CS 513
Solutions to #4

1. The *Longest Common Prefix* problem is defined as follows:

   Preprocess: $D = \{S_1, \ldots, S_n\}, S_i \in \Sigma^m$, that is D is a set of n strings, each of which is of length m.

   Queries: $LCP(i, j)$ returns k if $S_i[1 \ldots k] = S_j[1 \ldots k]$ but $S_i[k+1] \neq S_j[k+1]$.

   Give an algorithm for this problem.

   Answer: Let a *trie* of a set of strings be a rooted tree where the edges are labelled by the symbols of the alphabet, where no two edges coming from a node have the same label, and where, for each string S_i, there is a node v_i in the trie, such that the string derived from concatenating the edge labels from the root to v_i is S_i. Furthermore, the trie is the minimal such tree (that is, don't add any bogus extra nodes that don't lead to string nodes). In other words, the trie is the tree we would get by tracing down from the root with each string, following edges whose labels match the string characters, until we reach a node where none of the outgoing edge labels correspond to the next character of the string. See Figure 1 for an example. From this contructive description,

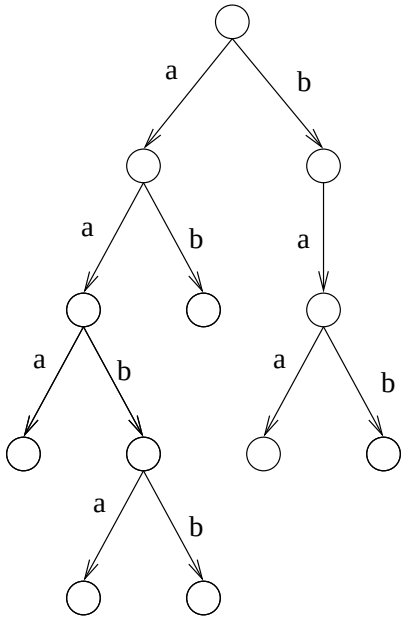

Figure 1: Trie of string set $\{aaa, aaba, aabb, aab, ab, baa, bab\}$

we get an algorithm for building a trie in linear time (as long as the alphabet size

is constant– what happens when the alphabet size isn't constant?). Notice that the longest common prefix of two strings is given by their least common ancestor in the trie. So we can answer lcp queries in constant time after linear preprocessing, both for building the trie and for doing lca preprocessing.

2. Given a rooted tree T, define $T(v)$ to be the set of leaves below node v in T. For a set of nodes V', define $LCA_T(V')$ to be the shallowest node u in T such that $u = LCA(w, x)$, for $w, x \in V'$.

   Given two rooted trees S and T with the same leaf labels, define, for each node v in S, $M(v) = LCA_T(S(v))$, so it's the least common ancestor in T of all leaves below v in S.

   Give an algorithm to compute M.

   Answer: We'll solve the problem for binary S. I'll leave the generalization to higher degree to you. First we preprocess T for lca queries in $O(n)$ time. Now suppose we want to compute $M(v)$ and we've computed $M(l)$ and $M(r)$, where l and r are the children of v. I claim that $M(v) = LCA_T(M(r), M(l))$. Before proving this claim, notice that if we just compute M bottom up from the leaves, we get a linear time algorithm, since it takes constant time to compute M at any node. Now for the proof of the claim.

   Proof of the claim: Notice that all leaves in $S(l)$ are below $M(l)$ and all leaves in $S(r)$ are below $M(r)$. So all leaves in $M(v)$ are below $LCA_T(M(r), M(l))$. Suppose that there is some node u in T such that all node in $M(v)$ are below u and u is below $LCA_T(M(r), M(l))$. Then, since for some pair x, y of leaves in $S(l)$, $M(l)$ is their least common ancestor, $M(l)$ must be below u, and by symmetry, $M(r)$ must be below u. But then u is a common ancestor of $M(l)$ and $M(r)$ below their lca, giving a contradictino. Thus no such u exists, and $LCA_T(M(r), M(l))$ is the lca of $M(v)$, as claimed.

