Written Qualifying Exam
Theory of Computation

Fall, 1997

Friday, September 26, 1997

This is nominally a *three hour* examination, however you will be allowed up to four hours. There are six questions; answer all six questions. All questions carry the same weight.

- Please print your name on the back of your exam envelope next to your exam identification number. Do NOT write your name on the exam booklets.
- Use a separate booklet for each question. The exam booklets have been labeled by question number; please be sure to use the correct booklet for the question you are answering.

Read the questions carefully. Keep your answers brief. Assume standard results, except where asked to prove them.

Problem 1 [10 points]
Consider the following farthest vertex problem. The input is a rooted tree where each edge is undirected and has an integer length, possibly negative.

a. Give a linear time algorithm to determine for each vertex v the most distant vertex in its subtree; formally, this means find a vertex w in v's subtree such that the distance from v to w is at least as large as the distance from v to x for any other vertex x in v's subtree. Record the vertex w and the distance to w at vertex v.

b. Using the results of the algorithm of part (a), if helpful, give a linear time algorithm to find for each vertex v the most distant vertex *outside* its subtree. More formally, find a vertex y outside v's subtree, such that the distance from v to y is at least as large as the distance from v to any other vertex z outside v's subtree.

Problem 2 [10 points]
Consider the following algorithm A for finding a *best root* in a tree. Each edge has a positive integer length. The best root is a vertex r, such that the distance to the vertex v farthest from r is minimized. Formally, choose r so that $\max_v\{\text{distance}(r, v)\}$ is minimized. Prove the algorithm is correct.

A:
Step 1. Choose an arbitrary start vertex s. By means of a depth first search starting at s, find a vertex u most distant from s.
Step 2. By means of a second depth first search starting at u, find a vertex v most distant from u.
Step 3. Find r, a vertex on the path from u to v which is nearest the midpoint of this path.

Thus you need to prove r is a solution to the problem.

Go to the next page

Problem 3 [10 points]
Consider the following *nearby string* problem. The input consists of two strings u and v over a finite alphabet Σ and a parameter k, an integer. The problem is to determine whether there is a subsequence w of characters present in both u and v where w is obtained by deleting a total of at most k characters from u and v. Let $n = |u| + |v|$.
e.g. $u = bbcabe$, $v = bbabf$, $k = 3$; then $w = bbab$ is a solution; but if $k = 2$ there is no solution.

Suppose you are given a function $Match(u, v, r, s)$ which reports the length of the longest common substring of u and v ending at the rth character in u and the sth character in v.
e.g. $Match(u, v, 5, 4) = 2$ (the substring in question is ab).
Further suppose $Match$ runs in $O(1)$ time. (In fact, such a function exists, given $O(|u| + |v|)$ preprocessing, but this is outside the scope of this question.)

a. Give a recursive function $Nearby\_Match(u, v, i, j, k)$ which determines if there is a nearby match of the strings comprising the first i characters of u and the first j characters of v. Remember k is the total number of deletions allowed. Your function should have a depth of recursion of no more than k.

b. Suppose dynamic programming is used with the function in part (a). Show that this function then runs in $O(nk^2)$ time (this may require modification of your answer for part (a)).

GO TO THE NEXT PAGE

Problem 4 [10 points]
Classify the following languages as regular, context free but not regular, context sensitive but not context free, or none of these. Justify your answers.

a. $L_1 = \{v \mid v \in \{a,b\}^*$ and the number of a's and b's in v are equal$\}$.

b. $L_2 = \{v \mid v \in \{a,b\}^*$ and for each prefix w of v, the number of a's and b's in w differ by at most $5\}$.

Problem 5 [10 points]
The problem of factoring a number m is to report the primes whose product equals m. e.g. $factor(24) = 2, 2, 2, 3$. All numbers here are assumed to be in the standard binary notation. Currently, we do not know whether we can factor an integer in polynomial time.

Suppose $P = NP$. Give a polynomial time algorithm for factoring.
Note that NP is a class of *languages* but the integer factoring problem amounts to the computation of a *function*.

