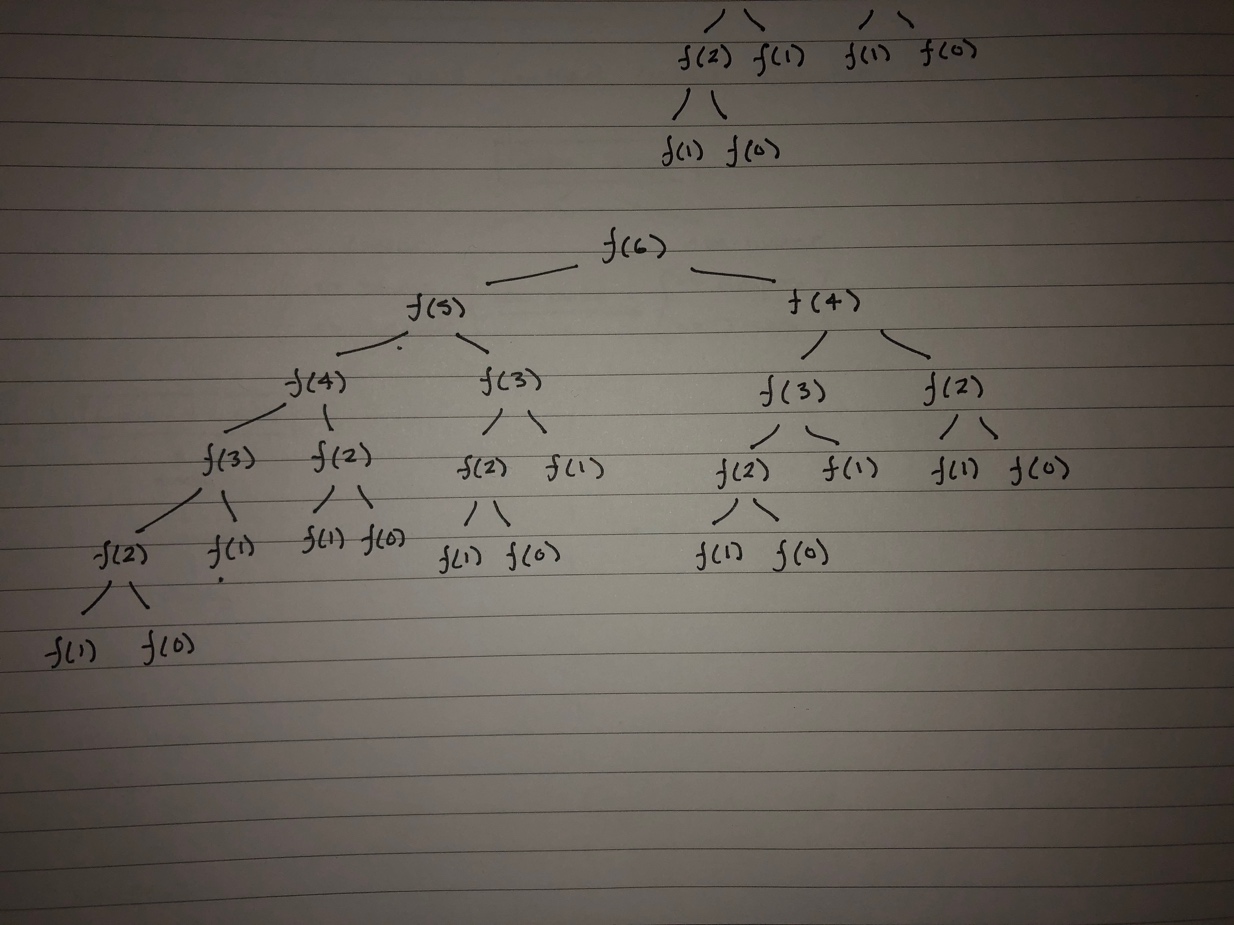
CPSC 319 Data Structures and Algorithms

Assingnment 1

Winter 2020

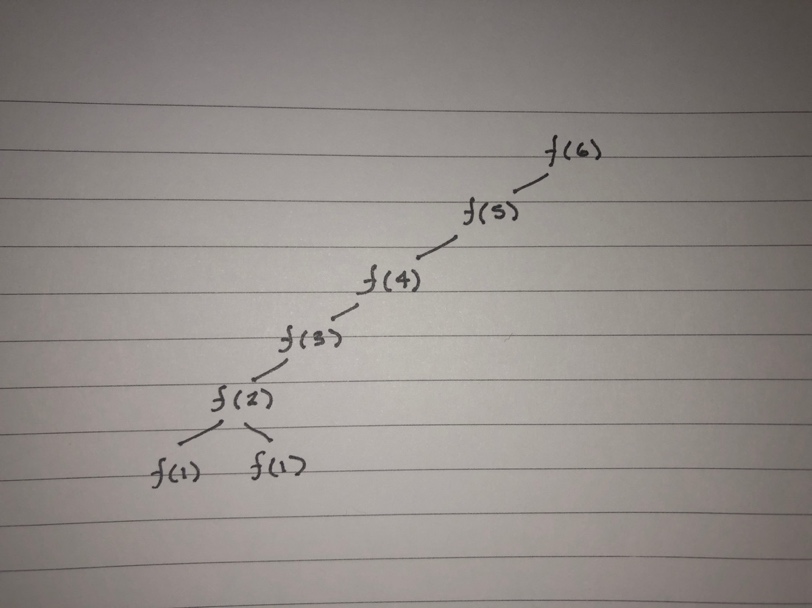
Question 1

1. Algorithm 1 performs redundant calculations. When recursing to calculate f(n), the algorithm often calculates f(n) after already have calculating the nth fibonacci number in a previous step. The algorithm may calculate the same value for f(n) multiple times.
2. 
3. - F(2): Calculated 5 times

* F(3): Calculated 3 times
* F(4): Calculated 2 times
* F(5): Calculated 1 time
* F(6): Calculated 1 time

Question 2

1. Algorithm 2 does not perform redundant calculations. When the function recurses, it checks if the value has already been calculated previously.

