1) Consider three sorted vectors u,v,w of equal size n. The value med(u,v,w) is the median of the 3n-size vector obtained from the concatenation of u,v,w.
   a. Show that we can compute med(u,v,w) in O(n) time.

      We create a 3n-size vector where we insert all elements of u,v,w. Then, we apply quickselect.

   b. Show that we can compute med(u,v,w) in $O(\log^2(n))$ time.

      Cut one vector, say w, in two halves w_L,w_R. Assign w_L to u and w_R to v. Intuitively, we would like to simulate the computation of med(x,y) where x,y are the sorted vectors containing all elements in $u \cup w_L$, $v \cup w_R$. However, we must do that without computing explicitly x,y (that would cost us O(n)). The algorithm works as follows:
      i) compute the medians med(x), med(y). If med(x) = med(y), then we are done.
      ii) otherwise, without loss of generality med(x) < med(y). We compute recursively med(x_R,y_L).
      For i), it suffices to compute med(x) = med(u, w_L) and med(y) = med(v, w_R). Both values can be computed in O(log(n)). For ii), we need to compute $u \cap x_R$ and $w_L \cap x_R$, resp. $v \cap y_L$ and $w_R \cap y_L$. It can be done in O(log(n)) by binary searching med(x) in u and w_L, resp. med(y) in v and w_R. Overall,we can simulate each step of the computation of med(x,y) in O(log(n)). And therefore the total runtime is in $O(\log^2(n))$.

   c. Show that we can compute med(u,v,w) in O(log(n)) time.

      For technical reasons, that shall become clearer in what follows, we consider the more general case where u,v,w may be of different sizes.

      We first compute med(u), med(v), med(w) in O(1) each. Without loss of generality, $med(u) \leq med(v) \leq med(w)$. If med(u) = med(w), then we are done as it is also med(u,v,w). Otherwise, med(u) < med(w). We removed the min(u.size,w.size)/2 smallest elements from u and the min(u.size,w.size)/2 largest elements from w, leaving v unchanged. Note that it means that even if we start from equal-size vectors, our recursive algorithms may have to deal with different-size vectors at later recursive steps.

      The recursion stage stops when one of the vectors, say u, has too few remaining elements (at most two). However, we can determine in O(log(n)) the positions where to insert the elements of u in v, by performing at most two binary searches. In doing so, we can now efficiently simulate the sorted vector v' whose elements are in $u \cup v$. We continue computing med(v',w) in O(log(n)).

2) Consider a fixed n-size rooted tree T. Recall that the distance between two nodes u and v, denoted in what follows d(u,v), is the number of edges on the uv-path in T. The *total sum* of a node v, denoted in what follows D(v), is equal to $\sum_u d(u,v)$. Finally, a *median* is a node v that minimizes D(v). Show that we can compute a median in O(n).

   We will do something stronger: computing D(v) for every node v. In doing so, we can output all the medians in additional O(n). For that, we adopt a classical dynamic

programming approach. More precisely, for every node u we compute: the size s(u) of its rooted subtree, and the sum p(u) of all distances d(u,v), for v a descendant of u. If u is a leaf, then s(u) = 1 and p(u) = 0. Otherwise, let $x_1, x_2, ..., x_k$ be the children of node u. Then, $s(u) = 1 + \sum_i s(x_i)$ and $p(u) = \sum_i [p(x_i)+s(x_i)]$. Note that for the root r of the tree, p(r) = D(r). Finally, we compute for every node u the sum q(u) of all distances d(u,v) between u and a node v that is *not* in its subtree. This can be done by backward dynamic programming. For the root r, q(r) = 0. Otherwise, let y be the father of node u. We have q(u) = [q(y) + (n-s(y))] + [(p(y) – p(u) – s(u)) + (s(y)-s(u))]. Here, the first term represents the contribution of all nodes not in the subtree rooted at y, while the second term represents the contribution of all nodes in the subtree rooted at y but not in the subtree rooted at u. We are done as D(u) = p(u) + q(u) for every node u.

3) Consider the following two operations allowed on a tree T:
 - insert(u,v): insert a new node u with parent node v
 - lca(u,v): compute the lowest common ancestor of u and v.
An insertion can be done trivially in O(1). Propose an implementation so that both operations can be done in worst-case O(log(n)), n being the number of nodes.

For every node u, we maintain its level (distance to root) and pointers p(u,j), $j=0....\lfloor ln(u \rightarrow level) \rfloor$ so that p(u,j) denotes the 2^j closest ancestors to u.

