PSUEDOCODE QUESTION 1A

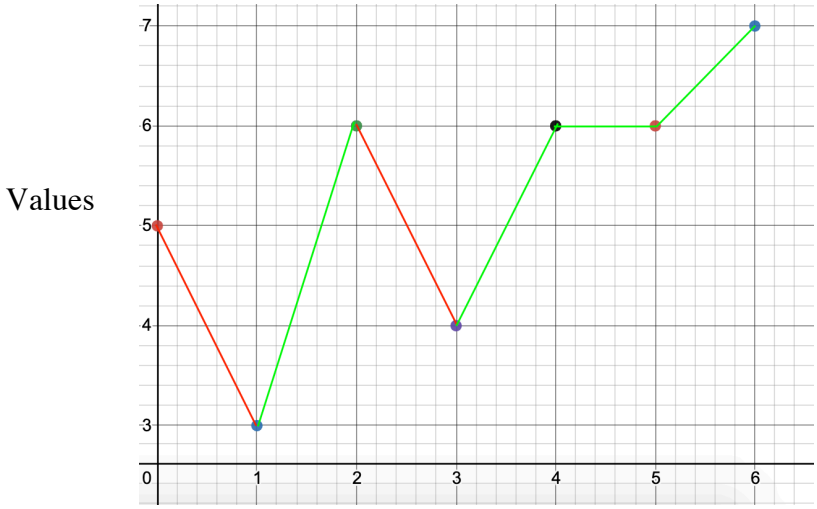
```
1    def max_ascent_function(A, n):
2         //INPUT: array A of n integers
3
4    max_ascent = 0         //Length of the max ascent
5    max_ascent_index = 0   //Starting index of the max ascent
6
7    curr_ascent = 0        //Length of current ascent
8    cur_ascent_index = 0   //Index of current ascent
9
10   FOR i ← 1 to n-1 DO  //Loop over array A (1 & n-1 inclusive)
11
12        IF A[i] >= A[i-1] THEN              //Checks for ascent
13             IF curr_ascent == 0 THEN    //Checks if new ascent
14                  curr_ascent_index = i   //Stores start index
15             ENDIF                         //For line 13
16        curr_ascent = curr_ascent + 1  //Add to ascent length
17        ELSE                           //For line 12
18             //Checks if current ascent bigger than max ascent
19             IF curr_ascent > max_ascent THEN
20                  max_ascent = curr_ascent    //Update values
21                  max_ascent_index = curr_ascent_index
22             curr_ascent = 0          //Reset current ascent
23        ENDIF                    //For line 12
24   ENDFOR                        //For line 10
25
26   //Special case handling
27   IF curr_ascent > max_ascent THEN   //Updating values for
28        max_ascent = curr_ascent      //When ascent is at end
29        max_ascent_index = curr_ascent_index
30   ENDIF     //For line 27
31
32   IF max_ascent != 0 THEN  //Check if ascent exist and print
33
34        PRINT("The maximal length of ascent would be " +
          max_ascent+1 + ", and A[" + x[max_ascent_index-1] +
          ".." + x[max_ascent_index+max_ascent-1] + "] = " +
          x[max_ascent_index-1:max_ascent_index+max_ascent] + "
          is the longest ascent in array X")

35   ELSE
36        PRINT("No ascent in this array")
37   ENDIF     //For line 32
```

1a) Briefly justify the motive(s) behind your design.

The algorithm uses a single for-loop to go over the array and keep a track of every ascent present in the array. Should the current ascent be greater than the maximal ascent it simply replaces the variables associated with the maximal ascent and then prints all the details required to the console.

