Instructions

- Write and submit source files with the exact names specified in each exercise.
- Do not submit any file, folder, or archive, other than what is required.
- Your code must work with Python 3.
- You may only use the following, limited subset of the Python language and libraries.

  You may only use the following built-in types:

  - numeric types, such as `int`
  - sequence types, such as arrays, tuples, and strings (so, no sets or dictionaries)

  With arrays or other sequence types, you may only use the following operations:

  - direct access to an element by index, as in `print(A[7])` or `A[i+1] = A[i]`
  - append an element, as in `A.append(10)`
  - delete the last element, as in `del A[−1]` or `A.pop()`
  - read the length, as in `n = len(A)`

  You may use the `range` function, typically in a for-loop, as in `for i in range(10)`

  You may not use any library or external function other than the ones listed above.
- If an exercise requires you to analyze the complexity of an algorithm written in Python, write your analysis as a code comment either at the beginning of the source file or anyway near the corresponding Python function.
- Document any known issue, using code comments if necessary.
- Submit each file through the iCorsi system.

▸**Exercise 1.** Consider the following algorithm ALGO-X(A, k) that takes a sequence A of n numbers and a positive integer k:

```
ALGO-X(A, k)
B = ALGO-Y(A, 1, A.length + 1)
c = 0
for i = 1 to B.length
    if i ≤ k
        c = c + B[i]
    else return c
return c
```

```
ALGO-Y(A, i, j)
D = empty sequence
if j − i == 1
    append A[i] to D
elseif j − i > 1
    k = ⌊(i + j)/2⌋
    B = ALGO-Y(A, i, k)
    C = ALGO-Y(A, k, j)
    b = i
    c = k
    while b < k or c < j
        if c ≥ j or (b < k and B[b] < C[c])
            append B[b] to D
            b = b + 1
        else append C[c] to D
            c = c + 1
return D
```

Answer the following questions in a text file `ex1.txt` or in a PDF file `ex1.pdf`.

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. *(5)*

Question 2: Analyze the complexity of ALGO-X. Is there a difference between the best- and worst-case complexity? If so, describe a best-case and a worst-case input of size n, as well as the behavior of the algorithm in each case. *(5)*

Question 3: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X, but with a strictly better complexity in the average case. Analyze the complexity of BETTER-ALGO-X. Notice that if ALGO-X modifies the content of the input array A, then BETTER-ALGO-X must do the same. Otherwise, if ALGO-X does not modify A, then BETTER-ALGO-X must not modify A. *(20)*

►**Exercise 2.** Consider the following algorithm ALGO-X(A, x) that takes a sorted sequence A of n numbers and a positive number x.

```
ALGO-X(A, x)
for i = 1 to A.length
    if ALGO-Y(A, i, A.length + 1, A[i] + x)
        return TRUE
return FALSE
```

```
ALGO-Y(A, i, j, x)
while j > i
    k = ⌊(i + j)/2⌋
    if x < A[k]
        j = k
    elseif x > A[k]
        i = k + 1
    else return TRUE
return FALSE
```

Answer the following questions in a text file `ex2.txt` or in a PDF file `ex2.pdf`.

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. *(5)*

Question 2: Analyze the complexity of ALGO-X. Is there a difference between the best- and worst-case complexity? If so, describe a best-case and a worst-case input of size n, as well as the behavior of the algorithm in each case. *(5)*

Question 3: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X, but with a strictly better complexity in the worst case. Analyze the complexity of BETTER-ALGO-X, showing a best-case and a worst-case input. Notice that if ALGO-X modifies the content of the input array A, then BETTER-ALGO-X must do the same. Otherwise, if ALGO-X does not modify A, then BETTER-ALGO-X must not modify A. *(20)*

►**Exercise 3.** Given a sequence of $2n$ numbers $A = x_1, y_1, x_2, y_2, \ldots, x_n, y_n$ representing the Cartesian coordinates of n points in the plane, $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots p_n = (x_n, y_n)$, consider the line segments p_i-p_j defined by pairs of distinct points in A. You may assume that no two points in A are identical. That is, $i \neq j$ implies $p_i \neq p_j$.

