Problem Set 1 Solutions

Problem 1 *Call a number $x \in \mathbb{N} = \{1, 2, 3, \ldots\}$ a* ***palindromic number*** *if, written as a decimal string X without leading zeros, it's a palindrome ($X = X^R$). Write a formula for D_n, the number of n-digit palindromic numbers. By induction, prove your formula correct. What is D_{20}?*

Let's enumerate: $D_1 = 9$, $D_2 = 9$, $D_3 = 90$, $D_4 = 90$, $D_5 = 900$, $D_6 = 900$, and so on, so evidently $D_{20} = 9 \cdot 10^9$ and, in general, $D_n = 9 \cdot 10^{\lfloor (n-1)/2 \rfloor}$. We use induction to prove that this formula is true in general. As already mentioned, it holds for D_1 and D_2, the basis of our induction. There are multiple ways to set up the inductive step; here's one. We observe that for even n greater than or equal to 2, $D_{n+1} = 10 \cdot D_n = 10 \cdot 9 \cdot 10^{\lfloor (n-1)/2 \rfloor} = 9 \cdot 10^{\lfloor (n+1)/2 \rfloor} = 9 \cdot 10^{\lfloor n/2 \rfloor}$ because there are ten possible values for the middle digit (justifying the first equality) and n is even (justifying the last equality). Similarly, for even n greater than or equal to 2 we have that $D_{n+2} = 10 \cdot D_n = 10 \cdot 9 \cdot 10^{\lfloor (n-1)/2 \rfloor} = 9 \cdot 10^{\lfloor (n+1)/2 \rfloor}$ because there are ten possible values for the middle-two digits (justifying the first equality).

Problem 2

(a) *Professor Lazybones' computer has two print queues (queue-1 and queue-2) for the same printer. The printer is in a room that's a 1-minute from his office. One or both of the queues goes down unpredictably, and the professor wants to see which of the print queues, if any, is up. The professor has a one-page file that's open on his desktop and decides to use it to test each queue. He could spool the file to queue-1, walk to the printer, see if the printout is there, walk back to his office, and then do it all again, now spooling the file to queue-2. But this will take two trips to the printer room. Describe an alternative strategy that will need only one trip. It should not involve printing anything other than the one open file. Assume the printer does not print header pages and that status of a queue (up or down) doesn't change while the professor is testing it.*

Spool the file one time to queue-1 and twice to queue-2. The number of copies on the printer indicates which of the print queues is up: 0 copies means both queues are down; 1 copy means queue-1 up, queue-2 down; 2 copies means queue-2 up, queue-2 down; and 3 copies means both queues are up.

(b) *Generalize: how do you solve the problem when there are n print queues, $\{1, \ldots, n\}$?*

Send 2^{i-1} copies of the file to print-queue i. Count how many copies get printed. Write it as an n-bit binary number. The bits, numbered right to left starting at 1, indicate the status of that print queue.

(c) *Prove that your (generalized) algorithm is optimal. You will need to define what you mean by optimal, including the class of algorithms with respect to which you claim optimality.*

The class of algorithms I consider are those that send a_i copies of the file to print queue i and then use the number of printouts as a way to reconstruct which print-queues are up. One is allowed to select any nonnegative values for $a_1, \ldots, a_n$; these values define the algorithm. Define its *cost* as the maximal number of pages that might get printed out, which is $a = \sum a_i$. Consider an algorithm optimal if, among all algorithms in the class described, it has least cost.

Now the algorithm we described for part (b) has cost $\sum_{i=1}^{n} 2^{i-1} = 2^n - 1$. We prove one can't do better by using some other $a_1, ..., a_n$. There are 2^n possibilities for the status of each print queue, and if two of these result in the same number of pages printed out, the professor won't know which of the two possibilities is going on. So, for whatever $a_1, \ldots, a_n$ we choose in any correct solution, it will have to be the case that the 2^n values $a_S = \sum_{i \in S \subseteq \{1,\ldots,n\}} a_i$ are all distinct. Enumerate these values $s_1, \ldots, s_{2^n}$ in increasing order: $s_i \in \mathbb{N}$ and $s_i < s_{i+1}$. So $s_1 \geq 0$, so $s_2 \geq 1$, so $s_3 \geq 2$, and so on, so $s_{2^n} \geq 2^n - 1$. In words: the *cost* of any correct algorithm from the class described, s_{2^n}, must be at least $2^n - 1$. But we gave a solution with that cost. So our solution is optimal.

Problem 3. *For each of the following counting problems, give not only your answer, but explain how it is computed, justifying any formulas employed. None of these should be overly tedious or require computer calculation.*

(a) *Let $L = \{\texttt{a}, \texttt{b}, \texttt{ab}\}$. How many strings of length* 10 *are in L^*?*

The answer is 1024, since $\{\texttt{a}, \texttt{b}, \texttt{ab}\}^* = \{\texttt{a}, \texttt{b}\}^*$.

