Algorithms in your answers can be either explained (in plain text), written in pseudocode or in a programming language amongst C++ or Python. It has no consequence on the notation.

Correctness and Complexity of each algorithm must be justified. If your answer is not justified, then you get at most half of the points. You may also receive up to half of the points if your algorithm is correct but sub-optimal.

The result of a question can be used to answer another question even if you did not manage to answer to it.

Course materials are allowed.

1) In what follows, let v be an n-size vector of integers, and let S be an n-size vector of empty stacks. Consider the following algorithm:

```
for i=0…n-1 do:
  compute the smallest index j s.t. all elements in stack S[j] are strictly larger than v[i]
  S[j].push(v[i])
done
```

   a. Propose an O(n*log(k))-time implementation of the above algorithm, where k denotes the number of *nonempty* stacks in S at the end of the procedure. **/1**
   -- Simpler variant: propose an O(n*log(n))-time algorithm. **/.5**

   We maintain the largest index k of a non-empty stack in some auxiliary variable (initially, k:=-1). For every $j < k$, we observe that the top element of S[j] is *smaller* than or equal to the top element of S[j+1]. Indeed, if it were not the case then the top element of S[j+1] could have been pushed in S[j], that contradicts the minimality of j+1. As a result, for each element v[i] we can proceed as follows:
   i) if k=-1, then j=0 (we now set k:=0). It takes O(1) time.
   ii) else, we compare v[i] to the top element e of S[k]. If $v[i] \geq e$, then j=k+1 (we now set k:=k+1). If takes O(1) time.
   iii) from now on, $k > -1$ and v[i] is less than the top element of S[k]. We compute by binary search the smallest index $j \leq k$ such that the top element of S[j] is larger than v[i]. It takes O(log(k)) time.

   b. An increasing sequence of v is a sequence of indices $i_1 < i_2 < \ldots < i_k$ such that $v[i_1] \leq v[i_2] \leq \ldots \leq v[i_k]$. Propose an O(n*log(n))-time algorithm for computing a *longest* increasing sequence in v. **/1**

   We apply the previous algorithm to insert all elements of v in stacks. Each time we insert an element e in S[i], for $i > 0$, the current top element of S[i-1] is smaller than or equal to e. Then, we also keep a pointer pred(e) from e to this element.
   In doing so, an increasing sequence of length k can be computed backward by starting from the top element of the last non-empty stack, then using the pointers pred to previous stacks. To show that no increasing sequence can have more than k elements, we first observe that all elements in a stack S[j] form a strictly decreasing sequence. In particular, for any increasing sequence of v, there can be at most one element of the sequence in each stack S[j]. The runtime is in O(n*log(k)), and so it is also in

O(n*log(n)).

c. Given k sorted lists, whose sizes sum to n, propose an O(n*log(k))-time algorithm for merging their n elements into one sorted list. **/1**

If k=1, then we are done. Otherwise, we partition the lists in two groups of k/2 lists, that are merged in sorted list L_1,L_2. We apply a classical merge procedure (``interclasare'') to L_1,L_2. The runtime of each recursive stage in the algorithm is in O(n), and the recursive depth is in O(log(k)). Therefore, the runtime is in O(n*log(k)).

d. Deduce from the above an O(n*log(k))-time algorithm for sorting a vector v, where k denotes the length of a longest increasing sequence. **/1**

We insert all elements of v in strictly decreasing stacks. As proved in question a), it can be done in O(n*log(k)) time. Then, we apply the procedure of question c) to the k nonempty stacks of S. It also takes O(n*log(k)) time. As proved in question b), k is the length of a longest increasing sequence of v.

2) A point is an ordered pair (x,y) of integers. The infinite distance between two points (x,y) and (x',y') is equal to max{|x-x'|,|y-y'|}. In what follows, let v be an n-size vector of points. The *diameter* of v equals the maximum distance between two points in v.

a. Given a point (x,y), let A,B,C,D partition the plane such that:
- all points in A have abscissa ≤ x and ordinate ≥ y;
- all points in B have abscissa ≥ x and ordinate ≥ y;
- all points in C have abscissa ≥ x and ordinate ≤ y;
- all points in D have abscissa ≤ x and ordinate ≤ y.

For a point (x',y') in A (resp., in B, C, D), show that we can rewrite what the infinite distance between (x,y) and (x',y') is, without using absolute values. **/1**

Case (x',y') in A. Then, the infinite distance is max{x-x',y'-y}.
Case (x',y') in B. Then, the infinite distance is max{x'-x,y'-y}.
Case (x',y') in C. Then, the infinite distance is max{x'-x,y-y'}.
Case (x',y') in D. Then, the infinite distance is max{x-x',y-y'}.

b. Deduce from the previous question that, after a pre-processing of v in O(n*log(n))-time, for any (x,y) we can compute in O(log(n)) time the *maximum* infinite distance between (x,y) and a point in v. – Hint: use a range tree. **/1**

Let us show how to compute the maximum infinite distance between (x,y) and a point in A. We construct 2-range trees T_1, T_2. In T_1, for each point (x',y') in v, we insert the point p_1=(x'+y',y'). In T_2, for each point (x',y') in v, we insert the point p_2=(x',x'+y'). It takes O(n*log(n)) time.

i) Consider the following range query on T_1:

compute min{ a-b | (a,b) s.t. a ≤ x+y and b ≥ y }.

