1) Let v be an n-size vector. The *q-quantiles* of v are its kn/q order statistics, for k = 1...q.

   a. Show that we can compute all q-quantiles in O(n*min(q,log(n))).

      We can sort the vector, that takes O(n*log(n)) time. Another possibility is to call q times (one for each quantile) the algorithm quickselect, and then the runtime is in O(n*q).

   b. Show that we can compute all q-quantiles in O(n*log(q)) time. Discuss about the optimality of this runtime.

      We call quickselect for computing the kn/q order statistic, for k = q/2. It takes O(n) time. In doing so, we also partitioned v in two subvectors v_L,v_R, that contain the elements smaller and larger than the kn/q order statistic, respectively. Now, half of the remaining q-quantiles is in v_L and the other half in v_R. Note that it implies that by applying recursively our algorithm to v_L,v_R, we can next compute two more q-quantiles, then four more q-quantiles, and so forth. In particular, the recursive depth until we computed all q-quantiles is in O(log(q)). Since at each recursion level, all vectors considered are disjoint subvectors of v, the runtime at each recursion level is in total O(n).

      This is essentially optimal as, for q=n, computing all q-quantiles in order reduces to sorting.

      <u>Remark:</u> the same approach works for computing any q order statistics (not necessarily the q-quantiles) in O(n*log(q)) time.

2) Consider a *sorted* n-size vector v and a *sorted* m-size vector u. The median of u and v, denoted in what follows by med(u,v), is the median of the (n+m)-size vector obtained from concatenating u and v together.

   a. Show that med(u,v) can be computed in O(n+m) time.

      We just concatenate u and v into one (not sorted) vector w, and then we apply to it quickselect. Note that this approach works even if u and/or v are not sorted...

   b. Show that if m=n, then med(u,v) can be computed in O(log(n)).

      Let med(u), med(v) be the respective medians of u,v. We can compute both values in O(1) because we assume u and v to be sorted. If med(u) = med(v) = k, then we also have med(u,v) = k. Otherwise, without loss of generality med(u) < med(v). As we have n/2 elements from u, plus > n/2 elements from v, greater than med(u), no element in the left half of u can be equal to med(u,v). Similarly, no element in the right half of v can be equal to med(u,v). So, let u_R,v_L be respectively the right half of u and the left half of v. We can apply recursively our algorithm to u_R,v_L since one can easily check that med(u,v) = med(u_R,v_L). We stop the recursion as soon as there remains at most two elements in each vector, in which case we can solve the problem in O(1). Since the size of the input vectors is halved at each recursion stage, the total runtime is in O(log(n)).

   c. Show that med(u,v) can be computed in O(log(max(n,m))).

Without loss of generality, $m \neq n$. Let us consider the more general problem of computing the kth order statistic of u and v (for the median, $k = (n+m)/2$).

During a first stage of the algorithm, we essentially apply the previous approach of 2.b. More precisely, we first compute in O(1) the medians med(u), med(v).
<u>Case med(u) = med(v).</u> If $k=(n+m)/2$, then we are done (we output med(u)). If $k < (n+m)/2$, then we recurse on u_L, v_L with $k'=k$. If $k > (n+m)/2$, then we recurse on u_R, v_R with $k'=k-(n+m)/2$.
<u>Case med(u) < med(v).</u> If $k=(n+m)/2$, then we recurse on u_R, v_L with $k'=k-m/2$. Note that we may not have $med(u,v) = med(u_R,v_L)$ in general, as $m \neq n$. If $k < (n+m)/2$, then we recurse on u, v_L with $k'= k$. If $k > (n+m)/2$, then we recurse on u_R, v with $k'= k - m/2$.
<u>Case med(v) < med(u).</u> Similar to the previous case.

This above approach works until we can no longer make a recursive call. Indeed, when this happens, the smallest vector (say, u) stays with at most two elements, but the other vector may still have a non-constant (not even logarithmic) number of remaining elements, that prevents us from solving the problem in O(1) during this last stage. So, instead, we do binary search for each of the (at most two) elements in u to determine where they should be inserted in the sorted vector v. In doing so, we know by how much we should shift the current position of the kth element in v in order to reach the kth order statistic of u,v. Note that we can shift either left or right (depending on where we should insert elements from u in v) and that we must shift by at most 2 positions in the vector.

Overall, the first stage where we can do recursive calls lasts O(log(m)+log(n)) steps. Indeed at each stage one of u,v has its size halved. The at most two binary searches of the last stage take O(log(max{n,m})) time. The final runtime is also in O(log(max{n,m})).

d. Show that med(u,v) can be computed in O(log(min(n,m))).

