All your algorithms must be written in pseudo code, and justified.

If an algorithm is correct but it does not have the required complexity, then you get half of the points

1) A vector `a[]` with n integer elements is cube-repetition-free if no element is the cube of another element, i.e., there are no indices i,j such that `a[i] = a[j]`^{3}. Propose an O(n*log(n))-time algorithm in order to decide whether a vector is cube-repetition-free. <u>You are not allowed to use `pow()` or other similar functions in your (peudo)code</u>! **/1**

2)
   a. Propose an algorithm in order to enumerate all permutations of {1,2,…,n} in O(n) space.
   <u>Ex</u>: for n = 3, the algorithm must output 1,2,3 1,3,2 2,1,3 2,3,1 3,1,2 3,2,1 (not necessarily in this order). **/1**
   b. Propose an O(n)-time algorithm which, given a vector `a[]` with n numbers and a permutation (encoded as a vector `b[]` of size n) shuffles the elements of a[] so that, for any i, element a[i] is written in position b[i] in the output.
   <u>Ex:</u> if a[] = [2,4,1,3,7] and b = [1,0,3,2,4] then the output must be [4,2,3,1,7]. **/1**
   c. Consider the following sorting algorithm: we enumerate all permutations and we shuffle the input vector `a[]` according to each permutation sequentially. We stop if (after we shuffled the data) the vector becomes sorted.
   What is its complexity? It is optimal? **/1**

3)
   a. Recall what the properties of a balanced binary research tree (AVL) are. **/1**
   In the remainder of this exercise, we assume to be given an implementation of AVL, which you can freely use in your (pseudo)code.
   b. Let us assume that each node in the AVL stores the number of nodes in its rooted subtree (say, in a local variable `v.order`, where v denotes the label of the node considered). Propose an O(1)-time algorithm in order to update this information after a left rotation (resp., after a right rotation). **/1**
   c. Using the previous question, explain how, given a number x, you can compute in O(log(n)) time the number of nodes with a smaller value than x in the AVL. **/1**

4) We consider a generic problem where we are given n functions `f1, f2, …, fn`. Given as input a positive number `x`, we must output (as quickly as possible) the value `max{ fi(x) : 1 <= i <= n }`.

   We assume for simplicity that all functions fi are linear functions with positive coefficients : `fi(t) = ai*t+bi`. In this situation, we can encode each function fi as an ordered pair (`ai,bi`) of two positive numbers.

   a. We say that `fi` is dominated by `fj` if `ai <= aj` and `bi <= bj`. In this situation, we always have `fi(x) <= fj(x)`. Propose an O(n*log(n))-time algorithm in order to remove all dominated functions. **/1**
   From now on, we assume that there is no dominated function.
   b. Let `fi, fj` be such that `ai < aj` (and so, `bi > bj`). Compute the value $x_{i,j}$ such that: if x <= $x_{i,j}$ then `fi(x) >= fj(x)`; otherwise (if x >= $x_{i,j}$) then `fi(x) <= fj(x)`. **/1**

c. Let `fi, fj, fk` be such that `ai < aj < ak`. If $x_{jk} <= x_{ij}$ then deduce from the previous question that we can safely discard the function `fj`. **/1**

d. Deduce from the above that after an O(n*log(n))-time pre-processing, we can answer any query in O(log(n)) time. **/1**