It can be solved in O(log(n)). Its output is min{ x' | (x',y') in A s.t. x-x' ≥ y'-y }. Indeed, we have (a,b) = (x'+y',y'), and x'+y' ≤ x+y if and only if y'-y ≤ x-x'.

ii) Consider now the following range query in T_2:

compute max{ b-a | (a,b) s.t. a ≤ x and b ≥ x+y }.

It can be solved in O(log(n)). Its output is max{ y' | (x',y') in A s.t. x-x' ≤ y'-y }. Indeed, we have (a,b) = (x',x'+y'), and x'+y' ≥ x+y if and only if y'-y ≥ x-x'.

By combining i) and ii), we compute the maximum infinite distance between (x,y) and a point in A. We use a similar approach for points in B,C,D.

3) In what follows, let T be an n-node rooted tree, given with a heavy-path decomposition.

a. For each heavy-path P, let its elements be ordered by increasing levels (distance to the root). We insert all nodes in P in a self-balanced binary search tree T(P). Explain how, being given the trees T(P), the following queries can all be answered in O(log(n)) time:
- **head(v)**: outputs the head (node closest to the root) of the heavy-path containing v.
- **tail(v)**: outputs the tail (node furthest to the root) of the heavy-path containing v.
- **next(v)**: outputs the heavy child of v (i.e., the unique child of v in the same heavy-path), if it exists. **/1**

head(v): outputs the smallest element in T(P) (always go left).
tail(v): outputs the largest element in T(P) (always go right).
next(v): outputs the successor element to v in T(P). One way to do that is to split at v, then to compute the smallest element in the right subtree.

b. For every node v in T, let T_v be the subtree rooted at v and let $T_v' = T_v \setminus T_{next(v)}$ where we exclude the rooted subtree of its heavy child. We assume that each node v stores the size (number of nodes) of T_v'. Under this hypothesis, show that after pre-processing all the search trees T(P) in total O(n) time, we can answer in O(log(n))-time to the following type of queries:
- **size(v)**: outputs the size of the subtree rooted at v. **/1**

To each node v in P, we associate weight $w(v) = |T_v'|$. Then, by dynamic programming on T(P), each node v may store the sum of all weights w(x), for x in the subtree rooted at v *in T(P)* (not in T). It takes O(|P|) time per heavy-path P, and therefore O(n) time in total. Now, to answer to a query **size(v)**, let P be its heavy-path. We compute $\sum \{ w(u) \mid u \text{ in } P \text{ and } u.level \geq v.level \}$, that is indeed equal to the size of the subtree rooted at v in T. Furthermore, the latter is a range query, and as such we know how to solve it in O(log(n)) time.

c. Show that, if we insert a new leaf in T, then at most O(log(n)) edges need to modify their status (i.e., whether or not they are contained in some heavy-path) in order to update the heavy-path decomposition of T. *Important: you are not required to propose an efficient algorithm in order to find these edges!* **/1**

There are only O(log(n)) light edges (not on a heavy-path) on the path between the root and the new leaf. To update the heavy-path decomposition, it suffices to check whether any of these light edges becomes heavy (thus creating at most O(log(n)) new heavy edges, and also at most O(log(n)) new light edges).

d. Deduce from the above a dynamic data structure which maintains a tree T *and* a heavy-path decomposition, subject to the following operations:
- **insert(u,v)**: add a new leaf u with father node v
- **head(v)**: outputs the head of the heavy-path containing v
- **size(v):** outputs the size of the subtree rooted at v
- **lca(u,v):** outputs the least common ancestor of u and v

Complexity: O(log(n)) amortized per operation. **/1**
-- Simpler variant: $O(\log^2(n))$ amortized per operation. **/.5**

We store for each node its level. We also store each heavy-path in a splay tree. In doing so, we know how to answer to **head(v)** (see question a) and **size(v)** (see question b) in amortized O(log(n)).

Consider an operation **insert(u,v)**. We set u.level = v.level +1. If v was previously a leaf, then we also insert u in the same heavy-path as v. Then, while v is not the root of T, we proceed as follows:

i) let P be the heavy-path containing v. Set v = head(P).
ii) splay T(P) at v (v becomes the root of T(P)).
iii) if v is not the root of T then, let w be its parent and let P' be its heavy-path.
- compute size(v), size(w).
  Since v is the root of T(P), and w is the tail of P', it can be done in O(1).
- decide whether edge vw becomes heavy.
- if yes, then split T(P') at w, then join T(P'),T(P).
- v := w.

Remark: this is the same as an access operation in a link/cut tree. In particular, the amortized cost is in O(log(n)). By not using a splay tree, we could achieve $O(\log^2(n))$.

To answer to **lca(u,v)**, we slightly adapt an algorithm seen in class: while u and v are not in the same heavy-path P, let u'=head(u) and v'=head(v). Without loss of generality, u'.level ≥ v'.level. We replace u by the father node of u'. Again, this is essentially the same as two access operations in a link/cut tree, and therefore the amortized runtime is in O(log(n)).