algorithm 4 is better in terms of cost-benefit for small numbers. But as the value of n approaches infinity, both the algorithms are proportional as the Big-Oh for both the algorithms is O(n). So, for very large values of n, both algorithms are similar in performance and cost.

7. The recurrence relationship (f(n)) is 1, if n = 0, 1 and 78 + f(n/2) if n > 1. Continuing for k times, we have f(n) = (78\*k) + f(n/2k) until we reach the base case, where we assume n/2k = 1, therefore n = 2k becomes k = log2n. Then we see that f(n) = (78\*k) + f(n/2k) = (78\*log2n) + f(1) = 78\*log2n + 1. So, the algorithm is logarithmic and requires 78\*log2n + 1 times unit to solve a problem of size n. So, O(g(n)) = O(log2n), then f(n) <= c\*g(n) becomes 78\*log2n + 1 <= 1log2n, for c = 79, when n >= no = 2.

8. Algorithm 1 is good for numbers under 25 but after that overall it is a very inefficient. Algorithm 2 is good for under 300 (as shown in graph 2) but progressively gets worse as the allocated memory space reaches its limit for quick access. Algorithm 3 is more efficient then 2, being able to maintain a quick speed for very long calculations, but falls under the same hardware limitations as 2, as the ram can only allocate memory so fast and holds a limited amount of memory. This is not a problem for quick calculations, but very large arrays can strain the memory enough to slow down the process. Algorithm 4 is very efficient for long term calculations but can be slower then 2 and 3 in short term. 5 is most efficient at long term calculations.

**Trends in Data**

Algorithm 1 vs Algorithm 2 (Graph [1])

Graph 1 shows the difference between algorithm 1 and 2. algorithm 1 (Series1) spikes up around 25. Whereas, algorithm 2 (Series2) looks linear compared to algorithm 1 and it is way more efficient.

Algorithm 3 vs Algorithm 4 (Graph [2])

Graph 2 shows the similarities between Algorithm 3 (Series1) and Algorithm 4 (Series2). They are both linear (shown in question 3 and 4), but Algorithm 4 is more efficient.

Graph for small values (Graph [3])

Graph 3 shows all five algorithms 1 to 5 (Series1 to 5 respectively). We see that algorithm 2, 3 and 4 are very similar for values smaller than 20. But algorithm 1 and 5 are inefficient for small values.

Graph for big values (Graph [4])

As we saw in graph 1 and 3 that algorithm 1 in inefficient. This is a plot of algorithm 2 to 5 (series1 to 4 respectively). We see that for big values (around 1000) algorithm 4 and 5 are the best.