COMP 352 – Assignment 1

FOR INSTRUCTOR: DR. HAKIM MELLAH

GEORGE MAVROEIDIS (40065356)

DUE: MONDAY SEPTEMBER 30, 2019

Question 1:

a) The code and text files for part A can be found on the corresponding folder within the zip file.

Algorithm *arraySwitch*(A ,n)

Input: Array A of n integers

Output: Array A of n integers (possibly modified)

If n < 4

{return}

for **i ← 0** to **ceiling[$(n)/2$] EXCLUDED,** iterate **i = i+2** do

if **($i + 1 < ceiling[\frac{n}{2}]$** then

int temp ← A[i+1]

A[i+1] ← A[i]

A[i] ← temp

For **i ← ceiling[$(n-1)/2$]** to **n-1 EXCLUDED,** iterate **i = i+2** do

if **($i + 1 < n$)**

A[i+1] = A[i] * A[i+1]

return A

b) The loops and if statements inside them depend on the size of the array "n", therefore the complexity in terms of Big-O is **O(n)**. The loops are not nested and there are no recursive calls included, therefore, this confirms for no higher order terms.

c) The algorithm requires N units for A, space for n and the local variable i as well. Therefore, the space complexity depends on the size of the array. The space complexity, in terms of Big-O, is **O(n)**.

QUESTION 2 (ZOOM IN FOR BETTER VIEW):

Question 2:

A) $n^{22}\log n + n^7$ is $O(n^7 \log n)$?

$n^{22}\log n + n^7 \leq c\,g(n)$

$\rightarrow n^{22}\log n \leq cn^7\log n \quad \frac{n^{22}\log n}{n^7\log n} \leq c \quad n^{15} \leq c$, $\boxed{c=1,\ n_0=1}$

$\rightarrow n^7 \leq cn^7\log n \quad \frac{1}{\log n\ (\log_2 n)} \leq c$, $\boxed{c=2,\ n_0=2}$

Yes, it is $O(n^7 \log n)$ ✓

B) $10^7 n^5 + 5n^4 + 9000000\,n^2$ is $\Theta(n^3)$?

$\Omega(n^3) \qquad O(n^3)$

$\rightarrow c_1 n^3 \leq 10^7 n^5 \leq c_2 n^3 \quad \frac{c_1}{10^7} \leq n^2 \leq \frac{c_2}{10^7}$, $\boxed{c=10^7,\ n_0=1}$

$\rightarrow c_1 n^3 \leq 9000000\,n^2 \leq c_2 n^3 \quad \frac{c_1}{9000000} \leq \frac{1}{n} \leq \frac{c_2}{9000000}$ $\boxed{c=9000000,\ n_0=1}$

$\rightarrow c_1 n^3 \leq 5n^4 \leq c_2 n^3 \quad \frac{c_1}{5} \leq n \leq \frac{c_2}{5}$ $\boxed{c=5,\ n_0=1}$

Yes it is $\Theta(n^3)$ ✓

c) n^n is $\Omega(n!)$? Note: $n! = n(n-1)(n-2)\dots(2)(1)$, $1! = 1$

$cn! \geq n^n$ $\boxed{\begin{matrix} c=1 \\ n_0=1 \end{matrix}}$ Yes, it is $\Omega(n!)$ ✓

d.) $0.01n^9 + 800000n^2 + n$ is $\Theta(n^9)$?

$\rightarrow c_1 n^9 \leq 0.01n^9 \leq c_2 n^9$ ($c_1 \leq 0.01 \leq c_2$) $\boxed{\begin{matrix} c=0.01 \\ n_0=1 \end{matrix}}$

($c_1 n^9$: $O(n^9)$; $c_2 n^9$: $\Omega(n^9)$)

$\rightarrow c_1 n^9 \leq 800000n^2 \leq c_2 n^9$ $\frac{c_1}{800000} \leq \frac{1}{n^7} \leq \frac{c_2}{80000}$ $\boxed{\begin{matrix} c=800000 \\ n_0=1 \end{matrix}}$

$\rightarrow c_1 n^9 \leq n \leq c_2 n^9$ $c_1 \leq \frac{1}{n^8} \leq c_2$, $\boxed{\begin{matrix} c=1 \\ n_0=1 \end{matrix}}$

Yes, it is $\Theta(n^3)$ ✓

e) $n^8 + 0.0000001n^5$ is $\Omega(n^{13})$?