Note that, after an insert(u,v) operation, we can set u->level = v->level +1, we set p(u,0) = v, and otherwise p(u,j) = p(p(u,j-1),j-1). Therefore, theses pointers can be initialized in O(log(n)).

To compute lca(u,v), without loss of generality u->level ≤ v->level. If this inequality is strict, then we replace u by its ancestor on the same level as v, that can be computed in O(log(n)). Thus, from now on we assume u,v to be on the same level L.
We call a procedure lca(u,v,k), where k denotes an upper-bound on the distance (number of levels) between u,v and their lca. Initially, k=L.
If k=0 then u=v, and we are done.
Else, if k=1, then either u=v or u,v have a common father node, that is their lca.
Otherwise, let $j = \lfloor \log(k) \rfloor$. We compute w=p(u,j-1) and w'=p(v,j-1). If w=w', then we call lca(u,v,2^{j-1}). Otherwise, we call lca(w,w',k-2^{j-1}).
Observe that $k \geq 2^j > k/2$. Therefore, $2^{j-1} \leq k/2$ whereas $k-2^{j-1} \leq k - k/4 = 3k/4$. In particular, the recursion depth of this algorithm is at most O(log(n)).

4) Let T be an (*unknown*) n-node binary tree whose nodes are labeled from 0 to n-1. We are given a preorder and an inorder of T, as vectors vec_p and vec_i (permutations of {0,1,...,n-1}). Show that we can construct T in O(n).

We first construct a Hash-table H, whose keys are the nodes of T, and such that to each node we associate its respective positions in vec_p and vec_i. It can be done in expected O(n), simply by traversing these two vectors.

The root of T is always $vec_p[0]$. Let j be such that $vec_p[0] = vec_i[j]$ (computed in expected O(1) from the Hash-table). Then, in vec_i, the nodes in the left subtree are between position 0 and j-1, whereas the nodes in the right subtree are between positions j+1 and n-1. Equivalently, the subvector $vec_i[0..j-1]$ is an inorder of the left subtree and the subvector $vec_i[j+1...n-1]$ is an inorder of the right subtree. We also have that

$vec_p[1...j]$ is a preorder of the left subtree and that $vec_p[j+1...n-1]$ is a preorder of the right subtree. As a result, we can apply our algorithm recursively in order to construct the left and right subtrees of T. Overall, the total expected runtime is O(1) per node, hence in expected O(n).

Remark: we can avoid using a Hash-table here. In fact, we can just create a vector map such that, if map[i] = j, then $vec_p[i] = vec_i[j]$. In doing so, the deterministic complexity of our algorithm is in O(n) (i.e., it is no more an expected runtime).

5) An ``abstract list'' is an indexed data structure that supports the following operations: insertion, deletion, read and write at a given index. The linked-list is one possible implementation of ``abstract list'', with worst-case O(n) for some operations. Propose a faster implementation where every operation can be done in worst-case O(log(n)).

We use a self-balanced binary search tree implementation (say, an AVL). If a node in the tree represents the element at position i in the list, then all nodes in its left subtree are at some positions j < i, whereas all nodes in its right subtree are at some positions j > i. For each node, we further retain the size sz of its rooted subtree.

Read/Write at position i. Let r be the root of the tree. If there are k nodes in its left subtree, then node r is in position k in the list. In particular, if k=i then we are done. Else, if k > i then we search for node i to the left. Otherwise, we search for node i-(k+1) to the right. -- Note that we already saw a similar idea in class for computing order statistics. – Overall, the runtime is in O(log(n)).

Delete element of position i: we acess to this element in O(log(n)) – see the Read/Write operation – and then we remove this element from the tree. All rotations to maintain the AVL property also preserve the relative order (positions) of all elements in the list. We just have to recompute the size of their subtrees for O(1) nodes after each rotation.

Insertion of a new element at position i: we insert a new element in the AVL. For all nodes visited during this insertion, we can compute on the fly their respective positions in the list – see the Read/Write operation --. Therefore, we can compare the position of each visited element to the position i of the new element in O(1). All rotations to maintain the AVL property also preserve the relative order (positions) of all elements in the list. We just have to recompute the size of their subtrees for O(1) nodes after each rotation.

6) Show that we can simulate a stack with three queues in amortized $O(\sqrt{n}*\log(n))$ per operation.

