# ASSIGNMENT 1

WRITTEN QUESTIONS PART

Data Structures and Algorithms

COMP 352

Laila Alhalabi # 40106558

Date: February 5, 2021

**Question 1**

Consider an array A[1..n] of random length and random positive integer values. We define a subarray of A as a contiguous segment of A. We denote the subarray from position k to position l (both included) as A[k..l]. The subarray A[k..l] is an *ascent* if A[j] ≤ A[j + 1] for all j where k ≤ j < l. In other words, an ascent is a nondecreasing segment of A.

Develop well-documented **pseudo code** (not java code) that compute the maximal length of an ascent in A. For instance, given an array A =[5; 3; 6; 4; 6; 6; 7; 5], your algorithm should display:

The maximal length of an ascent would be 4, and A[4..7] = [4; 6; 6; 7] is the longest ascent in the array A.

(Notice that this is just an example. Your solution must not refer to this particular example). Your algorithm **cannot** use any auxiliary storage such as 'extra' arrays to perform what is needed.

a) Briefly justify the motive(s) behind your design.
b) What is the time complexity of your algorithm, in terms of Big-O?
c) What is the space complexity of your algorithm, in terms of Big-O?

Solution

```
Algorithm arrayAscent(a[])
Input array a of n positive integers
Output maximal length of an ascent in a
Initialize maxLength ← 1, length ← 1, starts ← 0
for i ← 1 to n - 1 do
      if a[i] ≤ a[i+1] then
            length ++
      else
           if length > maxLength then
              maxLength = length
           starts ←  i - max
          length ←  1
 if Length > maxLength then
     maxLength ←  length
  starts ←  (n – (maxLength +1))
print "The maximal length of an ascent is " + maxLength
    + ", and [ " + a[starts] + ".." + a[(starts + max - 1)] + " ] is the longest ascent "
```

A) *arrayAscent*(*a[]*) method takes an array of integers and iterate over them to check if the integers are increasing, and if so then it increases the *Length* by 1 as it loops. However, if there was found a descent integer, then it assigns the *Length* calculated to *maxLength* and 1 to *Length* and loops until finding the longest ascent in the array. Finally, it compares *Length* and *maxLength* and returns the longest ascent and *maxLength* that were found without using any storage tools.

B) The used method was a single loop which is a linear search that iterates from 0 to *n*, so the time complexity of the algorithm is O(*n*).

C) The space complexity of the algorithm is O(1) as it is a linear search.

**Question 2**

Develop well-documented **pseudo code** (not java code) that finds the largest and smallest sum of two elements in an array of n integers, $n \geq 1$. The code must display the values and the indices of these elements, and the value of that sum. For instance, given the following array [13;9;47;35;6;29;69;12] your code should find and display something similar to the following (notice that this is just an example. Your solution must not refer to this particular example):

- The two indices with largest sum between their values are: index 2 and index 6, storing values 47 and 69, and the value of their sum is 116.
- The two indices with smallest sum between their values are: index 1 and index 4, storing values 9 and 6 and the value of their sum is 15.

In case of multiple occurrences of the smallest or largest sum, the code should display the first found occurrence.

a) Briefly justify the motive(s) behind your design.
b) What is the time complexity of your solution? You must specify such complexity using the Big-O notation and explain clearly how you obtained such complexity.
c) What is the space complexity of your algorithm?