$\rightarrow n^8 \geq cn^{13}$ $\boxed{\begin{matrix} c=1 \\ n_0=1 \end{matrix}}$ $\rightarrow 0.0000001n^5 \leq cn^{13}$ $\boxed{\begin{matrix} c=0.0000001 \\ n_0=1 \end{matrix}}$

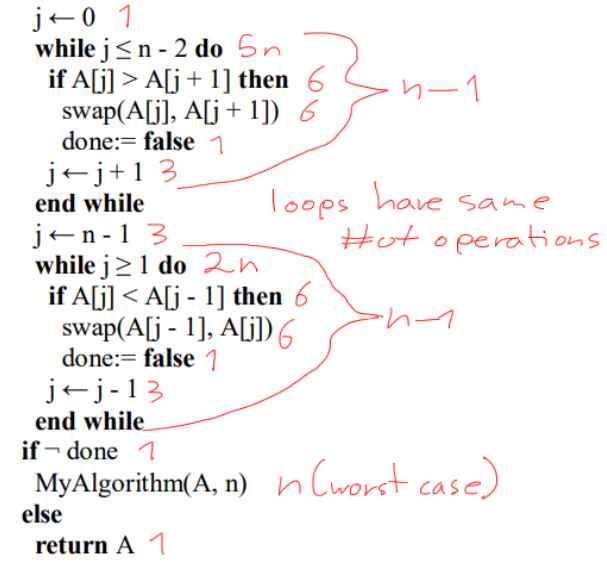
Yes, it is $\Omega(n^{13})$ ✓

f) $n!$ is $O(3^n)$

$n! \leq c3^n$ $\boxed{\begin{matrix} c=1 \\ n_0=1 \end{matrix}}$ Yes, it is $O(3^n)$ ✓

Question 3:

```
Algorithm MyAlgorithm(A, n)
 Input: Array of integer containing n elements
 Output: Possibly modified Array A
  done ← true
  j ← 0
  while j ≤ n - 2 do
   if A[j] > A[j + 1] then
    swap(A[j], A[j + 1])
    done:= false
   j ← j + 1
  end while
  j ← n - 1
  while j ≥ 1 do
   if A[j] < A[j - 1] then
    swap(A[j - 1], A[j])
    done:= false
   j ← j - 1
  end while
 if ¬ done
  MyAlgorithm(A, n)
 else
  return A
```



This is an approximated guess of counting the number of operations

a) The algorithm contains two loops that iterate the whole array (excluding the last index) depend on the input size. One loop iterates from left to right and the other loop iterates from right to left. Therefore, if one loop has n time complexity, then both together have 2n. This algorithm also includes recursion, which occurs when the algorithm above has not changed the done variable to false. The worst case could be (n-1)/2 recursions, meaning (n-1)/2 invocations. Thus, the algorithm is repeated (n-1)/2 times, meaning an additional order of complexion is added. The Big-0 and Big-Omega time complexity for this algorithm is **$O(n^2)$ for worst case and $\Omega(n)$ for best case.** Best case is when no change is made in the array and no recursive calls are made, which is explained later why.

