Reminder to students for all exercise sheet: after preprocessing an n-node tree in O(n) time, we can compute the least common ancestor of any two nodes in O(1) time.

1) Show that after pre-processing an n-node tree in O(n) time, one can answer in O(1) time to the following type of queries q(e): ``does there exist a node in the tree whose value equals e?''

   We simply insert all values of the tree in a hash table. In particular, for this very type of queries it does not matter whether we store values in a vector or in a tree or etc...

2) Show, for any of the queries below, that we can answer them in O(1) time after an O(n)-time pre-processing:
   a. q(u,v,e): ``Is the edge e=xy present on the unique path between nodes u and v?''

      We pre-compute in O(n) time the levels (distances to the root) and the scheme for fast lca computation. Now, for answering a query q(u,v,xy), let us assume without loss of generality that x.level < y.level. Let w=lca(u,v), that we can compute in O(1) time. We answer YES if and only if w.level ≤ x.level and either lca(u,y) = y or lca(v,y) = y.

   b. q(u,v): ``What is the sum of all values stored on the path between u and v?''

      For each node x, we store the sum s(x) of all nodes on the path from x to the root. It can be done in O(n) time, i.e., during a BFS traversal. We also pre-compute the scheme for fast lca computation. Now, for answering a query q(u,v), let w = lca(u,v). We output s(u) + s(v) – 2*s(w) + w.value.

3) Show, for any of the queries below, that we can answer them in O(log(n)) time after an O(n)-time pre-processing:
   a. q(u,v): ``What is the maximum value stored on the path between u and v?''

      We pre-compute the levels of all nodes, and a heavy-path decomposition in O(n) time. For each heavy-path P, we also store all its values in a Cartesian tree CT[P] (considering P as a vector indexed from the node of least level to the node of deepest level). It can be done in O(|P|) time, and so in total O(n) time for all heavy-paths. Now, for answering a query q(u,v), let w = lca(u,v), that can be computed in O(log(n)) time from the HP decomposition. For each HP P that is fully contained on the path between u and w, resp. between v an w, we can output the maximum value (root of CT[P]) and keep the largest element we found. Then, we further need to consider the HPs P_u,P_v,P_w that contain u,v,w respectively and may not be fully contained on the uv-path. We use CT[P_u] in order to output the maximum value stored between u and either: w if it is also in P_u, or otherwise with the head h(P_u) of this HP. Similarly, we use CT[P_v] in order to output the maximum value stored between v and either: w if it is also in P_v, or otherwise with the head h(P_v) of this HP. Finally, let x,y be nodes on P_w such that x is on the uw-path, y is on the vw-path and x,y have maximum depth (note that one of x or y equals w). It is also possible to compute x,y in O(log(n)) time by using the compressed tree (in fact, this is implicitly done while we compute w). Without loss of generality x.level ≤ y.level. Then, we use CT[P_w] in order to output the maximum value between y and w.

b. q(u,v,r): ``is the value r stored in some node on the path between u and v?''

We pre-compute the levels of all nodes, and a heavy-path decomposition in O(n) time. For each heavy-path P, we also store all its values in a hash table H[P] (considering P as a vector indexed from the node of least level to the node of deepest level). We associate to each value e in the table H[P] the vector $Vec_P[e]$ of all nodes of which it is the value, ordered by increasing level. It can be done in O(|P|) time, and so in total O(n) time for all heavy-paths. Now, for answering a query q(u,v,r), let w = lca(u,v), that can be computed in O(log(n)) time from the HP decomposition. For each HP P that is fully contained on the path between u and w, resp. between v an w, we can check in O(1) time whether r is a a key of H[P]. Then, we further need to consider the HPs P_u,P_v,P_w that contain u,v,w respectively and may not be fully contained on the uv-path. Let x_u, x_v, x_w such that: x_u is the minimum-level node of P_u that lies on the uw-path (either w or the head of P_u), x_v is the minimum-level node of P_v that lies on the vw-path, and x_w is the maximum-level node of P_w that lies on either the uw-path or the vw-path. We use $H[P_u]$ (resp., $H[P_v]$ and $H[P_w]$) in order to determine whether value r is stored between u and x_u (resp., v and x_v, w and x_w).

4) Show that the offline variant of 3)b), where we are given in advance m queries to be answered, can be solved in O(n+m) time.

For each query $q(u_i,v_i,r_i)$, let $w_i = lca(u_i,v_i)$. All the values w_i can be computed in O(n+m) time by using a fast lca scheme. Now, we perform a DFS traversal. Each time we visit a node x for the first time (preorder), we ensure that all values stored on the path between x and the root are stored in a hash table, with each value being associated to its number of repetitions on that path. It can be done as follows:

- each time we visit a node y for the first time (preorder), let r = y.value. We check whether r is already a key in the table. If not, then we add r to the table with associated value 1. Otherwise, we increment the value already associated to r by 1.

