COMP 352: Data Structure and Algorithms
Assignment 1

Duc Nguyen, ID 40064649

Q1:

Part A Pseudo Code

Algorithm ReverseOrMultiply (A, n, i, j, isCross)

Input: An array of integer A, an integer n ($n \geq 4$) indicates the size of the array, a non-negative integer i, a non-negative integer j, a boolean value 'isCross' keeps track of whether or not the algorithm crossed the middle index of the array.

Output: modified array A

```
if j >= n then
  return A

if n mod 2 == 0 then
    if j < Floor (n/2) then
       swap(A[i], A[j])
       return ReverseOrMultiply(A, n, i+2, j+2, false)
    else
       if (cross == false) and (Floor(n/2) mod 2 ==1)
         i <- i+1
         j <- j+1

       mult(A[i], A[j])
       return ReverseOrMultiply(A, n, i+2, j+2, true)
else
    if j<= Floor (n/2) then
        swap(A[i], A[j])
        return ReverseOrMultiply(A, n, i+2, j+2, false)
    else
        mult(A[i], A[j])
        return ReverseOrMultiply(A, n, i+2, j+2, true)
```

Note: The above algorithm relies on 2 other methods: swap, and mult. These 2 methods could be easily implemented. Thus, for the sake of simplicity, I decided to not include here. Also, I have written a test program for this question (see attached folder testAlgorithmQ1) to implement the algorithm, and hopefully make it easier to understand in practise, although it is beyond the scope of the question.

Part B Time complexity in term of Big-O

The algorithm is linear recursion since it examines index from 0 up to the last element of the array, and it makes only one recursive call each time it is invoked. Additionally, the recursive call is its last step each time. Thus, the algorithm is a tail recursive algorithm with time complexity of **O (n)**.

Part C Space complexity in term of Big-O

The algorithm used tail recursive method. Each recursive call, the stack only goes up once time. Therefore, the memory is remained of space complexity **O (1)**.

Q2: Prove or disprove the following statements

a.

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

$f(n) = n^{22} \log n + n^7$

$n^{22} \leq n^{22}$

$\log n \leq \log n$

$n^7 \leq n^{22}$
Thus,

$f(n) \leq n^{22} \log n$

$n^{22} \log n \geq n^7 \log n \Rightarrow O(n^7 \log n)$ is not Big-O of the function $f(n)$

Conclusion: the statement is incorrect

b.

$10^7 n^5 + 5n^4 + 9000000n^2 + n$ is $\theta(n^3)$

$10^7 n^5 + 5n^4 + 9000000n^2 + n \leq cn^5$

Thus, the function is of O (n^5)

$n^5 \geq n^3$

Conclusion: The function is not $\theta(n^3)$ since it is not $O(n^3)$. The statement is incorrect.

c.

n^n is $\Omega(n!)$

Function $f(n) = n^n$ is of exponential function, while n! is of factorial function.

$n^n = n * n * n * \cdots * n \geq n! = n * (n-1) * (n-2) * \cdots * 1$

Conclusion: The statement is correct. Function $f(n)$ is of $\Omega(n!)$.

d.

$0.01n^9 + 800000n^7$ is $\theta(n^9)$

The function has $O(n^9)$ since $0.01n^9 + 800000n^7 \leq cn^9$

The function also has $\Omega(n^9)$ since $n^9 \leq 0.01n^9 + 800000n^7$

Conclusion: The statement is correct. Function $f(n)$ is of $\theta(n^9)$ since it is both $O(n^9)$ and $\Omega(n^9)$

e.

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

$n^8 \leq n^8$

$0.0000001n^5 \leq n^8$

$f(n) \leq n^8$

$n^8 \leq n^{13}$

Thus, $f(n) \leq n^{13}$

Conclusion: The statement is incorrect. $\Omega(n^{13})$ is not the Big-Omega of $f(n)$

f.

$(n!)$ is $O(3^n)$
Take n = 7 to be the case.
(n!) = 7! = 5040
$3^n = 3^7 = 2187$
$n! > 3^n$
Conclusion: $O(3^n)$ is not the Big-O of $f(x)$. The statement is incorrect.

Q3:

Pseudo Code

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
      return MyAlgorithm(A, n)
   else
      return A
```

Part A Big-O and Big-Omega

Big-O By going through 2 while loop each time it being called, the algorithm sorts the given array firstly from left to right and then, right to left on the second while loop.

- In the first while-loop, the algorithm runs with time complexity of n
- In the second while-loop, the algorithm runs again with time complexity of n
- Resulting in time complexity of each recursive call of n+n = 2n.
- Each recursive call, the algorithm will push the largest number it found (which currently is not in the correct order) to its correct position, and put the smallest number it found (which currently is not in the correct order) to its corresponding position.
- Thus, worst case scenario is when the algorithm has to do $\frac{n}{2}$ recursive calls.
- **Conclusion:** The time complexity of the algorithm is of **O** (n^2)

Big-Omega The best case scenario of this problem is when the array was already organized from the beginning. As a result, the algorithm will not even have to load the second call to itself.

Conclusion: The time complexity of the algorithm is of Ω**(n)**

Part B Tracing algorithm (hand-run) for an array A = (4,11,5,3,2). Resulting A?

- **Input:** A = (4,11,5,3,2)
- **First call**

  First while loop:

  (4, 11, 5, 3, 2)

  (4, **5, 11**, 3, 2)

  (4, 5, **3, 11**, 2)

  (4, 5, 3, **2, 11**)

  Second while loop:

  (4, 5, 3, 2, 11)

  (4, 5, **2, 3**, 11)

  (4, **2, 5**, 3, 11)

  (**2, 4**, 5, 3, 11)

- **Second call**

First while loop:

(2, 4, 5, 3, 11)

(2, 4, **3, 5**, 11)

Second while loop:

(2, 4, 3, 5, 11)

(2, **3, 4**, 5, 11)

- **Third call**
- **return (2, 3, 4, 5, 11)**

Part C The purpose of MyAlgorithm? What can be asserted about its result given any arbitrary array A of n integers?

Purpose: The purpose of MyAlgorithm is to sort the given array and return the new array in smallest to largest order.

Part D Can the runtime of MyAlgorithm be improved easily? how?

MyAlgorithm can be only improved in syntax since the runtime is already the most efficient for sorting.

Part E Implement the algorithm using actual Java code.

The implementation of the algorithm was written in Java. Please check out the folder 'testAlgorithmQ3' for source file.
There is a mismatch when comparing Big-O to the actual number of execution. This mismatch was caused because Big-O is only estimated by the function $g(n)$ taken from the function $f(n) \leq cg(n)$, while the number of execution was also accounted for the constant c also.