Special Topics in Complexity Theory, Spring 2006, Yap
Homework 3 (With Solutions)
Due: Wed Mar 8.

INSTRUCTIONS

- REMEMBER OUR HOMEWORK RULES: you prepare a solution in writing, and then we will have appointments in which you need to give an oral defence of your solution. I will grade BOTH the oral defence AND your written solution.

---

1. (10 Points)
Note that the original question has been modified/refined.

Consider the following three definitions of the square root function:

   - $sqrt_1 \ :\subseteq \ \mathbb{R} \to \mathbb{R}$, with $sqrt(x) = \uparrow$ if $x < 0$, and $sqrt(x) = \sqrt{x}$ for $x \geq 0$.
   - $sqrt_0 : (0, \infty) \to \mathbb{R}$. This is the restriction of $sqrt(x)$ to $(0, \infty)$.
   - $sqrt : [0, \infty) \to \mathbb{R}$. This is the restriction of $sqrt(x)$ to $[0, \infty)$.

Show the computability or uncomputability of these functions.

SOLUTION We will show:
(i) The function $sqrt_1 \ :\subseteq \ \mathbb{R} \to \mathbb{R}$ is uncomputable.
(ii) The function $sqrt : [0, \infty) \to \mathbb{R}$ is computable.
(iii) The function $sqrt_0$ is computable.

Proof:
(i) Suppose M is an OTM to compute $sqrt_1$. To obtain a contradiction, consider the oracle function $\overline{0^-}$ for zero where $\overline{0^-}[p] = -2^{-p}$ for all p. Consider the output of M on input $\overline{0^-}$ and $p = 1$. Clearly, M must halt after some t steps. Now, consider another oracle function $\overline{x}$ where $\overline{x}[p] = \overline{0^-}[p]$ if $p \leq t$, and $\overline{x}[p] = -2^{-t}$ for all $p > t$. Clearly, $\overline{x}$ represents the negative number 2^{-t}, but M on input $\overline{x}$ and $p = 1$ will halt with an output as before, contradiction.
(ii) Of course, (ii) implies (iii). However, for our proof, we shall show (iii) independently. We shall now derive (ii) from (iii), i.e., by assuming that $sqrt_0$ is computable. To show the computability of $sqrt$, we must ensure that, in case the input represents 0, then we output a representation of 0. We do this as follows. On input $\overline{x}$ and p where $x \geq 0$, we first compute $\overline{x}[2p+2]$ and compare this to 2^{-2p-1}. If $\overline{x}[2p+2] \geq 2^{-2p-1}$ then $x \geq 2^{-2p-2}$, and we can compute a p-bit approximation of $\sqrt{x}$ using an algorithm for $sqrt_0$ (from part(i)). Otherwise, we simply output 0. We show that this output of 0 is correct: since $\overline{x}[2p+2] < 2^{-2p-1}$, we have $x < 2^{-2p}$, or $\sqrt{x} < 2^{-p}$. Hence 0 is a p-bit approximation to $\sqrt{x}$.
(iii) We now show the computability of $sqrt_0$.

(a) *The function $sqrt_0$ is approximable.* This means that for each $x \in \mathbb{D} \cap (0, \infty)$, and $p \in \mathbb{N}$, we can compute some $\widetilde{\surd}(x)[p] \in D$ which is a p-bit approximation to $\sqrt{x}$.

There are well-known efficient Newton-type methods for doing this. However, since we do not care about time bound, we can naively compute z by binary search among the elements of $\mathbb{D}_{p+1}[0, \lceil y \rceil]$. More precisely, suppose we know an interval $[a, b]$ containing $\sqrt{x}$ where $a, b \in \mathbb{D}_{p+1}[0, \lceil y \rceil]$. let $c = \lfloor (a+b)/2 \rfloor_{p+1}$ is the nearest element in $\mathbb{D}_{p+1}$ to that is $\le (a+b)/2$. By computing c^2 and computing it to x, we can decide whether $c \ge \sqrt{x}$ or $c \le \sqrt{x}$. If $c^2 \ge x$, then we narrow our interval to $[a, c]$; otherwise we narrow it to $[c, b]$.