- each time we visit a node y for the last time (preorder), let r = y.value. We decrement the associated value to r in the hash table by one. If this value falls to 0, then we drop the key r from the hash table.

For every node x, let H[x] denote the content of the hash table whenever we visit x. For answering to a query $q(u_i,v_i,r_i)$, it suffices to output and compare the values associated to r_i in $H[u_i]$, $H[w_i]$, $H[v_i]$ during the DFS traversal. Indeed, the answer to this query is yes if and only if w_i.value = r_i, or the value associated to r_i in either $H[u_i]$ or $H[v_i]$ is greater than the one in $H[w_i]$.

5) Show that we can reduce the problem of computing lca queries in a fixed tree to the range minimum query problem on some fixed vector (we have seen in class, with Cartesian trees, that the converse is also true!).

Solution 1: We do a DFS traversal. Every time we visit a node, we add a new entry in our vector (note in particular that a node is repeated as many times as it is visited during the DFS). To each element in the vector, we assign as its value the preorder of the corresponding node in the tree. Furthermore to each node u, let preorder(u) be the position in the vector that corresponds to the first visit of node u during the DFS. Now, in order to compute the lca of u and v, it suffices to compute the minimum value between preorder(u) and preorder(v).

Solution 2: Slight variation of the previous solution. We postorder all nodes, associating to each element in the vector the preorder of its corresponding node in the tree. Now, to compute the lca of u and v, as before we start computing the minimum element between u and v. Let x be the corresponding node. Unless u and v are on a common path to the root (that can be checked in O(1) by comparing their preorders and postorders), in general we cannot have x is the lca of u and v. Indeed, the lca of u and v must be postordered after u and v! However, x is a child of this lca that is either on the path to u or the path to v. So, we are done by outputting the father node of x.

6) A matching is a set of pairwise disjoint edges (i.e., no two edges can be incident to a same node). It is maximum if it has the largest possible cardinality.
   a. Propose an O(n) time algorithm for computing a maximum matching in a tree.

      Solution 1: Greedy. We first observe that the unique edge which is incident to a leaf must be always present in some maximum matching. Indeed, if v is a leaf, u is its father node, and there is a maximum matching which does not contain the edge uv, then we may simply replace any edge incident to u in the matching by uv. Therefore, we may compute a maximum matching greedily as follows: while the tree is not emptied, we select an arbitrary leaf v, we add its unique incident edge uv to the matching, then we remove u, v and all other leaves incident to u. Note that during the algorithm, the tree may become disconnected.

      Solution 2: Dynamic programming. We compute for each node x the following two values $m_1(x)$ = maximum matching in the subtree rooted at x, and $m_2(x)$ = maximum matching in the subtree rooted at x that does NOT contain any edge incident to x. Observe that $m_1(x) = m_2(x) = 0$ for a leaf. Otherwise, let $y_1, y_2, \ldots, y_d$ be the children of x. We have that $m_2(x)$ is the sum of all $m_1(y_i)$. Morever, $m_1(x)$ is the maximum value taken over $m_2(x)$ and all $1+m_2(x) - m_1(y_i) + m_2(y_i)$. We are done by outputting $m_1(\text{root})$.

   b. Show that after a pre-processing in O(n) time, we can answer in O(1) time to the following type of queries q(e): ``is the edge e=xy part of some maximum matching?''.

      We start applying the dynamic programming solution of 3.a.

      In doing so, we can detect all children y of the root r such that the edge yr is in a maximum matching. Indeed, this is the case if and only if $m_1(r) = 1 + m_2(r) - m_1(y) + m_2(y)$. Let us put label 1 to every such an edge, If furthermore $m_1(r) > m_2(r)$ (r must be incident to an edge in the matching), and y is the *unique* child of the root such that ry is in a maximum matching, then we label this edge by 2.

      Then, we continue traversing each node of the tree (modified DFS). Upon visiting a node x, let z be its parent.
      - If the edge xz has label 2, then we label 0 any incident edge xy. Indeed, the edge xz must be in any maximum matching of the tree, and so no other incident edge can also be in a maximum matching.

- Otherwise, any maximum matching in the subtree rooted at x can be completed into a maximum matching of the whole tree. In particular, an incident edge xy is in a maximum matching if and only if $m_1(x) = 1 + m_2(x) – m_1(y) + m_2(y)$. If there are at least two such edges, then we put label 1 to each of those.

- Finally, consider the special case when xz does not have label 2, and y denotes the unique child of x such that $m_1(x) = 1 + m_2(x) – m_1(y) + m_2(y)$. If xz has label 1, then there exists a maximum matching that does not contain xy, and so we put label 1 to xy. Otherwise, xz is not part of any maximum matching of the tree, that implies that the restriction of *any* maximum matching of the whole tree to the subtree rooted at x must also be a maximum matching of this subtree. In this situation, we put 1 to the edge xy if $m_1(x) = m_2(x)$, and 2 otherwise.

Finally, in order to answer a query q(e), we just need to verity whether the label of edge e is non-zero.