Without loss of generality, $n < m$.

We slightly modify the previous approach from 2.c. First we propose a faster (but slightly less general) recursive stage than for 2.c. Let med(u), med(v) be the respective medians of u,v. If med(u) = med(v) = k, then we also have med(u,v) = k. Otherwise, without loss of generality med(u) < med(v). Let u_R, v_L be respectively: vector u with its n/2 smallest element removed, and vector v with its n/2 biggest elements removed. Note that v_L is the lower half of v but that u_R is not the upper half of u. Furthermore, we can apply recursively our algorithm to u_R, v_L since one can easily check that $med(u,v) = med(u_R,v_L)$. *Indeed, we always remove the same number of elements to the left and to the right*. We stop the recursion as soon as there remains at most two elements in v. It takes O(log(n)) recursive calls, and so, O(log(n)) time.

Now, during the final stage recall that we need to shift by at most two positions in order to reach med(u,v) from med(u). Therefore, we do not need to know exactly where to insert the elements from v in u, but just to compare these at most two elements to the 5 middle elements of vector u. This can be done in

$O(1)$ time, that is much faster than binary search. Then, the total runtime is dominated by the first stage of the algorithm (where we still can do recursive calls), that is in $O(\log(n))$.

3) Consider two n-size vector v and w, such that all values in w are non-negative, and let $W = \sum_i w[i]$. A weighted median of v,w is a value x such that:
   - $\sum \{ w[j] \mid v[j] \leq x \} \leq W/2$
   - $\sum \{ w[j] \mid v[j] > x \} \leq W/2$

   a. Propose a randomized algorithm for computing a weighted median, that achieves expected $O(n)$ time complexity.

   We just use (randomized) quickselect. Each pivot (randomly selected) partitions the vector v in left/right subvectors v_L,v_R of smaller/larger elements, in $O(n)$. In doing so, we can compute the respective total weights W_L,W_R for both subvectors. If $\max\{W_L,W_R\} \leq W/2$, then our pivot is a weighted median. Else, if $W_L > W/2$, then we recurse on v_L (increasing the weight of its last element by $W - W_L$). Otherwise, we recurse on v_R (increasing the weight of its first element by $W - W_R$). The expected runtime of the algorithm follows the same recursive equation as for the classic version of quickselect (for unweighted median), and therefore it is also in $O(n)$.

   b. Propose a worst-case $O(n)$-time algorithm for computing a weighted median.

   Same as before but we use the deterministic version of quickselect (``median of medians'').

4) Let M be an n x n matrix whose every column and row is sorted. Propose an $O(k*\log(k))$-time algorithm to compute the kth smallest element in the matrix.

We actually do something stronger: we compute all k smallest elements in the matrix sequentially. For that, at every step i, we maintain a set S_i of elements in the matrix (along with their coordinates in the matrix) such that the ith smallest element of the matrix is always in S_i; in fact, it is always the smallest element in S_i. Initially, S_1 just contains the element M[0,0], that is indeed the minimum element because we assume each row and column to be sorted. Then, to obtain S_{i+1} from S_i, we extract from S_i its smallest element v_i, that is by construction the ith smallest element from the matrix; let $v_i = M[a_i,b_i]$, then we replace v_i by $M[a_i+1,b_i]$ and $M[a_i,b_i+1]$ (if they exist). Note that $M[a_i+1,b_i]$ is the next element on the same (sorted) row as v_i, while $M[a_i,b_i+1]$ is the next element on the same (sorted) column as v_i.

If we naively stores S_i as a set, then the ith step of the algorithm would run in $O(|S_i|)$ time. By induction, $|S_i|$ is at most i, and therefore the total runtime would be linear in $1+2+...+i...+k = O(k^2)$. Instead, all elements of S_i are stored in a self-balanced binary search tree. In doing so, each step runs in $O(\log(k))$ time.

<u>Remark</u>: here, it would be more natural to store S_i in a heap rather than in a binary search tree. Indeed, what we are doing here, but without telling it (because heaps were not presented in class at that point), is simulating a priority queue by using a self-balanced binary search tree.

5) In what follows, we are discussing about the optimal time for constructing a binary search tree (not necessarily balanced), being given n elements to be stored in this tree. We already know how to do this in O(n*log(n)), e.g. by using AVL.

   a. Show that if we randomly insert these n elements in a binary search tree (using a naive implementation), then the expected runtime is also in O(n*log(n)).