Question 1: In a source file `ex3.py` write two Python functions, `count_vertical(A)` and `count_horizontal(A)`, that given the sequence A structured as above, return the number of vertical and horizontal segments in A, respectively. Also, write an analysis of the complexity of your solution as a comment in the source file. *(10)*

Question 2: In the same source file `ex3.py`, write a Python function `intersection(A)` that returns `True` if A contains at least one vertical segment that intersects at least one horizontal segment, or `False` otherwise. Also, write an analysis of the complexity of your solution as a comment in the source file, in particular describing a worst-case input. *(20)*

Two segments intersect when they have at least one point in common. For example, a vertical segment $(1,7)$-$(1,0)$ intersects an horizontal segment $(0,1)$-$(10,1)$. Similarly, vertical segment $(1,7)$-$(1,0)$ intersects horizontal segment $(1,0)$-$(3,0)$. However, vertical segment $(1,7)$-$(1,0)$ does not intersect horizontal segment $(0,10)$-$(10,10)$.

For example, `intersection([9,3,5,6,0,9,3,2,6,7,7,9,3,5,1,8,8,4,9,0])` must return `False`, since the input does not contain intersecting vertical and horizontal segments.

Instead, `intersection([5,1,9,0,2,3,2,2,9,2,5,4,0,3,7,2,8,6,4,2])` must return `True`, since horizontal segment $(2,2)$-$(9,2)$ intersects vertical segment $(5,1)$-$(5,4)$; and `intersection([2,6,8,6,3,6,7,5,5,3,1,6,7,1,5,0,8,8,5,6])` must return `True` because horizontal segment $(2,6)$-$(8,6)$ intersects vertical segment $(8,6)$-$(8,8)$.

►**Exercise 4.** Given a sequence of numbers $A = a_1, a_2, a_3, \ldots, a_n$, we say that a subsequence $a_i, a_{i+1}, \ldots, a_j$ of length $j - i + 1 \geq 2$ is strictly increasing if $a_i < a_{i+1} < \cdots < a_j$, or strictly decreasing if $a_i > a_{i+1} > \cdots > a_j$. *(30)*

In a source file `ex4.py` write a Python function `increasing_or_decreasing(A)` that, given a sequence of numbers A, in time $O(n)$ returns the string `'increasing'` if A contains a strictly increasing subsequence that is longer than any strictly decreasing subsequence in A; or vice-versa the result is `'decreasing'` if A contains a strictly decreasing subsequence that is longer than any strictly increasing subsequence in A. If there are no strictly increasing or strictly decreasing subsequences, then the return value must be the string `'flat'`. If there are strictly increasing and strictly decreasing subsequences, but the maximal sequences of the two kinds are of equal length, then the return value must be `'equal'`. Also, write an analysis of the complexity of your solution.

You may use the following examples to test your code:

```
>>> increasing_or_decreasing([1])
'flat'
>>> increasing_or_decreasing([1,1,1,1,1])
'flat'
>>> increasing_or_decreasing([1,2,1,2,1])
'equal'
>>> increasing_or_decreasing([1,2,1,2,10,1])
'increasing'
>>> increasing_or_decreasing([1,2,3,2,8,10,1,0])
'equal'
>>> increasing_or_decreasing([1,20,11,10,1,0])
'decreasing'
```

Solutions

▷ *Solution 1.1*

ALGO-X returns the sum of the top-k elements of A.

▷ *Solution 1.2*

The complexity is $\Theta(n \log n)$. The algorithm uses *merge-sort* as the main subroutine, plus a linear scan that is at most $\Theta(n)$. So the dominating complexity is the complexity of *merge-sort*, which is $\Theta(n \log n)$ and is the same in the worst and best case.

▷ *Solution 1.3*

We can use the same idea of the classic divide-and-conquer *k-selection* algorithm for order statistics: we partition using a chosen pivot, then recurse, at most once.

```
BETTER-ALGO-X(A, k)
if k ≥ A.length
    return SUM(A)
v = random value in A
L = empty sequence
M = empty sequence
R = empty sequence
for i = 1 to A.length
    if A[i] < v
        append A[i] to L
    elseif A[i] > v
        append A[i] to R
    else append A[i] to M
if k < L.length
    return BETTER-ALGO-X(L, k)
if k − L.length ≤ M.length
    return SUM(L) + (k − L.length) ∗ v
return SUM(L) + M.length ∗ v
           + BETTER-ALGO-X(R, k − L.length − M.length)
```

```
SUM(A)
s = 0
for i = 1 to A.length
    s = s + 1
return s
```

The algorithm is really the same as *k-selection*, so the complexity analysis is the same: the worst case is quadratic, but the average and most common case is linear.