Lines 4 to 8: Declare variables
Line 10: Start for-loop
Lines 12 to 16: Checks if the current element is greater than its previous element (ascent) and if true checks whether it is a start of a new ascent. If true again, it saves the index and if not it simply increases the length of the current index by one
Lines 18 to 22: Occurs only if the current element and its previous element is not an ascent. In this case the current ascent breaks and it checks to see if the current ascent was bigger than the maximum ascent. If true then it replaces the variables associated with the maximum ascent and resets current ascent.
Lines 27 to 30: Special case handling, occurs only when the maximal ascent is present at the end of the array. Checks to see if it was greater than the maximum and then replaces the values

The technique used for this algorithm is known as "valleys and peaks".
Example: A = [5, 3, 6, 4, 6, 6, 7]. n = 7



The green lines repesent ascents in the array A. While the red lines represent descents. The algorithm goes over the array keeping a track of all the green ascents. Each ascent is compared to the maximum ascent to see if it is greater than the maximum ascent.

Similarly, from index 3 to 6 in the diagram an ascent is completed – but since it is the end of the array the loop exits. This is a special case and so the comparision is done outside of the loop.

Zafir Khalid (40152164)

1b) What is the time complexity of your algorithm, in terms of Big-O?

Line Number	Statement	Worst case executions
4	max_ascent = 0	1
5	max_ascent_index = 0	1
6	curr_ascent = 0	1
7	curr_ascent_index = 0	1
10	FOR i ← 1 to n-1 DO	
	Initialize i	1
	Compare i	n
	Increment i	n-1
12	IF A[i] >= A[i-1] THEN	n-1
13	IF curr_ascent == 0 THEN	n-1
14	curr_ascent_index = i	n-1
16	curr_ascent = curr_ascent + 1	n-1
19	IF curr_ascent > max_ascent THEN	n-1
20	max_ascent = curr_ascent	n-1
21	max_ascent_index = curr_ascent_index	n-1
22	curr_ascent = 0	n-1
27	IF curr_ascent > max_ascent THEN	1
28	max_ascent = curr_ascent	1
29	max_ascent_index = curr_ascent_index	1
32	IF max_ascent != 0 THEN	1
34	PRINT(…)	1
36	PRINT("No ascent in this array")	1

Summing up the worst case executions column we have the time function:

$$T(n) = 11 - 9 + 10n$$
$$T(n) = 2 + 10n$$

Since $T(n) = 2 + 10n$ is a linear polynomial, it follows that T(n) can be represented as O(n) by dropping the constant values.

$$Therfore\ we\ can\ conclude\ that\ the\ max\ ascent\ function\ is: \boldsymbol{O(n)}$$

1c) What is the space complexity of your algorithm, in terms of Big-O?

Throughout the algorithm no auxillary storage such as arrays, lists, stacks or queues are used. Only variables are used to store integers. The only array (A) in the algorithm is the one the user needs to input as a parameter.

Since the space complexity of variable assignments is O(1). We can conclude that:

$$The\ space\ complexity\ of\ the\ max\ ascent\ function\ is: \boldsymbol{O(1)}$$

Zafir Khalid (40152164)

PSEUDOCODE QUESTION 1B

```
1    def minmaxsum(A, n):
2           //INPUT: Array of n integers
3
4           //Initializing variables
5           smallest, sec_smallest = max(A)
6           smallest_index, sec_smallest_index = 0
7           largest, sec_largest = 0
8           largest_index, sec_largest_index = 0
9
10          //Looping through the array
11          FOR i ← 1 to n-1 DO
12                //Store current value of array
13                currval = A[i]
14                //Checks if greater than second largest
15                IF currval > sec_largest THEN
16                      //Checks if greater than largest
17                      IF currval > largest THEN
18                            //Swap variable values and save indices
19                            temp = largest
20                            largest = currval
21                            sec_largest = temp
22                            sec_largest_index = largest_index
23                            largest_index = i
24                      ELSE
25                      //Executes if sec_largest<currval<largest
26                            sec_largest = currval
27                            sec_largest_index = i
28                      ENDIF
29                ENDIF
30                //Checks if smaller than second smallest
31                IF currval < sec_smallest THEN
32                      //Checks if smaller than smallest
33                      IF currval < smallest THEN
34                            //Swap variable values and indices
35                            temp = smallest
36                            smallest = currval
37                            sec_smallest = temp
38                            sec_smallest_index = smallest_index
39                            smallest_index = i
40                      ELSE
41                //Executes if sec_smallest<currval<smallest
42                            sec_smallest = currval
43                            sec_smallest_index = i
44                      ENDIF
45                ENDIF
46          ENDFOR
47    //Print statements
48    print("The two indices with largest sum between their values are: index
      " + sec_largest_index + " and index " + largest_index + ", storing
      values " + sec_largest + " and " + largest + ", and the value of their
      sum is " + largest+sec_largest + ".")

49    print("The two indices with smallest sum between their values are:
      index " +sec_smallest_index + " and index " + smallest_index + ",
      storing values " + sec_smallest + " and " + smallest + ", and the value
      of their sum is " + (smallest+sec_smallest) +".")
```