(b) We need a basic estimate: *If $|\rho| \le 1/2$ then* $\sqrt{1+\rho} = 1 \pm \frac{5\rho}{8}$.

This amounts to a linearized estimate of the squareroot of $(1+\rho)$, when ρ is small. For instance, this implies that if $|\rho| \le 1/2$ then $\sqrt{1+\rho} = 1 \pm \rho$. The proof is standard, using Taylor's expansion of $\sqrt{1+\rho}$. We leave it as an Exercise.

(c) We are ready to present the OTM for computing $sqrt_0$, assuming an input oracle $\overline{x}$ for $x > 0$, and a precision $p \in \mathbb{N}$. We want to compute some $z \in \mathbb{D}$ such that $z = \sqrt{x} \pm 2^{-p}$.

- STEP 1. Iteratively compute $\overline{x}[2n]$ for $n = 1, 2, 4, 8, \ldots$, until $\overline{x}[2n] \ge 2^{-2n+1}$ holds. This iteration must terminate because $x > 0$.
- STEP 2. Determine the unique ℓ such that $2^{-2\ell+1} \le \overline{x}[2n] < 2^{-2\ell+2}$. This implies that $\ell \le n$. It follows that

$$x \le \overline{x}[2n] + 2^{-2n} < 2^{-2\ell+3}$$

  and

$$x \ge \overline{x}[2n] - 2^{-2n} \ge 2^{-2\ell}.$$

  Hence $2^{-\ell} \le \sqrt{x} < 2^{-\ell+1}$.
- STEP 3. Compute $y := \overline{x}[\ell + p + 4]$. Then $y = x \pm 2^{-\ell-p-4} = x(1 \pm 2^{\ell-p-2})$.
- STEP 4. Using our approximation function $\widetilde{\surd}$ above, compute an approximation $z := \widetilde{sqrt}(y)[p+1] = \sqrt{y} \pm 2^{-p-1}$. Output this z. To see that this output is correct, it suffices to show that

$$\sqrt{y} = \sqrt{x}\sqrt{1 \pm 2^{\ell-p-2}} = \sqrt{x}(1 \pm \frac{5}{8} 2^{\ell-p-2}) = \sqrt{x} \pm 2^{-p-1}.$$

♠

2. (15 Points)
Let $f : [0, 1] \to \mathbb{R}$ be a computable total function. Define $\max(f) = \max\{f(x) : x \in [0, 1]\}$.

(a) Show that $\max(f)$ is a computable real.
(b) Can this be extended to the case where f is partial?

SOLUTION (a) Let m be a recursive modulus function and $\widetilde{f}$ a recursive approximation of f. We compute a p-bit approximation of $\max(f)$ as follows: we compute $n = m(p+1)$ and then compute

$$y^* = \max\{\widetilde{f}(d)[p+1] : d \in \mathbb{D}_n\}.$$

Let x^* be a maximum point, i.e., $f(x^*) = \max(f)$. If $d^* = \lfloor x^* \rfloor_n$, then

$$\begin{aligned} y^* &\geq \widetilde{f}(d^*)[p+1] \\ &\geq f(d^*) - 2^{-p-1} \\ &\geq (f(x^*) - 2^{-p-1}) - 2^{-p-1} = f(x^*) - 2^{-p}. \end{aligned}$$

If $y^* = \widetilde{f}(d)[p+1]$ then $f(x^*) \geq f(d) \geq y^* - 2^{-p-1}$. Thus y^* is an p-bit approximation of $\max(f)$.
(b) If f is partial, then $\max(f)$ may be undefined. For example, $1/x$ is a partial function with nominal domain $[0,1]$ but $1/x$ does not attain its maximum over its proper domain $(0,1]$.