▷ *Solution 2.1*

ALGO-X returns TRUE if and only if there are two distincts elements $A[i]$ and $[j]$ at distance x from each other, meaning $A[i] - A[j] = x$ (with $i \neq j$), or FALSE otherwise.

▷ *Solution 2.2*

ALGO-X essentially invokes a binary search (ALGO-Y) for each element of $A[i]$ in the remainder of the array. The best-case complexity is constant, which corresponds to an input array of size n in which the first element is $A[1] = y$, and there is an element $A[\lfloor n/2 \rfloor + 1] = y + x$. The worst-case complexity is instead $\Theta(n \log n)$, which corresponds to an input array that contains no to elements at distance x, for example, $A = [2, 4, 6, 8, 10, \ldots, 2n], x = 1$.

▷ *Solution 2.3*

Since A is sorted, we can find two elements $A[i]$ and $A[j]$ at distance $A[j] - A[i] = x$ with a linear scan. Again, since A is sorted, we simply advance the index of the higher (further) element when the distance is less than x (so as to increase the distance), or we advancing the base index i when the distance is higher than x (so as to decrease the distance):

```
BETTER-ALGO-X(A, k)
i = 0
j = 1
while j < A.length
    if A[j] < A[i] + x
        j = j + 1
    elseif A[j] > A[i] + x
        i = i + 1
    else return TRUE
return FALSE
```

The best-case complexity is constant, for example with $A = [1, 2, \ldots, n], x = 1$. The worst-case complexity is when we don't find two elements at distance x. For example, $A = [2, 4, \ldots, 2n], x = 1$.

▷ *Solution 3.1*

```
def count_vertical(A):
    #
    # Complexity: \Theta(n^2), since we go through all the pairs of
    # points.
    #
    n = len(A)//2
    c = 0
    for i in range(n):
        for j in range(i + 1, n):
            if A[2*i] == A[2*j]:
                c = c + 1
    return c

def count_horizontal(A):
    #
    # Complexity: \Theta(n^2), since we go through all the pairs of
    # points.
    #
    n = len(A)//2
    c = 0
    for i in range(n):
        for j in range(i + 1, n):
            if A[2*i+1] == A[2*j+1]:
                c = c + 1
    return c
```

▷ *Solution 3.2*

```
def intersection(A):
    #
    # Complexity: \Theta(n^4). Consider in fact the worst-case input:
    # A = [0,1,0,2,0,3,0,4,0,5,...,0,n].   In this case, we go through
    # the n(n-1)/2 vertical segments, and for each one of them we go
    # through each of the same n(n-1)/2 pairs of points looking for
    # intersecting horizontal segments.
    #
    n = len(A)//2
    for v1 in range(n):
        for v2 in range(v1+1,n):
            if A[2*v1] == A[2*v2]:
                x = A[2*v1]
                y1 = A[2*v1+1]
                y2 = A[2*v2+1]
                for h1 in range(n):
                    for h2 in range(h1+1,n):
                        if A[2*h1+1] == A[2*h2+1]:
                            y = A[2*h1+1]
                            x1 = A[2*h1]
                            x2 = A[2*h2]
                            if ((y >= y1 and y <= y2) or (y >= y2 and y <= y1)) \
                               and ((x >= x1 and x <= x2) or (x >= x2 and y <= x1)):
                                print(y,x1,x2)
                                print(x,y1,y2)
                                return True
    return False
```

▷ *Solution 4*

```
def increasing_or_decreasing(A):
    inc = 0
    j = 0
    for i in range(1,len(A)):
        if A[i] > A[i-1]:
            if i - j > inc:
                inc = i - j
        else:
            j = i
    dec = 0
    j = 0
    for i in range(1,len(A)):
        if A[i] < A[i-1]:
            if i - j > dec:
                dec = i - j
        else:
            j = i
    if inc > dec:
```

```
        return 'increasing'
    elif dec > inc:
        return 'decreasing'
    elif inc == 0:
        return 'flat'
    else:
        return 'equal'
```