7) The distance between two nodes in a tree is the number of edges on the path in the tree between these two nodes. The eccentricity of a node is its maximum distance to any other node. Show that after a pre-processing in O(n), we can output the eccentricity of any node in O(1).

We pre-compute for each node x the height of its subtree (maximum distance between x and a leaf in the subtree rooted at x), denoted h(x). Note that in doing so, we computed the eccentricity of the root. Then, for each node x, we want to compute the maximum distance g(x) between x and a node which is not in its subtree. Indeed, the eccentricity of x is exactly max{g(x),h(x)}. Furthermore, if x is the root then g(x) = 0. Otherwise, let z be the father of x. We retain the two children a_z,b_z of node z with maximum height value. Then, $g(x) = 1 + \max\{g(z),1+h(a_z)\}$ if $x \neq a_z$, and $g(x) = 1 + \max\{g(z),1+h(b_z)\}$ otherwise.

8) A level-ancestor query q(v,i) asks, for a node v and some i what the ith closest ancestor to v is (i.e., if i = 1 then q(v,i) is the father of v, if i = 2 then q(v,i) is the father of the father of v, and so forth).

   a. Show that after a pre-processing in O(n*log(n)) time, we can answer to any level-ancestor query in O(log(n)) time.

   For each j < log(n) and every node x, we compute a pointer from x to its 2^j closest ancestor p(x,j). Let us call these new pointers ``shortcuts''. Since p(x,0) is the father of x and p(x,j+1) = p(p(x,j),j), then we can compute all the shortcuts in O(n*log(n)).

   Now, we can answer any query q(v,i) in O(log(n)) time by using these shortcuts, as follows. Let j = 1. While j < i we double the value of j. If j = i, then we are done. Otherwise, we are left solving the query q(p(x,j/2), i-j/2).

   b. Show that after a pre-processing in O(n) time, we can answer to any level-ancestor query in O(log(n)) time (for your information, it is actually possible to achieve O(n) pre-processing time and O(1) query time).

   We pre-compute all levels (distance to root) and a heavy-path decomposition. Now, to answer a query q(v,i) we just go up in the compressed tree. More precisely, let P be initialized to the HP containing u. While h(P).level – u.level < i we replace P by its father in the compressed tree. Finally, we output the node at distance u.level – i from the root in P.

c. Show that the offline version of the problem, where we are given in advance m level-ancestor queries to be answered, can be solved in O(n+m) time.

   We perform a DFS traversal of the tree. Each time we visit a node for the first time (preorder), we put it in a stack. Each time we visit a node for the last time (postorder), we remove it from the stack. Now, suppose that we have some query q(v,i) to be answered. At the time we visit v for the first time during the DFS, its level-i ancestor is exactly the ith node in the stack. If we use a static implementation of a stack (as an n-size vector), then we can output this level-i ancestor in O(1) time during the DFS.

9) In what follows, c(u,v) denotes the number of nodes closer to u than to v (i.e., if d(x,y) denotes the distance between two nodes, then c(u,v) equals the number of nodes x such that d(u,x) < d(v,x)).

a. Show that after a pre-processing in O(n) time, we can compute c(u,v) in O(1) for every *edge* uv of the tree.

   Without loss of generality, u is a child of v. Then, c(u,v) is equal to the size of the subtree rooted at u, while c(v,u) = n – c(u,v). The sizes of all rooted subtrees can be pre-computed in O(n) time.

b. Show that after a pre-processing in O(n) time, we can compute c(u,v) in O(log(n)) for every two nodes u and v that are related (i.e., one is an ancestor of the other).

   We precompute for each node x its level x.level and the size s(x) of its rooted subtree. Now, to compute c(u,v), let us assume without loss of generality that u is a descendant of v. Let d = u.level – v.level. If d = 2p is even, then let x and y be the level-(p-1) ancestor and level –p ancestors of u, respectively. Then, c(u,v) = s(x) and c(v,u) = n – s(y). Otherwise, d = 2p+1 is odd, and then let x be the level-p ancestor of u. Then, c(u,v) = s(x) and c(v,u) = n – s(x).

c. Show that after a pre-processing in O(n) time, we can compute c(u,v) in O(log(n)) for every two nodes u and v (possibly, unrelated).

   Let w = lca(u,v). Let also d = u.level + v.level – 2*w.level (distance between u and v). Without loss of generality, u.level – w.level ≥ v.level – w.level.

   - If d = 2p is even, then u.level – w.level ≥ p. Let x and y be respectively level-(p-1) ancestor and level –p ancestors of u. Observe that both x and y are on the uw-path. In particular, c(u,v) = s(x). If y ≠ w, then we also have c(v,u) = n – s(y). Else, let z be the level-(p-1) ancestor of v, we have c(v,u) = s(z).

   - If d = 2p+1 is odd, then u.level – w.level ≥ p+1. Let x be the level-p ancestor of u. Observe that x is on the uw-path and x ≠ w. Then c(u,v) = s(x) and c(v,u) = n – s(x).