However, if $\max(f)$ is defined, then we claim that $\max(f)$ is computable. Indeed, its proper domain $\texttt{domain}(f)$ is a countable union of open sets. In particular, there is an interval $[a,b]$ such that $\max(f)$ is attained when f is restricted to $[a,b]$. But this implies that $\max(f)$ is computable, using the arguments of part(a). ♠

3. (15 Points)
We know that all computable functions are continuous. Show that the converse is false.

SOLUTION There are several ways to show this.

Perhaps the simplest is to use the result of the previous exercise: $f : [0,1] \to \mathbb{R}$ which is piecewise linear with $f(0) = f(1) = 0$ and one break-point at $x = 1/2$. Also define $f(1/2) = \alpha$ where α is uncomputable. Thus, $\max(f) = \alpha$. By the previous exercise, if f were computable, so would α.

A less satifying one would be to exploit the fact that the domain of a computable is recursively open. If we define $f \ :\subseteq\ \mathbb{R} \to \mathbb{R}$ such that $f(x) = 0$ iff $x \in [0,1]$ and $f(x) = \uparrow$ iff $x \notin [0,1]$. If f is computable then $\texttt{domain}(f) = [0,1]$ would be recursively open. But this means $[0,1]$ is the union of open sets, which is impossible.

As another example of this kind, suppose we insist on an example where $\texttt{domain}(f)$ is an open interval. If α is uncomputable, then we claim that either $(-\infty, \alpha)$ or (α, ∞) is not recursively open. We can then define a constant function g whose domain is one of these two open sets, and then g would be uncomputable. Suppose, for the sake of contradiction, both

$(-\infty, \alpha)$ and (α, ∞) were recursively open. It is then easy to see that there are recursive functions $\phi_L, \phi_R : \mathbb{N} \to \mathbb{D}$ such that for all $n \in \mathbb{N}$,

$$\phi_L(n) \leq \phi_L(n+1) < \alpha < \phi_R(n+1) \leq \phi_R(n)$$

and $\phi_L(n) \to \alpha$ and $\phi_R(n) \to \alpha$ as $n \to \infty$. Then, for any p, we can compute a p-bit absolute approximation of α by computing $\phi_L(n), \phi_R(n)$ for $n = 0, 1, 2, \ldots$ until $\phi_R(n) - \phi(n) < 2^{-p}$, and then we just output $\phi_L(n)$. This proves that α is recursive, contradiction.

♠

4. (15 Points)
Prove Theorem 2.22 (p.59) in the text book. This concerns a sequence of piecewise linear functions f_n that approximate a function $f \in C[0,1]$ computable in polynomial time.

SOLUTION See §9 of our lecture notes, where we give a simpler approach to linear approximations.

♠

5. (30 Points)
In class we discussed the Polynomial Time Hierarchy $\{\Sigma_i^P, \Delta_i^P, \Pi_i^P : i \geq 1\}$. whose initial level is just $\Delta_1^P = P$, $\Sigma_1^P = NP$ and $\Pi_1^P =$ co-NP. Give upper bounds on the complexity of the following languages by placing them in appropriate classes of the Polynomial Time Hierarchy.
(i) Consider finite undirected (unlabeled) graphs. If G is such a graph on n vertices, let $\overline{G} \in \{0,1\}^*$ denote the obvious representation of Boolean matrix that in turn represents G. Thus $|\overline{G}| = n^2$. Clearly, each G has up to $n!$ representations (by permuting the vertices, we get different matrices). Consider the following language:

$$L_1 = \{\overline{G} : \overline{G} \text{ is the lexicographically smallest representation of } G\}.$$

(ii) Consider Boolean functions which are represented by Boolean circuits C. Each C is given by a triple $(G, n, g_1, g_2, \ldots, g_m)$ where
(1) G is a DAG (directed acyclic graph) on the vertex set $V = \{1, \ldots, n+m\}$, such that vertex $n+m$ is a sink of the DAG, and the set of sources is the set $\{1, 2, \ldots, n\}$;
(2) Each of the g_i's indicates a type of Boolean gate: either $\neg$ (NOT-gate), $\vee$ (OR-gate), or $\wedge$ (AND-gate). We can encoded three gate-types by the integers $0, 1, 2$. The non-source node $n+i \in \{n+1, \ldots, n+m\}$ is associated with the gate-type g_i. The in-degree of a NOT-gate must be 1, and the in-degree of the other types is at least 1.

