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()`
  - sort with the `sorted()` function, as in `B = sorted(A)`

  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()`, `sorted()`, `__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.** Given a number k, a step-k sequence of length ℓ is a sequence of ℓ numbers $a_1, a_2, \ldots, a_\ell$ such that either $a_i = a_{i+1} + k$ for all pairs of adjacent elements a_i, a_{i+1}, or $a_i + k = a_{i+1}$ for all pairs of adjacent elements a_i, a_{i+1}. For example, the sequence $2, 3.5, 5, 6.5, 8$ is a step-1.5 sequence, and $7, 4, 1, -2$ is a step-3 sequence. *(20')*

In a source file `ex1.py` write a python function called maximal_step_k_length(A,k) that takes a sequence of numbers A, and a number k, and returns the maximal length ℓ such that there is at least one contiguous sequence of elements in A that form a step-k sequence. You solution must have a time complexity $O(n)$, where n is the length of A.

For example, maximal_step_k_length([2,4,5,6,8,6,4,2,0,2,4,6,10,3,1],2) must return 5.

►**Exercise 2.** Your sport watch is equipped with an altitude sensor that, every second, measures your altitude in meters. Given an array $A = [a_1, a_2, \ldots, a_n]$ of n consecutive altitude measurements, you want to determine whether you had a high-power run. A high-power run occurs when there is a certain total altitude gain over a period of time, where the total altitude gain is the sum of all altitude gains (positive altitude variations) over that period. For example, the sequence of measurements $10, 10, 12, 11, 10, 11, 12$ corresponds to a total altitude gain of 4 meters ($10, 12$ and then $10, 11, 12$). *(30')*

In a source file `ex2.py` write a Python function called high_power_run(A,h,t) that takes a vector A of altitude measurements (measured consecutively every second), an altitude gain h, and a time limit t, and returns True if A indicates a steep climb of at least h meters in at most t seconds, or False otherwise. Your solution must have a complexity $O(n)$.

For example, high_power_run([10,6,1,3,2,1,3,4,6,5,6,4,3,4],6,5) must return True, because the measurememts $1, 3, 4, 6, 5, 6$ indicate a total gain of 6 meters in 5 seconds. However, high_power_run([10,6,1,3,2,1,3,4,6,5,6,4,3,4],6,4) must return False, because there is no total gain of at least 6 meters in 4 seconds.

►**Exercise 3.** An array $A = [a_1, a_2, \ldots, a_n]$ of numbers is said to be in "peak" order if $a_i \geq a_{i-1}$ for all $1 < i \leq (n+1)/2$, and $a_j \geq a_{j+1}$ for all $(n+1)/2 \leq j < n$. In essence, A is in peak order when its first half is in ascending order while the second half is in descending order. In a source file `ex3.py`, write a Python function called peak_order(A) that takes an array of numbers A and reorders its elements into a peak order. peak_order(A) must change the array A *in-place*, and must run in $O(n \log n)$ time. *(20')*

►**Exercise 4.** A *left-rotation* of an array A is defined as a permutation of A such that every element is shifted by one position to the left except for the fist element that is moved to the last position. For example, with $A = [1, 2, 3, 4, 5, 6, 7, 8, 9]$, a *left-rotation* would change A into $A = [2, 3, 4, 5, 6, 7, 8, 9, 1]$.

Question 1: In a source file `ex4.py` write an algorithm rotate(A,k) that takes an array A and performs k left-rotations on A. The complexity of your algorithm must be $O(n)$, which means that the complexity must not depend on k. *(10')*

Question 2: In the same source file `ex4.py` write a function rotate_inplace(A,k) that takes an array A and, in $O(n)$ steps, performs k left-rotations *in-place*. In-place means that rotate_inplace(A,k) may not use more than a constant amount of extra memory. If your implementation of rotate(A,k) is already in-place, then you may use it directly to implement rotate_inplace(A,k). *(30')*

►**Exercise 5.** In a source file `ex5.py` write a function is_sorted(A) that returns True if A is sorted in either ascending or descending order. Analyze the complexity of is_sorted(A). *(10')*