2a) Briefly justify the motive(s) behind your design.

The algorithm uses a single for-loop to go over the array and checks every element to see if it is bigger than the second biggest and biggest number stored in the respective variables. It also checks to see if the current element is smaller than the second smallest and smallest number stored in the respective variables.

Lines 4 to 8: Declare variables
Line 11: Start for-loop
Lines 15 to 17: Checks if the current value is greater than the second largest value stored and also checks if the current value is greater than the largest value stored
Lines 19 to 23: If the current value is large enough, a swap occurs. A temp variable is created and the largest/second largest variables are swapped around.
Lines 26 to 27: If the value is bigger than the second smallest but not the smallest only the second smallest variables are manipulated
Lines 32 to 43: Checking and swapping of variables if they are small enough. Same process is used as that described in lines 15 to 27. Except in this case the values are checked to see if they are small and not large
Lines 48 to 49: Descriptive messages are printed out to the console to show the largest sum, minimal sum and their respective indices

The algorithm described loops over the array and stores the two largest values and two smallest values. By summing the two smallest values in the array we can obtain the smallest sum of the array. Similarly, by summing the two largest values of the array we can obtain the largest sum of the array.

2b) What is the time complexity of your algorithm, in terms of Big-O?

Line Number	Statement	Worst case executions
5	`smallest, sec_smallest = max(A)`	2
6	`smallest_index, sec_smallest_index = 0`	2
7	`largest, sec_largest = 0`	2
8	`largest_index, sec_largest_index = 0`	2
11	`FOR i ← 1 to n-1 DO`	
	`*Initialize i*`	1
	`*Compare i*`	n
	`*Increment i*`	n-1
13	`currval = A[i]`	n-1
15	`IF currval > sec_largest THEN`	n-1
17	`IF currval > largest THEN`	n-1
19	`temp = largest`	n-1
20	`largest = currval`	n-1
21	`sec_largest = temp`	n-1
22	`sec_largest_index = largest_index`	n-1
23	`largest_index = i`	n-1
26	`sec_largest = currval`	n-1
27	`sec_largest_index = i`	n-1
31	`IF currval < sec_smallest THEN`	n-1
33	`IF currval < smallest THEN`	n-1
35	`temp = smallest`	n-1
36	`smallest = currval`	n-1
37	`sec_smallest = temp`	n-1
38	`sec_smallest_index = smallest_index`	n-1
39	`smallest_index = i`	n-1
42	`sec_smallest = currval`	n-1
43	`sec_smallest_index = i`	n-1
48	`Print(…)`	1
49	`Print(…)`	1

Summing up the worst case executions column we have the time function:

$$T(n) = 11 - 20 + 21n$$
$$T(n) = 21n - 9$$

Since $T(n) = 21n - 9$ is a linear polynomial, it follows that T(n) is θ(n) since every linear polynomial is both Ω(n) and O(n).

Therfore we can conclude that the minmaxsum function is: $\boldsymbol{O(n)}$

2c) What is the space complexity of your algorithm, in terms of Big-O?

Throughout the algorithm no auxillary storage such as arrays, lists, stacks or queues are used. Only variables are used to store integers. The only array (A) in the algorithm is the one the user needs to input as a parameter.