b) The **Array A = (4,11,5,3,2) becomes A = (2,3,4,5,11).** The recursion call is made twice. On the second recursion call, the algorithm just enters the loops without accessing the conditional statements and the index just iterates. The else statement is reached, indicating that no change in the array was made during the last recursive call. The final result is the modified array**. It is sorted from smallest to biggest value.**

c) The algorithm is concluded to be a **sorting array algorithm**. The first loop finds a variable and pushes is towards the top, the second loop pushes another variable towards the bottom and the recursion occurs only if the array has not yet been sorted, depending on the change of the done variable. The done variable indicates whether the variable moving has reached its position, being on top of the lower values and bellow the higher values. If the done variable is true, the algorithm enters the while loops and does not change anything. In the end, the recursion invocation ends and the array is returned. If the array is already sorted, recursion does not occur and the while loops only iterate without accessing the if-statements. This is the best-case scenario, resulting in **Ω(n) time complexity.**

d) The runtime of this algorithm can be improved, but it depends on the amount of input. For smaller value, the algorithm with $O(n^2)$ complexity may result in less runtime. For larger arrays, algorithms of O(n) time complexity can be faster. The radix sort, which is in O(n), can be faster. Because key values, like the maximum, are stored for later use to speed up process. The great thing about this algorithm is that it is **tail recursive**, therefore, there is no recursive stack and the algorithm occurs first, then the recursive invocation is made.

e) The number of executions mainly depends on the condition of the array and not necessarily on the amount of input elements. An array with 10 elements and an array of 6, may at first give us the impression that the smaller array may finish faster, but if the bigger array has all of its elements already sorted, it will finish a lot faster, due to no recursive calls appearing. If the smaller array was sorted on the opposite way, it would appear to call more recursive calls, due to being the worst case. The provided file for this exercise provides results of different example arrays. The phrase **"EXECUTION"** indicates the amount of recursive calls made for each array example.

Programming question:

a) The code and text files for part A can be found on the corresponding folder within the zip file.

Pseudocode:

Algorithm MultipleTetranacci(n)

Input: positive integer n

Output: tetranacci number of integer n

if **n < 0** {throw new Exception}

if **n == 0** or **n == 1** or **n == 2** {return 0} // base cases

else if **n == 3** {return 1} // base case

else

return { **MultipleTetranacci (n-1)** + **MultipleTetranacci (n-2)** + **MultipleTetranacci (n-3)**+ **MultipleTetranacci (n-4)** }

Algorithm LinearTetranacci(n)

I**nput:** positive integer n

Output: tetranacci number of integer n

if **n < 0** {throw new Exception}

int a ← 0, b ← 1, c ← 1, d ← 2; // initializing base cases

int sum ← 0;

if **n == 0** or **n == 1** or **n == 2** {return 0} // base cases

else if **n == 3** or **n == 4** {return 1} // base cases

else if (n == 5) {return 2} // base case

```
else {
        for i ← 5 to n (excluded) do {
                sum ← a + b + c + d
                a ← b
                b ← c
                c ← d
                d ← sum
        }
        return sum
}
```

b) The first algorithm runs with **multiple recursion**. For every recursive call, it is possible for four more to occur. This is because the *tetranacci* numbers are the last 4 *tetranacci* numbers. Therefore, for *tetranacci(n),* we would need the sum for *tetranacci(n-1), tetranacci(n-2), tetranacci(n-3), tetranacci(n-4).* For every recursive invocation called, the stacks **increase exponentially**. This results in **O(n^4)** complexity.

The second algorithm uses only a loop that calculates the tetranacci numbers by variable sums. These sums occur via loop based on the number provided in the input. There are **no recursions occurring and no arrays**. The complexity is **only O(n).**

c) **Yes, it can occur.** The algorithm provided has a linear time complexity **without a recursion stack pilling up**. The recursion call appears its algorithm has been checked and completed. The base case is checked first (if n is less than 4), then the summation of the four previous tetranacci numbers occurs. Before the next recursive call is called, the other tetrannaci numbers are set within the array. The pseudocode of the tail-recursive Tetranacci algorithm can be found bellow.

Pseudocode:

Algorithm tailTetranacci(A,n)

Input: Array A of n integers

Output: Array A of n integers (possibly modified)

if **n < 4** then return A[n]

int sum ← A[0] + A[1] + A[2] + A[3]

for **i ← 0** to 3 (excluded) do

A[i] ← A[i+1]

A[3] ← sum (add the sum of tetranacci numbers to last index)

Return tailTetranacci(A, n-1) (now recursion goes backwards until base case is reached)