Harvard University
Computer Science 121

Problem Set 8

Due Tuesday, November 23, 2010 at 1:20 PM.
Submit a single PDF (lastname+ps8.pdf) of your solutions to cs121+ps8@seas.harvard.edu
LATE PROBLEM SETS WILL NOT BE ACCEPTED.
See syllabus for collaboration policy.

PROBLEM 1 (3+3+3+3 points)

For each of the following statements, state whether it is true or false, and prove your assertion. For parts A-C, if the statement is true, you should state what the actual parameters n_0 and c are, and why.

(A) $4n^3 + 2n^2 = O(n^3 + n)$.
(B) $\log_2 n = \Theta(\log_{10} n^2)$.
(C) $2^{n/2} = \Omega(2^n)$.
(D) $n^{100001} = o(1.00001^n)$. (*Hint:* you may want to recall l'Hôpital's rule.)

(A) **True.** Both are polynomials of degree 3. For example, letting $c = 6$, we get $f(n) \leq g(n)$ for $n > n_0 \equiv 0$ (where f and g are the left and right hand sides). We could also use any $c > 4$ with larger n_0.
(B) **True.** Note that $\log_{10} n^2 = \frac{2}{\log_2 10} \log_2 n$. So letting $c = \frac{\log_2 10}{2}$, it's true for $n > 0$.
(C) **False.** Assume true; then there exist c, n_0 such that $2^n \leq c2^{n/2}$ for all $n > n_0$. But this is true iff $2^{n/2} \leq c$ for all sufficiently large n, which does not hold, leading to a contradiction. For example, taking logs of both sides, we see that it is false for any $n > 2 \log_2 c$.
(D) **True.** Intuition is that the left side is polynomial and the right is exponential. A more formal proof is that the limit of $n^{100001}/1.00001^n$ as $n \to \infty$ is 0 (by applying L'Hopital's rule lots of times).

PROBLEM 2 (10 points)

Throughout the class we have focused on computability properties of *languages*. But often we want to understand computational problems that are not YES/NO problems, for example computing a function. In this problem we will see how such problems can be translated into "equivalent" languages (so focusing on languages is not a big restriction).

Given any function $f : \Sigma^* \to \Delta^*$, show how to define a language L_f such that (a) any algorithm to compute f can be computably transformed into an algorithm that decides L_f, and (b) conversely, any algorithm that decides L_f can be transformed into an algorithm that computes f.

(A) Given a function $f : \Sigma^* \to \Delta^*$, we define the language

$$L_f = \{x\$f(x) : x \in \Sigma^*\}$$

over the alphabet $\Sigma \cup \Delta \cup \{\$\}$, where \$ is some special symbol not contained in either of the alphabets Σ and Δ.

- Given an algorithm A which computes f, we can transform A into a new algorithm D that decides L_f. On input w, D first checks if $w = x\$y$ for some $x \in \Sigma^*$ and $y \in \Delta^*$. If w does not take this form, D rejects. Otherwise, if $w = x\$y$, D simulates A on input x and accepts if and only if the result is y.
- Conversely, given an algorithm D which decides L_f, we can transform D into an algorithm that computes f as follows. On input x, A runs over all possible strings $y \in \Delta^*$. For each y, simulate D on input $x\$y$: if D rejects, move on to the next value of y, and if D accepts, then return y. This process must halt since at some point there will be a value of y for which $f(x) = y$.

PROBLEM 3 (10+15 points)

Prove that the class P is closed under
(A) Concatenation.
(B) Kleene star. (*Hint:* Use dynamic programming. Look at the algorithm we gave in class for recognizing context-free languages via Chomsky Normal Form.)

(A) Let L_1, L_2 be an arbitrary languages in $\mathcal{P}$. Given a string $w \in \Sigma^*$ such that $w = \sigma_1\sigma_2\cdots\sigma_{|w|}$ where $\sigma_i \in \Sigma$ for all $1 \leq i \leq |w|$, we can use the following algorithm to determine whether $w \in L_1 \circ L_2$:

```
for i := 0 to |w|
        if σ_1 ··· σ_i ∈ L_1 and σ_{i+1} ··· σ_{|w|} ∈ L_2, then
                halt and answer YES
        end if
end for
halt and answer NO
```

On input w, the algorithm simply checks all (possibly empty) strings x, y such that $xy = w$; if $x \in L_1$ and $y \in L_2$, then the algorithm halts YES. Note that we can decide membership in L_1 and L_2 in polynomial time, say $p_1(n)$ and $p_2(n)$ respectively. Since we make $O(|w|)$ such checks, the running time of this algorithm is $O(|w|(p_1(|w|) + p_2(|w|)))$, which is again a polynomial.