Since the space complexity of variable assignments is O(1). We can conclude that:

$$The\ space\ complexity\ of\ the\ minmaxsum\ function\ is: \boldsymbol{O(1)}$$

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

Limit Asymptotic Theorem (Used throughout question 3)

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=L\quad\begin{cases}if\ L=0\ then\ f(n)\in O(g(n))\\ if\ L=c\ then\ f(n)\in\theta(g(n))\\ if\ L=\infty\ then\ f(n)\in\Omega(g(n))\end{cases}$$

a) $n^6\log(n)+n^4\ is\ O(n^4\log(n))$

$$\lim_{n\to\infty}\frac{n^6\log(n)+n^4}{n^4\log(n)}=\lim_{n\to\infty}\frac{n^4\log(n)\left(n^2+\frac{1}{\log(n)}\right)}{n^4\log(n)}=\lim_{n\to\infty}n^2+\frac{1}{\log(n)}=\infty$$

Therefore, $n^6\log(n)+n^4\ is\ O(n^4\log(n))$ is disproved.
The correct statement should read : $n^6\log(n)+n^4\ is\ \Omega(n^4\log(n))$

b) $10^6n^6+5n^3+6000000n^2+n\ \ is\ \theta(n^3)$

$$\lim_{n\to\infty}\frac{10^6n^6+5n^3+6000000n^2+n}{n^3}=\lim_{n\to\infty}\frac{n^3(10^6n^3+5+\frac{6000000}{n}+\frac{1}{n^2})}{n^3}$$

$$\lim_{n\to\infty}10^6n^3+5+\frac{6000000}{n}+\frac{1}{n^2}=\infty$$

Therefore, $10^6n^6+5n^3+6000000n^2+n\ \ is\ \theta(n^3)$ is disproved.
The correct statement should read : $10^6n^6+5n^3+6000000n^2+n\ \ is\ \Omega(n^3)$

c) $6n^n\ is\ \Omega(n!)$

$$\lim_{n\to\infty}\frac{6n^n}{n!}=\ 6*\lim_{n\to\infty}\frac{n^n}{n!}=6*\infty=\infty$$

$Since: n^n=(n\ *\ n\ *\ldots*\ n)\geq(1\ *\ 2\ *\ldots*\ n)\ =\ n!$
$We\ can\ conlude\ that\ n^n>>\ n!, and\ so\ \lim_{n\to\infty}\frac{n^n}{n!}=\infty$
Therefore, $6n^n\ is\ \Omega(n!)$ is proved.

d) $0.5n^8 + 700000n^5 \; is \; O(n^8)$

$$\lim_{n\to\infty} \frac{0.5n^8 + 700000n^5}{n^8} = \lim_{n\to\infty} \frac{n^8(0.5 + \frac{700000}{n^3})}{n^8} = \lim_{n\to\infty} 0.5 + \frac{700000}{n^3} = 0.5$$

Therefore, $\boldsymbol{0.5n^8 + 700000n^5 \; is \; O(n^8)}$ is disproved.
The correct statement should read: $0.5n^8 + 700000n^5 \; is \; \theta(n^8)$

e) $n^{13} + 0.0008n^6 \; is \; \Omega(n^{12})$

$$\lim_{n\to\infty} \frac{n^{13} + 0.0008n^6}{n^{12}} = \lim_{n\to\infty} \frac{n^{12}(n^1 + \frac{0.0008}{n^6})}{n^{12}} = \lim_{n\to\infty} n + \frac{0.0008}{n^6} = \infty$$

Therefore, $\boldsymbol{n^{13} + 0.0008n^6 \; is \; \Omega(n^{12})}$ is proved.

f) $n! \; is \; O(5^n)$

$$\lim_{n\to\infty} \frac{n!}{5^n} = 0$$

$Since: 5^n = (5 * 5 * \ldots * 5) \geq (1 * 2 * \ldots * n) = n!$
$We \; can \; conlude \; that \; 5^n >> n!, and \; so \lim_{n\to\infty} \frac{n!}{5^n} = 0$

Therefore, $\boldsymbol{n! \; is \; O(5^n)}$ is proved.