COMP352: Data Structure and Algorithms Observation Report

Author: Duc Nguyen

Part A

Pseudo Code:

No.1: Multiple Recursive Algorithm

Algorithm multipleTetra (k)

Input: a nonnegative integer k

Output: the result of summation of 4 numbers before the Tetranacci number index k

```
if (k <= 2) then
   return 0
else if (k <= 4) then
   return 1
else
   return (multipleTetra(k-1) + multipleTetra(k-2) + multipleTetra(k-3) +
          multipleTetra(k-4) )
```

No.2: Linear Recursive Algorithm

Algorithm linearTetra (k)

Input: a nonnegative integer k

Output: an array of Tetranacci numbers $(T_{k-3}, T_{k-2}, T_{k-1}, T_k)$

```
if (k <= 2) then
   return (0,0,0,0)
else if (k == 3) then
   return (0,0,0,1)
else
   (a, b, c, d) = linearTetra(k-1)
   temp = a + b + c + d
   return (b, c, d, temp)
```

Overall Observation:

The performance measured by Multiple Recursive Algorithm (MRA) is significantly slower than measured by Linear Recursive Algorithm (LRA). Moreover, the time complexity slope of MRA grows at a considerably faster rate than LRA's.The reason behind this low efficiency is because MRA used multiple recursive method, which resulted in an exponential time complexity. While LRA has linear time complexity.
At the point when the program tries to calculate Tetranacci (45), MRA takes nearly 41 minutes (2457324064655 ns) to calculate the result. Meanwhile, LRA only takes 31333 ns for the same task. Even if we acknowledge those results might be varied depends on the machine and the operating system in which the code ran, the execution time gap between two algorithms is still huge.

Part B

No.1 Multiple Recursive Algorithm:

The algorithm is of exponential complexity because each time it tries to calculate T_k, it has to call back $T_{k-4}, T_{k-3}, T_{k-2}, and T_{k-1}$.

> When the program finish calculating T_{k-4}, it continues to T_{k-3}, which requires T_{k-4} to be loaded.
>
> After finishing calculate T_{k-4} the second time and calculate T_{k-3}, it heads to T_{k-2}
>
> etc..

Summary: Each time the program executes, it will call 4 numbers before the requested index. When it finishes calculating one of these 4 numbers, the stack is cleared and does not save the result it calculated. Thus, when the program goes to the next element, it has to calculate the first number again. Resulting in four times complexity in each increase of the requested index. Consequently, the time complexity will be exponential $\mathbf{O}\ (2^n)$.

No.2 Linear Recursive Algorithm:

The algorithm is of linear. It has resolved the problem caused by the first algorithm by constantly saving the 4 values it calculated before and return it together. By this method, the program would not have to come back and calculate all the element one more time. Resulting in time complexity **O (n)**.

Part C

None of the 2 algorithms implemented in part A is tail recursion because tail recursion requires extra parameters to be implemented. However, tail-recursion

could be implemented easily since the problem could be solved by linear recursive algorithm.

Pseudo Code: Tail Recursive Algorithm

Algorithm tailTetra (k, $T_{k-3}, T_{k-2}, T_{k-1}, T_k$)

Input: a nonnegative index k, the value of $T_{k-3}, T_{k-2}, T_{k-1}, T_k$

Output: The value of T_k

```
if (k <= 2) then
   return 0
if (k == 3) then
   return d
total = a + b + c + d
return tailTetra(k-1, b, c, d, total)
```

Conclusion: The tail recursive method has a slightly better performance when comparing with the mentioned linear recursive algorithm. Even though tail recursion has the same time complexity as normal linear recursion, the reason makes it better might be the space complexity of the algorithm. Tail recursion only has space complexity of **O (1)**, while normal linear recursion has space complexity of **O (n)**