Problem 6 [10 points]
Let M be a deterministic finite state machine (fsm) with two read-only tapes, called the *input tape* and *advice tape*. The tape heads on these two tapes can only go from left to right, and their respective tape alphabets are Σ_1 and Σ_2. Suppose the strings $x \in \Sigma_1^*$ and $\alpha \in \Sigma_2^*$ are placed on these tapes, and M begins computing with the tape heads initially scanning the leftmost symbols of x and α. If M eventually halts and enters an "accept" state, we say *M accepts x with advice α*. Now let

$$A = a_1 a_2 a_3 \cdots$$

be an infinite string in Σ_2. For any $x \in \Sigma_1^*$ of length n, we say $M^{(A)}$ *accepts* x if M accepts x with advice $\alpha = a_1 a_2 \cdots a_n$. So α is just the prefix of A of length $n = |x|$. Let $L(M, A) \subseteq \Sigma_1^*$ be the language accepted by $M^{(A)}$. Show that there is an M and A such that $L(M, A)$ is non-r.e.

Note that you need to choose both A and M.

Solutions

Solution to Problem 1

a. The answer is readily computed by a postorder DFS. Each child of a vertex v should have the distance/name of the its most distant descendant computed, and then the information can be used to update v.

procedure $Desc(T)$;
begin
  $T.dist \leftarrow 0$;
  $T.vname \leftarrow T$;
  foreach child w of T **do**
    $Desc(w)$;
    if $T.dist < w.dist + length(T, w)$ **then**
      $T.dist \leftarrow w.dist + length(T, w)$;
      $T.vname \leftarrow w.vname$
    endif
  endfor
end$Desc$

This clearly runs in linear time.

b. Here each vertex should get the necessary information from its parent. Let $w.xname$, $w.xdist$ be the solution pair for vertex w. The parent of w needs two kinds of information. First, it should have the distance/name to the most distant vertex outside of its subtree. Second, it should know, for each child w, the distance/name for the farthest descendant that is not in w's subtree, since the descendant for v may be the farthest descendant for w. It follows that each vertex needs the solution to part (a) for both the "first-place" subtree, and the "second-place subtree."

So the solution is: Modify a with additional initialization
  $T.dist2 \leftarrow 0$;
  $T.vname2 \leftarrow T$;
and additional conditionals to update $T.vname2$ and $T.dist2$, which are the name and distance for a vertex that is as far as possible from T, a descendant of T, and is in a subtree in T that is not in the same subtree as $T.vname$. (There are trivial modifications to this spec that permit either or both distant vertices to be T itself.)

The code is then a preorder computation that assigns the appropriate (i.e. the largest) distance/vertex pair to ($w.xdist$ and $w.xname$). Foreach child w of v, $(w.xdist, w.xname)$ is assigned

$$\max \begin{cases} (0, w) & \\ (length(v,w) + v.xdist, v.xname) & \\ (length(v,w) + v.dist, v.vname) & \text{if } v.vname \neq w.vname, \\ (length(v,w) + v.dist2, v.vname2) & \text{if } v.vname = w.vname. \end{cases}$$

This clearly runs in linear time.

Solution to Problem 2 Let s, u, v and r be as defined in the problem. Let $l = \max\{distance(r,v),\ distance(r,u)\}$. Correctness follows from two facts.

First, there is no vertex z with $\max\{distance(z,u), distance(z,v)\} < l$, since r is by definition as close to the midpoint of the path from u to v as possible, all edge lengths are positive, and the subgraph connecting any such z to both u and v must include the path from u to v since T is a tree.

Second, it must be shown that $distance(r,w) \leq l$ for all $w \in T$. But suppose that there were a w such that $distance(r,w) > l$. Consider the path p_w from r to w, and the path p_u from r to u. If these paths have no edges in common, then the path from u to w (which would have length $distance(w,r) + distance(r,u)$) would be longer than the path from u to v, which contradicts the definition of v. So they must have edges in common. Now consider the path from s to u. Let it join the path from v to u at x. We claim that x is between v and r (inclusive), as otherwise the edges from s to x to r to v is a path with a length that would be larger than the path from s to x to u (since $distance(x,u)$ would not be any larger than $distance(r,v)$, if x were between r and u.) But if x is as claimed, the path from s to r to w would then be larger than the path from s to u, which contradicts the definition of u. Thus there is no such w. This proof can be explained more easily with pictures to illustrate each (nonexistent) case.

Solution to Problem 3

a. The following two claims justify the recursive solution given below.

Claim 1: If w is a longest common subsequence for strings x and y, then wa is a longest common subsequence for xa and ya, where a is a single character.

Claim 2: If w is a longest common subsequence for strings x and y, then wu is a longest common subsequence for xu and yu, for any string u.

The proofs are immediate.

