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

  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 `del A[len(A)−1]`
  - read the length, as in `n = len(A)`
  - shrink to a given length, as in `del A[length:]`
  - sort in-place as in `A.sort()`

  You may use `for` iterations as follows:

  - iteration over the elements in a sequence, as in `for a in A:`
  - range iteration, as in `for i in range(10):`

  You may define classes but only with a single, constructor method `__init__(self ,...)`

  You may not use any function or object or method or module except for the types and methods and functions from the standard library or built-in types listed above, namely `append()`, `len()`, `print()`, `range()`, `sort()`, `__init__()`.
- If an exercise requires you to analyze the complexity of an algorithm, 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 comments in the code.
- Submit each file through the iCorsi system.

►**Exercise 1.** Consider a binary search tree implemented in Python with the following node class:

```
class Node:
    def __init__(self,k):
        self.left = None
        self.right = None
        self.key = k
```

In a source file `ex1.py` write a Python function bst_range_weight(T,a,b) that takes a well-balanced binary search tree T (where T is the root node of the tree) and two keys a and b, with $a \leq b$, and returns the number of keys in T that are between a and b. Assuming there are m such keys, then the algorithm should have a complexity of $O(m) + o(n)$ for a tree of size n. Analyze the complexity of bst_range_weight(T,a,b). *(10')*

►**Exercise 2.** Let (a, b) represent an interval (or range) of values x such that $a \leq x \leq b$. Consider an array $X = [(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)]$ of n pairs of numbers representing n intervals (a_i, b_i).

Question 1: In a source file `ex2.py`, write a Python function intervals_union(X) that takes an array X representing n intervals, and returns a minimal set of intervals representing the to *union* of all the intervals in X. Notice that the union of two disjoint intervals can not be simplified, but the union of two overlapping intervals can be simplified into a single interval. For example, a correct solution for the simplification of $X = [(3, 7), (1, 5), (10, 12), (6, 8)]$ is $X = [(10, 12), (1, 8)]$. Analyze the complexity of your implementation of intervals_union(X). *(30')*

Hint: recall that in Python a pair (1,3) is simply a sequence (or "tuple") of two elements. So, given an array X=[(3,70),(1,5),(10,12),(6,8)], you can access the i-th interval as X[i] as a tuple, and then the beginning and end of that interval with X[i][0] and X[i][1], respectively.

Question 2: In the same source file `ex2.py` write a Python function fast_intervals_union(X) that simplifies the given array just like intervals_union(X) but with a $O(n \log n)$ complexity. If your implementation of intervals_union(X) already has an $O(n \log n)$ complexity, then you may use it directly to implement fast_intervals_union(X). *(20')*

►**Exercise 3.** Consider the following algorithm ALGO-X(A) that takes an array A of numbers: *(20')*

ALGO-X(A)

for $i = 3$ **to** $A.length$
    for $j = 2$ **to** $i - 1$
        for $k = 1$ **to** $j - 1$
            if $|A[i] - A[j]| == |A[j] - A[k]|$
            or $|A[i] - A[k]| == |A[k] - A[j]|$
            or $|A[k] - A[i]| == |A[i] - A[j]|$
                return TRUE
return FALSE

Analyze the complexity of ALGO-X and write an algorithm called BETTER-ALGO-X(A) that is functionally equivalent to ALGO-X(A) (for all A) but with a strictly better asymptotic complexity than ALGO-X(A). Write BETTER-ALGO-X as a Python function better_algo_x(A,k) in a source file called `ex3.py`. Analyze the complexity of better_algo_x(A,k).

►**Exercise 4.** Write an in-place partition algorithm called MODULO-PARTITION(A) that takes an array A of n numbers and changes A in such a way that (1) the final content of A is a permutation of the initial content of A, and (2) all the values that are equivalent to $0 \mod 10$ precede all the values equivalent to $1 \mod 10$, which precede all the values equivalent to $2 \mod 10$, etc. For example, with an input array $A = [7, 62, 57, 12, 39, 5, 8, 16, 48]$, a correct run might change A (in-place) to $A = [12, 62, 5, 16, 7, 57, 8, 48, 39]$. *(30')*

Write MODULO-PARTITION as a Python function `modulo_partition(A)` in a source file called `ex4.py`. Analyze the complexity of `modulo_partition(A)`.

►**Exercise 5.** In a source file `ex5.py` write a function `is_pithagorean_triple(a,b,c)` that, given three integers representing the sides of a triangle, returns `True` if a, b, and c identify a right triangle. Analyze the complexity of `is_pithagorean_triple(a,b,c)`. *(10')*