3. The Set Cover (SC) problem is defined as follows:

   Input: A collection C of subsets of a finite set S, and a postive integer $K \leq |C|$.

   Output: Yes, if there is a subsets C' of C such that $|C'| \leq K$ and $\cup_{c \in C'} c = S$, No, otherwise.

   The Vertex Cover (VC) problem is defined as follows:

   Input: A graph $G = (V, E)$ and a postive integer $K \leq |V|$.

   Output: Yes, if there is a subsets V' of V such that $|V'| \leq K$ and for every edge $\{a, b\} \in E$, $\{a, b\} \cap V' \neq 0$ No, otherwise.

   Show that if there is a polynomial time algorithm for SC, there is a polynomial time algorithm for VC.

Answer: Suppose you are given a graph $G = (V, E)$ and you want to know if it has a vertex cover of size at most K. Let's create an instance of SC. Let $S = E$, and for each $v \in V$, let c_v be the set of all edges adjacent to v in G. Let C be the set of all such c_v. By definition, G has a K VC iff S, C has a K SC. Creating S and C takes polynomial time, so we are done.

4. Let $T = (V, E)$ be an edge weighted tree such that $e \in E$ has minimal weight. Let T_1 and T_2 be the trees derived from T by removing e. Then we define a *cartesian tree* of T to be a binary tree such that e is the root, and the left and right children of e are the cartesian trees of T_1 and T_2, respectively. If either T_1 or T_2 are singleton nodes, then their cartesian trees are empty.

   Give an algorithm for finding a cartesian tree of a tree. The faster the algorithm, the better your grade.

   Answer: A simple application of Union-Find [Tarjan 1975] gives a $O(n \log n)$ algorithm which is optimal as shown in the next problem. Given an edge-weighted tree $T = (V, E)$ find its cartesian tree as follows.

   - Sort the edges in E by decreasing weight.
   - For each node $v \in V$
     - $make - set(v)$.
   - For each edge (u, v) in decreasing order of weight
     - $union(find(u), find(v))$ (label the root as (u, v)).

   Correctness: We prove the algorithm is correct by induction on the number of edges $|E|$ of T. If $|E| = 1$ the cartesian tree obtained is correct. Assume now that the algorithm is correct for $|E| < m$ and consider the case $|E| = m$. The last node added by the algorithm to the cartesian tree corresponds to the lightest edge in T, call it e, and it is placed as the root of the cartesian tree by the union operation. The left and right subtrees of the cartesian tree correspond to the subtrees of T rooted at the end points of e. This is correct according with the definition of a cartesian tree. On the other hand, the left and right subtrees are cartesian trees of subtrees with less than m edges and therefore they are correct by the inductive hypothesis.

   Running time: The $\Theta(n)$ union or find operations take overall $\Theta(n\alpha(n))$ steps, where $\alpha(n)$ is related to a functional inverse of Ackermann's function and is very slow-growing. Then, the running time is dominated by sorting the edges, i.e. $O(n \log n)$.

5. Give a lower bound of $\Omega(n \log n)$ for constructing the cartesian tree of a tree.

   Answer: Assume for the sake of contradiction that there exists a $o(n \log n)$ algorithm that builds a cartesian tree of a tree. Then, we can sort n numbers in $o(n \log n)$ steps as follows. Consider a *star* of degree n, call it S. Assign each number in the set of numbers to be sorted as the weight of one edge and build the cartesian tree of S. The cartesian tree of a star is a chain of nodes sorted by their labels, where the labels

represent the weights of the edges of the star. Therefore, we have an algorithm that sorts n numbers in $o(n \log n)$. But this violates the comparison-based sorting lower bound, hence, the $\Omega(n \log n)$ lower bound for constructing the cartesian tree of a tree holds.