The circuit $C = (G, n, g_1, \ldots, g_m)$ computes a Boolean function $f_C : \{0,1\}^n \to \{0,1\}$ where we view vertex $n+m$ as the output. The **size** of C is m. Let $\overline{C}$ denote some representation of C. We say $\overline{C}$ and $\overline{C'}$ are equivalent if C and C' computes the same Boolean function. E.g., G can be represented by the Boolean matrix as in part (i). It is can be checked in polynomial time whether $\overline{C}$ is well-formed (i.e., represents a Boolean function as specified above). Consider the language

$$L_2 = \{\overline{C} : \ C \text{ has minimum size among equivalent circuits}\}.$$

SOLUTION (i) Complexity of L_1, where $\overline{G} \in L_1$ iff $\overline{G}$ is lexicographically minimum among the representations of G.

Let ρ be the usual notation for finite undirected unlabelled graphs. Clearly, $L_1 \in \Pi_1^P = \text{co-}NP$ because $w \notin L_1$ iff

$$\begin{array}{ll} P(w): & \text{There is some string } w' \text{ such that} \\ & \text{either } \rho(w) = \uparrow \\ & \text{or } (\rho(w) = \rho(w') \text{ and } w' <_{\texttt{LEX}} w). \end{array}$$

We must determine the predicate $P(w)$ in non-deterministic polynomial time (by guessing w'). Checking if $\rho(w) = \rho(w')$ is actually non-trivial, not known to be deterministic polynomial time, but in NP. If you check $\rho(w) = \rho(w')$ by calling an NP oracle, then you would have shown $L_1 \in \Pi_2^P$.

Here is the trick to do better: we can rewrite $P(w)$ in a way to make the desired result clear.

$$\begin{array}{ll} P(w): & \text{There is some string } w' \text{ and permutation } \pi \text{ such that} \\ & \text{either } \rho(w) = \uparrow \\ & \text{or } (\rho(w) = \downarrow \text{ and } w' = \pi(w) \text{ and } w' <_{\texttt{LEX}} w). \end{array}$$

Here, we may assume that the vertices of the graph $\rho(w)$ are $1, \ldots, n$, and π represents some permulation of $1, \ldots, n$. Moreover, n is at most $|w'|$.

(ii) Complexity of L_2, where $\overline{C} \in L_1$ iff $\overline{C}$ is lexicographically minimum among circuits equivalent to C.

Let ρ be a "usual" notation for circuits. Unfortunately, the previous trick (of guessing π at the same time as guessing w') does not work. That is because the equivalence of two circuits does not seem to have compact witnesses (like π is a witness for isomorphic graphs). To check if two circuits C and C' are equivalent, we basically have to try ALL inputs to C and C' and see if they always give the same output. So we are content with the following predicate: $w \notin L_2$ iff

$$\begin{array}{ll} P(w): & \text{There is some string } w' \text{ such that} \\ & \text{either } \rho(w) = \uparrow \\ & \text{or } (\rho(w) \equiv \rho(w') \text{ and } w' <_{\texttt{LEX}} w). \end{array}$$

where "$C \equiv C'$" means that for all inputs $x \in \{0,1\}^n$, $C(x) = C'(x)$. Clearly, the predicate $\rho(w) \equiv \rho(w')$ is in Π_1^P. Hence co-L_2 is in $NP^{\Pi_1^P} = NP^{NP} = \Sigma_2^P$. Hence $L_2 \in \Pi_2^P$.

♠