We maintain the at most $\sqrt{n}$ latest element in queue q_1, and all other elements in queue q_2. The queue q_3 is empty, it is just used in order to perform efficiently the push/pop operations.

Push. We add the new element in q_1 (using enqueue). If q_1.size is greater than $\sqrt{(n+1)}$ after the push, then we remove the earliest element in q_1 (dequeue) and we add it in q_2 (enqueue).

Pop. If q_1 is empty, then we reverse the order of all elements in q_2, using q_3 as an

intermediate queue. Doing so, we can extract all latest $\sqrt{n}$ elements and enqueue them in q_1. We reverse back all elements in q_1, using q_3 as an intermediate queue. Then, we reverse back all elements in q_2, still using q_3 as an intermediate queue. From now on, q_1 is not empty. We reverse all elements in q_1 using q_3, we dequeue the latest element, then we reverse back all elements.

The worst-case complexity of a push is O(1), but the worst-case complexity of a pop is O(n*log(n)) – time needed for reverting all n elements in q_2. However, consider the potential function $(n_2 - n_1^2)*\log(n)$, where $n_i = q_i$.size. After a push, n_2 increases by at most 1 and n_1^2 decreases by at most $2* n_1 +1= O(\sqrt{n})$. Therefore, the amortized complexity of a push is $O(\sqrt{n}*\log(n))$. For a pop, there are two cases:
- Case q_1 is not empty: n_1^2 decreases by at most $2* n_1 +1= O(\sqrt{n})$. Therefore, the amortized complexity is $O(\sqrt{n}*\log(n))$.
- Case q_1 is empty: n_2 decreases by $O(\sqrt{n})$, and n_1^2 increases by O(n). Therefore, the potential decreases by O(n*log(n)), that compensates the O(n*log(n))-time reversal of all n elements in q_2.

Remark: a more intuitive explanation about the amortized time bound is that we need at least $\sqrt{n}$ pop in order to empty q_1. Therefore, whenever we pay O(n*log(n)) for reverting q_2, this cost can be divided by $\sqrt{n}$.

7) Consider an n-size vector v. Show that we can pre-process v in $O(n*\sqrt{n}*\log(n))$ so that the following types of queries can be answered in $O(\sqrt{n}*\log(n))$.
q(a,b,c,d): under the assumption $a \leq b < c \leq d$, compute the number of pairs (i,j) where i is in [a;b], j is in [c;d], and v[i] = v[j].

We use Mo's trick. More precisely, we partition v in $\sqrt{n}$ –size blocks. We construct a $\sqrt{n}$ x $\sqrt{n}$ matrix A such that A[r,s], r < s, denotes the number of pairs (i,j) such that i is in block r, j is in block s and v[i] = v[j]. Each entry of the matrix A can be computed in $O(\sqrt{n}*\log(n))$ by using a variant of Merge sort (see exercise 7 in seminar 1). Then, let B be such that B[r,s] is the sum of all entries A[r,s'], for s' between r+1 and s (``partial sums''). Let also C be the n x $\sqrt{n}$ matrix such that C[r,s] denotes the number of elements equal to v[r] in block s. Note that each entry of C is computed in O(log(n)) using binary search. Finally, let D be such that D[r,s] is the sum of all entries D[r,s'], for s' between 0 and s (partial sums). Overall, this pre-processing stage takes $O(\sqrt{n}*\log(n))$ time.

To answer a query, we first consider each block K_r fully between a and b. Let $K_s,K_{s+1},...,K_{s+t}$ be all blocks fully between c and d. We use matrix D in order to compute in O(1) the number of pairs (i,j) with i in K_r and j in one of $K_s,K_{s+1},...,K_{s+t}$. Since there are $O(\sqrt{n})$ blocks to be considered, the total runtime is in $O(\sqrt{n})$. Then, for each element v[j] of K_{s-1} that is in [c;d], resp. for each element v[j] of K_{s+t+1} that is in [c;d] (blocks not fully contained in the interval), we use matrix B in order to compute the number of pairs (i,j) such that i is contained in some block fully between a and b. We do symmetrically for the elements in a block not fully in [a;b] and the blocks fully in [c;d]. It takes $O(\sqrt{n})$ time in total because there are only $O(\sqrt{n})$ such elements. Finally, we directly compare the $O(\sqrt{n})$ extremal elements in [a;b] (those in a block not fully contained in this interval) with the $O(\sqrt{n})$ extremal elements in [c;d], that can be done in $O(\sqrt{n}*\log(n))$.