Solution

**Algorithm** *largest_smallest_sum*(*a*[])
**Input** array *a* of *n* positive integers
**Output** largest and smallest sum of elements in an array with their values and indexes.
**Initialize** *largest* ← a[0], *secondLargest* ← 0, *largestIndex* ← 0, *secondLargestIndex* ← 0
*smallest* ← a[0], *secondSmallest* ← 0, *smallestIndex* ← 0, *secondSmallestIndex* ← 0
**for** *i* ← 0 **to** *n* **do{**
**if** $n \leq 1$ **then print** *"Invalid"* **return**
**if** *largest* < *a*[*i*] **then**
*secondLargest* ← *largest*
*secondLargestIndex*← *largestIndex*
*largest* ← *a[i]*
*largestIndex* ← *i*
**else if** *a[i]* > *secondLargest* **and** ⌐*(a[i]* = *largest)* **or** *largest* = *secondLargest* **then**
*secondLargest* ← *a[i]*
*secondLargestIndex* ← *i*
**if** *smallest* > *a*[*i*] **then**
*secondsmallest* ← *smallest*
*secondSmallestIndex*← *smallestIndex*
*smallest* ← *a[i]*
*smallestIndex* ← *i*
**else if** *a[i]* < *secondSmallest* **and** ⌐*(a[i]* = *smallest)* **or** *smallest* = *secondSmallest* **then**
*secondLargest* ← *a[i]*
*secondLargestIndex* ← *i*
**}print** "Indices with largest sum:" *secondLargestIndex* + "and" + *largestIndex*
"Values:" a[*secondLargestIndex*] + "and" + a[*largestIndex*]
"Sum:" (a[*secondLargestIndex*] + a[*largestIndex*])
**print** "Indices with smallest sum:" *secondSmallestIndex* + "and" + *smallestIndex*
"Values:" a[*secondSmallestIndex*] + "and" + a[*smallestIndex*]
"Sum:" (a[*secondSmallestIndex*] + a[*smallestIndex*])

A) The used method was loop that iterates to find the two largest and smallest values meanwhile and stores their indexes and values in variables, then displays the two sums. The space complexity in this method is linear because the values are stored in variables. Even though there are better time complexity algorithms, space complexity is always more important which encouraged me to use this method.
B) The time complexity is O(*n*) since the used method is a single loop that checks the elements *n* times.
C) The space complexity is O(1) since the method uses linear search and stores values using variables.

**Question 3**

Prove or disprove the following statements, using the relationship among typical growth-rate functions seen in class.

a) $n^6 \log n + n^4$ is $O(n^4 \log n)$
b) $10^6 n^6 + 5n^3 + 6000000n^2 + n$ is $\Theta(n^3)$
c) $6n^n$ is $\Omega(n!)$
d) $0.5n^8 + 700000n^5$ is $O(n^8)$
e) $n^{13} + 0.0008n^6$ is $\Omega(n^{12})$
f) $n!$ is $O(5^n)$

Solution

For finding O(g(n)) we're always interested in finding the upper bound ($f(n) \leq g(n)$), while for finding the $\Omega(g(n))$ we're interested in finding the lower bound ($f(n) \geq g(n)$).

A) Declaring the functions $\mathbf{f(n) = n^6 \log n + 4}$ and $\mathbf{g(n) = n^4 \log n}$
$\mathbf{n^6 \log n > n^4 \log n \Rightarrow f(n) > g(n)}$. Since we are interseted in the worst case estimations, the statement is *false* and the true solution is $\mathbf{O(n^6 \log n)}$.

B) Declaring the functions $\mathbf{f(n) = 10^6 n^6 + 5n^3 + 6000000n^2 + n}$ and $\mathbf{g(n) = n^3}$
$\mathbf{10n^6 + 5n^3 + 6000000n^2 + n > n^3 \Rightarrow f(n) > g(n)}$ which is *false* and the true solution is $\mathbf{O(n^6)}$

C) Declaring the functions $\mathbf{f(n) = 6n^n}$ and $\mathbf{g(n) = n!}$
$\mathbf{6n^n > n! \Rightarrow f(n) > g(n)}$. Since we're intersted in the lower bound, then $\mathbf{\Omega(n!)}$ is *true*.

D) Declaring the functions $\mathbf{f(n) = 0.5n^8 + 700000n^5}$ and $\mathbf{g(n) = n^8}$
$\mathbf{0.5n^8 + 700000n^5 < n^8 \Rightarrow f(n) < g(n)}$. Since we're interested in the upper bound $\mathbf{O(n^8)}$ is *true*.

E) Declaring the functions $\mathbf{f(n) = n^{13} + 0.0008n^6}$ and $\mathbf{g(n) = n^{12}}$
$\mathbf{n^{13} + 0.0008n^6 > n^{12} \Rightarrow f(n) > g(n)}$. Since we're intersted in the lower bound, then $\mathbf{\Omega(n^{12})}$ is *true*.

F) Declaring the functions $\mathbf{f(n) = n!}$ and $\mathbf{g(n) = 5^n}$
$\mathbf{n! > 5^n \Rightarrow f(n) > g(n)}$. Since we're interested in the upper bound, $\mathbf{O(5^n)}$ is *false* and the true solution is $\mathbf{O(n!)}$.