      Let us assume that we choose the kth order statistic as the root (first element inserted in the tree). This happens with probability 1/n. Then, k-1 elements are to be inserted to its left and the others to the right. In particular, all remaining n-1 elements are to be compared with the root when we insert them in the tree, and then we are reduced to constructing separately (at random) binary research trees for k-1 and n-k elements, respectively. Therefore, the expected construction time satisfies $C(n) = n-1 + 1/n * \sum_k [C(k-1) + C(n-k)] = n-1 + 2/n*\sum_k C(k-1)$.

      We conclude by using a classical trick in average analysis. More precisely, we have $(n+1)*C(n+1) – n*C(n) = n*(n+1) – n*(n-1) + 2*C(n) = 2*(n+C(n))$. Equivalently, $(n+1)*C(n+1) = (n+2)*C(n) + 2n$. Let $K(n) = C(n)/(n+1)$. We obtain $K(n+1) = K(n) + (2n)/((n+2)*(n+1)) = K(n) + O(1/n)$. By induction, $K(n) = O(\sum_k 1/k) = O(\log(n))$. As a result, $C(n) = O(n*\log(n))$.

   b. Show that if we are given n *sorted* elements, then we can construct a balanced binary search tree in O(n) time.

      We choose the median as the root of the tree, and then we apply the algorithm recursively to the elements that are smaller/larger than it to construct the left/right subtrees. Since the elements are sorted, finding their median and partitioning the remaining elements (depending on whether they are smaller or larger) can be done in O(1). Therefore, the construction time satisfies $C(n) = O(1) + 2*C(n/2) = O(n)$.

   c. Show that in general, the optimal runtime for constructing a binary search tree is in O(n*log(n)). – Hint: Prove that you can use this tree in order to sort in less than O(n*log(n)).

      If we postorder the nodes in a binary search tree, then all values are sorted by increasing values. A postorder can be computed in O(n). Therefore, the time to sort n elements is the time to construct a binary search tree + O(n). It follows that constructing a binary search tree on n elements requires $\Omega(n*\log(n))$.

6) Consider an n-size balanced binary search tree T. The T-floor of an integer x is the greatest value stored in the tree that is smaller than or equal to x. Similarly, the T-ceil of an integer x is the least value stored in the tree that is greater than or equal to x. Show that we can compute the floor and the ceil of any value x in O(log(n)).

   We search for x in T in O(log(n)). If x is stored in T, then both ceil(x) and floor(x) are equal to x. Otherwise, we compute ceil(x) and floor(x) during the search in T, as follows. Each time during the search we go on the left subtree of some node u, ceil(x) = u.value. Each time we go on the right subtree of some node u, floor(x) = u.value.

7) Consider an n-size balanced binary search tree T. Show that we can pre-process T in O(n) such that the following types of queries can be answered in O(log(n)):

a. $q_1(x,y)$: what is the number of nodes u such that $x \leq u.value \leq y$?
b. $q_2(x,y)$: compute $\max\{ u.value \mid x \leq u.value \leq y \}$
c. $q_3(x,y)$: compute $\sum\{ u.value \mid x \leq u.value \leq y \}$

These are typical applications of range trees (recall that 1-range tree = balanced binary search tree). For each node u, we compute:

- $p_1(u)$: size of its subtree
- $p_2(u)$: maximum value stored in its subtree
- $p_3(u)$: sum of all values in its subtree.

It can be done in O(n) by dynamic programming.

Now, to answer a query $q_3(x,y)$, we start computing nodes u and v such that: u.value = ceil(x), v.value = floor(y). As shown in the previous exercise, it can be done in O(log(n)). Note that all values between x and y in T must be in fact between u.value and v.value. Let w = lca(u,v). Since T is balanced, it can be computed naively also in O(log(n)). Furthermore, note that $u.value \leq w.value \leq v.value$.

- let $u_0=u,u_1,\ldots,u_k=w$ be the uw-path in T (in the left subtree of w). We start summing all values u_i.value between x and y. If furthermore u_i is the left child of u_{i+1}, then all values in the right subtree of u_{i+1} must be also between x and y; in this situation, we add $p_3(u_{i+1}\text{->right})$ to the total.

- let $v_0=v,v_1,\ldots,v_k=w$ be the vw-path in T (in the right subtree of w). We start summing all values v_i.value between x and y. If furthermore v_i is the right child of v_{i+1}, then all values in the left subtree of v_{i+1} must be also between x and y; in this situation, we add $p_3(v_{i+1}\text{->left})$ to the total.

The generalizations to the other two types of queries are straightforward.