1. 

1. Each recursion is calculated only once, the values of f(2)-f(6) are all calculated only once.

Question 3

Algorithm 1 is very ineeficient for values of n between 0 and approximately 50. The algorithm is completely unviable for inputs greater than 50. Algorithm 2 begins to throw errors input sizes greater than approximately1000. Algorithm 2 has O(n) making it viable for inputs less than 1000. Based on the big-O running times of both algorithms, algorithm 2 is much more viable.

Question 4

Lines 1-6, 10 are all primitives, line 7 is a for-loop that executes n – 1 times since i starts at 2 ((i + 1 – 2)times). It has a cost of 1. The statement inside the for-loop has a cost of 2 and occurs n – 2 times (one time less than the for loop). From this we get f(n) = 7(1) + 1(n – 1) + 2(n – 2) = 3n + 2. F(n) is proportional to the dominant term 3n (linear) and therefore algorithm 3 is O(n) with c = 3 + 2 = 5 and n >= 1.

Question 5

Lines 1-5, 11 are all primitives, line 6 is a for-loop that executes n + 1 times (includes failure time). It has a cost of 1. The statements inside the for-loop (line 7) has a cost of 2 and occurs n times (one time less than the for loop). Lines 9 and 10 each have a cost of 1 and occur n times. From this we get: f(n) = 6(1) + 1(n + 1) + 2(n) + 2(n) = 5n + 7. F(n) is proportional to the dominant term 5n (linear) and therefore algorithm 4 is O(n) with c = 5 + 7 = 12 and n >= 1.

Question 6

Algorithms 3 and 4 both have big-O running times of O(n). Both running times grow lnearly with input size. Algorithm 3 requires allocated memory to create an array. This means that for very large inputs, the array may not fit in memory. Algorithm 3 initializes some local variables and iterates through them to compute the fibonacci number. The only potential problem is reaching a fibonacci number that will not fit in an int. It is possible to initialize the integers in a long or Integer instead which will allow storage of larger integers. Overall, Algorithm 3 occupies less space than algorithm 4. Since both algorithms are O(n), Algorithm 3 is a more viable procedure.

Question 7

To begin, algorithm 5 contains 4 primitives (lines 1 – 4,6). The call to matrix power is going to be recursive. Since the recursion is called with an argument size of n/2 each time, we must use the method of substitution to compute the number of times it is called. T(n) is 0 or 1 when n <= 1, and T(n/2) + c. First we will compute c by adding all primitives and funtion calls within function matrix power. The matrixmult method has primitives on line 1, lines 2-4 are three nested for-loops each executing 3 times with a cost of 1. Line 5 has a cost of 1 and an occurance of 8 times (2^3). Lines 10, 11 are nested for loops with cost 1 and occurance of 3 times each and line 12 has cost 1 and occurance 4. We get f(n) for method matrixmult is f(n) = 49. Now we know T(n) = T(n/2) + 1(55) now to get the expression for T(n/2) we substitute: n = n/2 -> T(n/4) + 2(55), n = n/2-> T(n/8) + 3(55) and so on. We observe a pattern and conclude that it holds: T(n/(2^k)) + 49k. The recursion will stop when k = logn. There fore we get T(n) = T(1) + 55log(n) = 55log(n) + 1. T(n) is proportoinal to 55log(n). Algorithm 5 is therefore O(logn) with c = 56 for n>=10.

Question 8

* Algorithm 1 is very ineffiecient, it performs redundant calculations and the runtime grows very rapidly. It is very inconvenient to use as past n = 49, the runtime was very long.
* Based on the data algorithm 2 is efficient since it has a big-O runtime of O(n). For large values of n (up to n = 1000), the algorithm’s runtime grows linearly making it a viable algorithm. The algorithm could not make an array of sizes n > 1000. This is most probably since storing an array of such size is difficult. The algorithm is therefore viable for input sizes less than approximatly n = 1000.
* Algorithm 3 is very efficient in most cases. Since the runtime grows linearly, it grows proportionally to the input size. For small values of n, the runtime is rapid, and for large values of n, the runtime is large, but proportional to the input size.
* Algorithm 4 is similar to algorithm 3. It is O(n) and so the runtime increases linearly as the input size increases. It is quite effiecient for both small and large values of n.
* Algorithm 5 has a big-O runtime of O(logn). This makes it extremely efficient for large values of n. For values of n less than approximately 1000, it is less effiecient than algorithms 2, 3, and 4. For large values of n, the runtime grows much slower than the other algorithms making it the best algorithm for large input sizes.

Figure 1

Figure 2

Figure 3