(B) Let L be an arbitrary language in $\mathcal{P}$; then there exists a Turing machine M and a strictly increasing polynomial p such that M decides L and M always halts on an input w in no more than $p(|w|)$ steps. Given a string $w \in \Sigma^*$ such that $w = \sigma_1\sigma_2\cdots\sigma_{|w|}$ where $\sigma_i \in \Sigma$ for all $1 \leq i \leq |w|$, we can use the following algorithm to determine whether $w \in L^*$. If $w = \epsilon$ we accept (because $\epsilon \in L*$ for any language L. For any other string, we note that a string u is in L^* if and only if either $u \in L$ or $u = u_1u_2$ such that $u_1, u_2 \in L^*$. We can therefore use dynamic programming (a method that consists of keeping track of solutions to subsets of the problem - in this case whether substrings are in L^*) to determine whether a string is in L^*. First we create a set R to store all of the strings that we know to be in L^* (this is initialized as the empty set). We then run through each of the substrings of w and check whether it is in L (which would imply that it is in L^*); if so, we add this string to R. We then run through the substrings (in order by length) and check whether they are the concatenation of two substrings known to be in L^* (ie in R); if so, we add this new substring to R. The pseudocode for this algorithm can be written as follows:

```
if w = ε, halt and answer YES
R := ∅
for i := 1 to |w|
        for j := i to |w|
                if σ_i σ_{i+1} ··· σ_j ∈ L then
                        R := R ∪ {(i, j)}
                end if
        end for
end for
for i := 2 to |w|
        for each (i, j) ∈ R
                if (1, i − 1) ∈ R then
                        R := R ∪ {(1, j)}
                end if
        end for
end for
if (1, |w|) ∈ R then
        halt and answer YES
else
        halt and answer NO
end if
```

Proof of Correctness. Suppose $w \in L^*$. Then clearly there exists a (possibly empty) series of non-empty strings $w_1, w_2, \ldots, w_n \in L$ such that $w = w_1 w_2 \cdots w_n$. After the first loop completes, we know that $(1, |w_1|), (|w_1| + 1, |w_1| + |w_2|), (|w_1| + |w_2| + 1, |w_1| + |w_2| + |w_3|), \ldots$ will be in R. As the second set of loops is executing i will eventually take on the value $|w_1| + 1$, causing $(1, |w_2|)$ to be added to R; later, it will take on the value $|w_1| + |w_2| + 1$, causing $(1, |w_3|)$ to be added to R, and so on. Eventually $(1, |w|)$ will be added, causing the algorithm to answer YES.

Conversely, suppose that the algorithm answers YES. Then it must be the case that $(1, |w|) \in R$, which means that there was some sequence of ordered pairs $(i_1, i_2), (i_2, i_3), \ldots, (i_{n-1}, i_n)$ all of which were in R at the completion of the first loop such that $i_1 = 1$ and $i_n = |w|$; but this implies the existence of a way to break up w into substrings $w_1, w_2, \ldots, w_n \in L$, which implies that w is in fact in L^*.

It remains only to be shown that this algorithm operates in polynomial time. The first algorithm iterates over all $O(|w|^2)$ substrings of w. Each of those strings is tested for membership in L; since all each is no longer than $|w|$ characters, each can be tested in at most $p(|w|)$ steps; thus the first stage of the algorithm takes at most $O(|w|^2 p(|w|))$ operations, which is polynomial because polynomials are closed under multiplication. The second algorithm iterates over $O(|w|)$ values in the outer loop and $O(|w|)$ values in the inner loop, and so takes time $O(|w|^2)$ in all.

PROBLEM 4 (10 points)

Show that there is a polynomial time algorithm which, given an NFA N and string w, determines whether N accepts w. Assume a multitape TM model of computation and analyze the degree of the polynomial as a function of both $|\langle N \rangle|$ and $|w|$.

Note that converting N to a DFA won't do the trick, because that step alone would be exponential in $|\langle N \rangle|$.

(A) $|N|$ is the number of states in the given NFA, and $|w|$ is the length of the input string w.

If there are epsilon transitions in the transition function, we can eliminate epsilon transitions by updating the transition functions, this takes $|N^2|$ time. In one of the tapes of the multi tape TM, keep an array of states and a boolean spot for each state. If a state is an element of current states in the computation, the boolean spot is turned to 1, otherwise it is kept 0.