8) Recall that in a 2-range tree (as seen in class), all values stored are bidimensional (2 coordinates –x and -y). These values are stored in balanced binary search tree T_x, using for their keys only the –x coordinates. Each node u in the tree stores a balanced binary search tree $T_y[u]$ where all values in its rooted subtree are stored, this time using for their keys the –y coordinates. The construction time is in $O(n*\log^2(n))$.

We consider the following modified construction of 2-range trees. Now, each node u of T_x stores all values in its subtree in an *array* $A_y[u]$ (no more a binary search tree), sorted by increasing –y coordinates (breaking ties using –x coordinates). If u is not a leaf, then let also u_L,u_R be its children nodes. For each element $A_y[u][i]$ we memorize the least indices i_L,i_R such that $A_y[u][i] \leq A_y[u_L][i_L]$ and $A_y[u][i] \leq A_y[u_R][i_R]$. Similarly, we memorise the largest indices j_L,j_R such that $A_y[u][i] \geq A_y[u_L][j_L]$ and $A_y[u][i] \geq A_y[u_R][j_R]$.

a. Show that the construction time can be improved to $O(n*\log(n))$.

We copy all values in two separate arrays, one B_x sorted by increasing –x coordinates, and one B_y sorted by increasing –y coordinates. It takes $O(n*\log(n))$. Then, we do as follows: We choose the median (x^*,y^*) of B_x as root for T_x. Note that $A_y[x^*,y^*] = B_y$. Let $B_{L,x}$, $B_{R,x}$ be the values with smaller/larger –x coordinate than the root (sorted by increasing –x coordinates).

By scanning B_y once, we also construct in O(n) time vectors $B_{L,y}$, $B_{R,y}$ of all values in $B_{L,x}$, $B_{R,x}$ sorted by –y coordinates. Then, for each value $B_y[i]$, we compute the indices $i_L[i]$, $i_R[i]$ and $j_L[i]$, $j_R[i]$ in $B_{L,y}$, $B_{R,y}$ by using the following formulas:

i) if $B_y[i] = B_{L,y}[i']$ for some i' then $i_L[i] = i'$.

ii) Otherwise $i_L[i] = i_L[i+1]$.

We have similar formulas for all other three indices. As a result, all these values can be computed in O(n) by dynamic programming.

We end up recursing on the disjoint pairs $B_{L,x}$, $B_{L,y}$ and $B_{R,x}$, $B_{R,y}$.

Recall that the initial sorting stage takes O(n*log(n)). The cost C(n) of the recursive stage satisfies C(n) = O(n) + 2*C(n/2). Hence, C(n) = O(n*log(n)).

b. Let each value (x_i,y_i) be assigned some weight $w(x_i,y_i)$. Show that after a pre-processing in O(n*log(n)) time, we can answer in O(log(n)) to the following type of queries $q(x_{min},x_{max},y_{min},y_{max})$: output max{ $w(x_i)$ | $x_{min} \leq x_i \leq x_{max}$ and $y_{min} \leq y_i \leq y_{max}$ }.

<u>Pre-processing</u>: at every node u of T_x, let $W_y[u] = w(A_y[u])$ (vector of weights of all points in the rooted subtree of u, sorted by increasing –y coordinates), and let us construct a Cartesian tree $CT_y[u]$ for $W_y[u]$. The runtime is linear in $A_y[u]$.size, and so the total runtime is at most the time needed to construct the 2-range tree, that is O(n*log(n)).

<u>Answer to a query</u>. Let r be the root of T_x. Using binary search on $A_y[r]$, we can compute in O(log(n)) time indices i_{min},i_{max} such that all values with their –y coordinates between y_{min},y_{max} are exactly those between i_{min},i_{max}. Then, we apply the same approach as for exercise 7 (see also the discussion about range queries in class) in order to localize all values whose –x coordinate is between x_{min},x_{max}. The result is a set of O(log(n)) rooted subtrees, and of O(log(n)) additional values. We check directly amongst the O(log(n)) additional values which of them also have their –y coordinate between y_{min},y_{max} and thus from now on we only focus on the O(log(n)) rooted subtrees of T_x. Note that, as we traverse T_x starting from its root r, we can compute for each visited node u the indices $i_{min}[u]$, $i_{max}[u]$ such that all values in $A_y[u]$ with their –y coordinates between y_{min},y_{max} are exactly those between $i_{min}[u]$, $i_{max}[u]$. For that, it suffices to use the four pointers i_L,i_R,j_L,j_R stored for each element of $A_y[u]$. In particular, if u denotes the root of one of our O(log(n)) rooted subtrees, then we can use the Cartesian tree $CT_y[u]$ in order to determine in O(1) what the largest value stored in $W_y[u]$ between $i_{min}[u]$, $i_{max}[u]$ is.