Boolean function $Nearby\_Match(u,v,i,j,k)$;
begin
  $l = Match(u,v,i,j)$;
  if $(k = 0$ **or** $l = i$ **or** $l = j)$ **then return** $(k \geq \max\{i-l, j-l\})$
  else return $(Nearby\_Match(u,v,i-l-1,j-l,k-1)$ **or**
      $Nearby\_Match(u,v,i-l,j-l-1,k-1))$
end

Since every recursive call reduces the parameter k to $k-1$, and there is a base case at $k = 0$, the depth of recursion is at most k. The algorithm simply removes longest matching substrings from the right ends of u and v (as per Claim 2), and when the right ends do not match, tries the remaining two options: deleting the rightmost characters from either u or v. This parallels the standard algorithm for finding the string edit distance between u and v.

b. We note that if $|i-j| > k$ then deleting k characters from the longer string, v say, leaves a string longer than u. Thus, in this case, there is no matching substring. We add this as an initial test to the above function.

The purpose of dynamic programming is to ensure that each recursive call with a given set of parameter values occurs at most once. There are $|u| \leq n$ choices for i; for each choice of i, there are at most $2k+1$ choices for j, and there are k choices for the final parameter. This yields a bound of $O(nk^2)$ on the number of recursive calls. Each recursive call takes $O(1)$ time, giving an overall running time of $O(nk^2)$.

Solution to Problem 4

a. L_1 is context free but not regular. L_1 is accepted by the following pda. On its stack it keeps track of the difference in the number of b's and a's scanned so far (this is simply a unary counter). The pda accepts only if the counter is at zero when the input is fully scanned.

To see L_1 is not regular, we note that the intersection of two regular languages is regular. Thus if L_1 were regular, then $L_1 \cap \{a^*b^*\} = \{a^ib^i \mid i \geq 0\}$ would be regular also. But it is a standard application of the pumping lemma for regular languages to show $\{a^ib^i \mid i \geq 0\}$ is not regular.

b. L_2 is regular. It is accepted by the following 12 state dfa. 11 states are used to keep track of the difference in the number of a's and b's scanned so far, for differences ≤ 5; the 12th state is entered if the difference exceeds 5 and once entered this state is not left. The first 11 states are the accept states.

Solution to Problem 5

Consider a simpler factoring problem: given a number n, we output 1 if n is prime, and otherwise, we output some number m such that $1 < m < n$ and $m|n$ (i.e., m divides n). Call this problem "SFAC" (Simple Factoring). It is enough to solve SFAC in polynomial time because integer factoring can be reduced to calling SFAC at most a linear number of times on numbers that are at most the original size. [Why?]

To solve SFAC, consider the following language:

$$L_0 = \{(n,p,q) : (\exists m) p \leq m \leq q \text{ and } m|n\}.$$

This language is clearly in NP: we guess a number m between p and q and verify that m divides n. Since $P = NP$, there is a polynomial time algorithm to recognize L_0. To solve SFAC, on input n, we perform a binary search for a factor of n: initially set $p = 2, q = n-1$ and check if $(n,p,q) \in L_0$. If not, we can output 1 since n is prime. Otherwise, we split $[p,q]$ into roughly two equal halves $[p,r]$ and $[r+1,q]$ and proceed to find one half that contains a factor. We can continue on any half that contains a factor. After about $2\log n$ queries to L_0, we will narrow the interval $[p,q]$ to size 1, and therefore find a factor.

Solution to Problem 6 Let $K \subseteq \mathbf{N}$ be any non-r.e. set of numbers. Let $A = a_1a_2a_3\cdots$ be a 0/1 string such that $a_i = 1$ iff $i \in K$. Let M on input x and advice $a_1a_2\cdots a_n$ (where $n = |x|$) accept iff $a_n = 1$. Clearly such an M can be constructed. We claim that $L(M,A)$ is not r.e. Note that $L(M,A) = \{x : |x| \in K\}$. If $L(M,A)$ were r.e., then we claim that K would be r.e. For any binary string $b \in \{0,1\}^*$, let $\#(b) \in \mathbf{N}$ denote the number represented by b in binary notation. To see this, suppose there is a partial computable procedure P that on input x, halts iff $x \in L(M,A)$.

It suffices to construct a computable procedure Q that on input $b \in \{0,1\}^*$, would halt iff $\#(b)$ is in K. This would prove that K is r.e., by the standard equivalence between computability on natural numbers and computability on their binary representations. The procedure on input b will generate a string w of length $\#(b)$ and invoke the procedure P on w. Then Q halts iff P halts on w. Clearly, Q halts iff $\#(b) \in K$, as claimed.