CSci 242: Algorithms and Data Structures

Instructor: Dr. M. E. Kim Date: September 3rd, 2019

Due: 11:59 PM, September 12th (Friday.), 2019. (No Extension) Name: Elena Corpus

**Home Assignment 1: 100 points + 10 points (optional)**

Q1. [22] **Min-Max** recursive algorithm

1. [10] Write a ***recursive*** algorithm in a pseudo code, **Min-Max**, for finding *both* the *minimum* and the *maximum* elements in an array A of *n* elements. Your algorithm calls itself ***only once*** within the algorithmand should *return* a pair (*a, b*) where *a* is the minimum element and *b* is the maximum element.

**Algorithm** **Min-Max**(A, *n*)

**Input:** an Array A of *n* elements

**Output:** a pair of (*a, b*) where *a* is the minimum element and *b* is the maximum element

Define findMaxMin ( array, left, right) then

If the right side is <= left + 1

If left array is < right array then

Return left, right

Else return right, left

1. [12] Implement your algorithm of Q1 either in Python or in Java programming language, returning a pair of the maximum and the minimum elements.



Q2. [25] For a given algorithm below,

**Algorithm** **Loop2**(*n*):

*s <--* 0

(i) **for** *i <--*  1 **to** *n* **do**

(ii) **for** *j <--*  *1* **to** *i* **do**

(iii) *s <--*  s + 2*i*

1. (A) [5] Count the number of primitive operations in each statement, (i) – (iii), of the algorithm and (B) [5] get the total number of primitive operations executed in the algorithm.

See the Handout 2.

(A) :

(i) : n because it is running n number of times

(ii) : 1 + 2 + 3 + … + n = n(n+1)/2 ; The inner loop is nested with the outer loop, thus the inner loop is running i number of times each time the outer loop runs

(iii) : running n(n+1)/2 times since it is in the loop

(B) : Total number is n(n+1)/2

[5] Give the smallest Big-Oh characterization of the running time in (B), in terms of *n.* e.g.) *O(n)*

According to the definition of big Oh, the running time is n(n+1)/2 = (n2 +n)/2 = O(n2)

1. [10] ***Prove*** your answer in 2) by the ***definition of big-Oh***. i.e. You have to find the positive constant *c* and *n0* that satisfies the condition of the big-Oh definition. See the examples in the slides # 22 - #26.

(n2 +n)/2 = O(n2)

Meaning :

(n2 + n ) / 2 ≤ c \* n2

(n2 + n) ≤ 2\* c \* (n2)

Thus c = 1 makes the inequality true because …

(n2 + n)/ 2 ≤ 1 \* n2

n2 + n ≤ 2n2

n ≤ n2

0 ≤ n

Thus satisfying the Big Oh definition

Q3. [10] Prove that *n2 log2 n*= Ω(*n2*) by the definition of big-Omega.

Find n0 and c ; c > 0 and n ≥ n0

n2 log(n)= Ω(n2)

n2 log(n) ≥ c(n2)

c = 1

n2 log(n) ≥ c(n2)

n2 log(n) ≥ 1(n2)

log(n) ≥ 1

This above equation is true, for all n ≥ 2

so, n2 log(n)= Ω(n2) for c = 1 and n0 = 2

Q4. [10] Prove that 2*n2 - 5n - 3* = (big-theta)(*n2*) by the definition of big-Theta.

2n2 - 5n - 3 = 𝛩(n2)

c1(n2) ≤ 2n2 - 5n - 3 ≤ c2(n2)

c1 = 1 and c2 = 2

c1(n2) ≤ 2n2 - 5n - 3 ≤ c2(n2)

c1(n2) ≤ 2n2 - 5n - 3 ≤ c2(n^2)

c1(n2) ≤ 2n2 - 5n - 3 and 2n2 - 5n – 3 ≤ c2(n2)

n2 ≥ 5n+3 and 5n+3 ≥ 0

Then, n ≥ 6

so, 2n2 - 5n - 3 = 𝛩(n2) given c1 = 1, c2 = 2 and n0 = 6

Q5. [10] In the **MaxsubFastest** algorithm to find the maximum subarray,

1. [10] Modify the **MaxsubFastest** algorithm so that it uses only *a single loop* and, instead of computing *n*+1 different *Mt* values, it maintains just a ***single variable*** **M** for *Mt*s. (Psuedocode)

currentMax <-- initialMin

endingMax <-- 0

For I in 1 to n do

endingMax <-- endingMax + A[I]

If currentMax < endingMax then

currentMax <-- endingMax

If endingMax < 0 then

endingMax <-- 0

Return currentMax

1. [Optional, 10] Modify the **MaxsubFastest** algorithm so that it returns both the ***value of the maximum subarray*** ***summation*** and the indices ***j*** and ***k*** that identify the maximum subarray A[*j* : *k*]. The running time of your algorithm should be O(*n*).

CurrentMax <== initialMin

EndingMax <-- 0

MaxArrayStart <-- -1

MaxArraryEnd <-- -1

FalseStart <-- 0

For I in 1 to n do

EndingMax <-- endingMax + a[I]

If currentMax < endingMax then

CurrentMax <-- endingMax

MaxArrayStart <-- falseStart

MaxArraryEnd <-- I

If endingMax < 0 then

FalseStart <-- I

EndingMax <-- 0

Return currentMax, maxArrayStart, maxArrayEnd

Q6. [23] 1) [10] Suppose you are given an integer ***c*** and an array, A, indexed from 1 to *n*, of *n* integers in the range from 0 to 5*n* (possibly with duplicates). i.e. 0 <= A[*i* ] <= 5*n ∀ I = {1, .., n}. (use hash table, and comment the use rather than coding the entire thing)*

Write an efficient algorithm that runs in **O(*n*)** time in a pseudo code for determining if there are two integers, A[*i*] and A[*j*], in A whose sum is ***c***, i.e. ***c*** = A[*i*] + A[*j*], for 1 <= *i* < *j* <= *n*. Your algorithm should return a *set of any pair of those indices (i , j).* If there were no such integers, return (0, 0).

Creating a hashmap <int, int> storing the numbers that are visited already

For each number n in array a then

If (c-n) is present then

Get i from hashmap

Return currentIndex, i

If NOT then

N <-- key

HashMap <-- currentIndex

Return (0,0)

2) [13] Implement your algorithm of Q7 either in Python or in Java programming language where A[1:10] = [30, 25, 10, 50, 35, 45, 40, 5, 15, 20] and c = 40.