Set q_0 to 1
For each symbol φ in the string w
    For each state q that is labeled 1
        For each $q_i \in \delta(q, \varphi)$
            Set q_i to 1
    Set q to 0 if it is not labeled in this pass
Accept if there exists a q_i labeled 1 that is in final set of states

The time complexity is bounded by $O(|w||N|^2)$ which is polynomial in $|w|$ and $|N|$.

If we include the representation factor, each state is represented by $log(|N|)$ bits. The final complexity becomes $O(|w||N|^2 log(|N|))$

PROBLEM 5 (10 points)

Prove that $\mathrm{ALL}_{\mathrm{DFA}} = \{\langle D \rangle : D \text{ is a DFA and } L(D) = \Sigma^*\}$ is in P.

(A) Construct a decider T for $\mathrm{ALL}_{\mathrm{DFA}}$:

T = On input $\langle D \rangle$, where D is a DFA,
1. Initially unmark all of D's states.
2. Mark the start state of D RED.
3. Scan for a RED state q. If none exists, then go to Step 6.
4. For all $q' \in \{\delta(q, \sigma) : \sigma \in \Sigma\}$, mark q' RED.
5. Mark q BLACK and go to Step 3.
6. If at least one non-accept state is marked BLACK, *reject*; otherwise *accept*.

T checks if there is a reachable non-accept state. If there is one, then the path from the start state to this particular state forms a string that the DFA will reject. Otherwise, any string will drive the DFA to an accept state.

Each step of T can be executed in polynomial time. The loop from Step 3 to 5 will execute at most N times, where N is the number of states in the DFA, and the representation of any DFA has length at least linear in the number of states. Hence T runs in polynomial time, and $\mathrm{ALL}_{\mathrm{DFA}} \in \mathrm{P}$.

PROBLEM 6 (Challenge!! 3 points)

It is known (though not trivial) that testing whether a binary number represents a prime number is in P. However, it is currently unknown whether or not a number (given in binary) can be *factored* in polynomial time. Explicitly construct a TM M that factors numbers such that M runs in polynomial time iff there exists a TM that factors numbers in polynomial time. We want you

to give us a correspondence between existence and construction. M should factor a number no matter what, but do it in polynomial time if in fact there exists *some* TM that factors numbers in polynomial time. M should take a binary number n as input and then halt with the (unique) prime factorization of n written on its tape.

(A) First note that we can associate each natural number with a TM by just taking the binary representation of a number and considering it as a string. We then create some encoding of TMs over the alphabet $\{0, 1\}$. Then each number is associated with a unique string that may or may not encode an actual TM.

Given a number x, run the TM encoded by 0 for one step. (If 0 is not a valid TM, just skip it). Then run the TM encoded by 1 for 1 step and the one encoded by 0 for 2 steps. Then run the TM encoded by 2 for a step, and the TM's encoded by 0 for 2 steps and for 3 steps run TM 1. Proceed to "dovetail" the TMs until you start to run TM $|x|^{|x|}$. Then stop doing this. If on the way any TM halts, check if its answer is correct. We can check if the answer is correct by simply multiplying the numbers together and making sure that that equal x, and then verifying that they are all prime. If so halt with that answer, if not disregard it. If at the end of all this dove-tailing if no TM has out put the correct answer, just factor the number by brute force.

This always outputs a correct answer because it checks any answers that one of the other TMs outputs for correctness before outputting it, and if none of these are correct it just uses brute force to correctly factor the number (which can clearly be done by exhaustive search).

If there is a polynomial time algorithm for factoring, then there exists a y such that TM y factors in time $|x|^n$ for some n when $|x| > C$ for some C. Then $y + |x|^n < |x|^{|x|}$ for large enough input. And if TM y is allowed to run for $|x|^n$ steps and if $|x| > C$ it will output the correct answer. We can always find an x large enough for both of these equations to hold. Then when $|x|$ is larger than this we will only run enough computations to find this answer. Meaning we will halt after running at most $|x|^n$ computations on TM y. So we will run at most $|x|^n + y$ computations on any TM. And because we do these one at a time (first we run it with 1 step, then for 2 steps, then for 3 steps then for $y + |x|^n$ steps) we will may actually run $(|x|^n + y)^2$ computations on any given TM. We only run any computations on $|x|^n + y$ TMs. So we run at most $(|x|^n + y)^3$ computations. Since y is a constant, this is bounded by x^{3n} simulated computations. To simulate we need quadratic time, so we actually may use x^{6n} computations plus those for checking incorrect answers. This can all be done in polynomial time.

If no algorithm exists, then this TM cannot run in polynomial time. For assume that this algorithm runs in polynomial time. We know that it factors numbers, so then a TM that factors numbers (namely this one) would exist, contradiction.