(b) *Let $L = \{\texttt{a}, \texttt{bb}\}$. How many strings of length* 10 *are in L^*?*

The answer is 89. Let $f(n)$ be the number of strings in L^* of length n. We have that $f(0) = 1$, $f(1) = 1$, $f(2) = 2$, and, for $n \geq 3$, that $f(n) = f(n-1) + f(n-2)$. The recurrence follow is because the number of ways to make a string in L^* of length n is the number of ways to make a string in L^* of length $n-1$ (follow it by $\texttt{a}$) plus the number of ways to make a string in L^* of length $n-2$ (followed it by $\texttt{bb}$). We are not overcounting because every nonempty string $w \in L^*$ **uniquely** parses as $w = xy$ with $y \in \{\texttt{a}, \texttt{bb}\}$. Our recurrence is the Fibonacci series, $f(0), f(1), f(2), \ldots = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$.

Problem 4. *Consider the infinite set of numbers $S = \{1, 10, 100, 1000, 10000, \ldots\}$. Prove that there are two numbers in S that differ by a multiple of $N = 314159265359$. Hint: pigeonhole principle.*

The claim is true for *any* infinite set of numbers S and any nonzero number N. Let $S = \{s_1, s_2, s_3, \cdots\}$ be an enumeration of S and let $T = \{s_1 \bmod N, s_2 \bmod N, s_3 \bmod N, \ldots\}$. Since $T \subseteq [N] = \{0, 1, \ldots, N-1\}$ and $|S| > N$ there exists, by the pigeonhole principle, $1 \leq i < I \leq N+1$ such that $s_i \bmod N = s_I \bmod N$. But then $s_I - s_i \equiv 0 \bmod N$, that is, s_i and s_I differ by a multiple of N.

Problem 5 Recall the DIOPHANTINE EQUATION problem: given a multivariate polynomial P with integer coefficients, determine if P has an integer root.

(a) *Prof. Rogaway claimed without proof that there is no algorithm to answer this question. But suppose I give you an oracle (a "magic box") to answer it. In a single computational step, it says yes or no according to whether or not P has an integer root. Given such an oracle, describe an algorithm that finds an integer root of any multivariate polynomial that has one (and reports No Root otherwise).*

The following oracle-using algorithm finds a root or identifies that there is none:

```
Algorithm FindRootWithHelpOfDecisionOracle ($P(x_1, \ldots, x_k)$)
if Oracle($P$)=no then return No Root
for $m \leftarrow 1$ to $\infty$ do
    for every $c_1, \ldots, c_k \in \{-m, \ldots, m\}$ do
        if $P(c_1, \ldots, c_k) = 0$ then return $(c_1, \ldots, c_k)$
```

Notice that (despite the apparently infinite **for** loop) the pseudocode above always **does** terminate: if P has a root then there is a smallest m such that P has a root in which every variable has magnitude at most m, and the algorithm will terminate when it gets to that m.

The technique used above is called *dovetailing* (I am hoping that you "invented" the method for solving this problem). We will use dovetailing quite a bit in the second half of this course.

Here now is an *alternate* solution a student suggested during office hours. If P is a polynomial over the integers with a formal variable x, and if a is an integer, let $P|_{x=a}$ be the polynomial (with one fewer formal variables) that we get by substituting a for x and simplifying. Consider the oracle-using algorithm

```
**Algorithm** AnotherWayToFindRootWithHelpOfDecisionOracle ($P(x_1, \dots, x_k)$)
**if** Oracle($P$)=no **then return** *No Root*
**for** $i \leftarrow 1$ **to** $k$ **do**
    **for** $a \leftarrow 1$ **to** $\infty$ **do**
        **if** Oracle($P|_{x_i=a}$) **then** $c_i \leftarrow a$, $P \leftarrow P|_{x_i=a}$, **break**
        **if** Oracle($P|_{x_i=-a}$) **then** $c_i \leftarrow -a$, $P \leftarrow P|_{x_i=-a}$, **break**
**return** $(c_1, \dots, c_k)$
```

The **break** statement means to exit the immediately enclosing **for** statement.

(b) *Let $s(n)$ be the maximum number of computational steps that your algorithm takes to run (on some fixed, oracle-containing computer) when the polynomial P is described by a string of length n. Explain why there is no algorithm to compute $S(n)$ for any function S such that $S(n) \geq s(n)$ for all n. (In brief, some functions just grow too fast to be computable.)*

Suppose for contradiction that there exists an algorithm FindS that calculates an $S(n)$ as above. Then consider the following algorithm that uses FindS to solve the Diophantine Equation problem:

```
**Algorithm** HasRoot($P(x_1, \dots, x_k)$)
Let $n$ be the length of the description of $P$
**for** $m \leftarrow 1$ **to** FindS($n$) **do**
    **for** every $c_1, \dots, c_k \in \{-m, \dots, m\}$ **do**
        **if** $P(c_1, \dots, c_k) = 0$ **then return** yes **return** no
```

This algorithm correctly decides if P has a root under the assumption that FindS(n) bounds the running time of FindRootWithHelpOfOracle. That's because for any P that has a root, we are certain to run with the m-value that will cause some root to be identified. Since we know (from my unproven claim) that it is impossible to make an algorithm that decides if P has a root, it must be the case that it is impossible to compute FindS as well.

Problem 6. *State whether the following propositions are true or false, carefully explaining each answer.*

(a) *$\emptyset^*$ is a language.* True. It's the language $\{\varepsilon\}$.

(b) *ε is a language.* False. It is a string, not a language. Of course $\{\varepsilon\}$ is a language, but that is quite different from ε.

(c) *Every language is infinite or has an infinite complement.* True. If L is infinite we are done, and if L is finite and over some alphabet Σ then $\overline{L} = \Sigma^* - L$ is an infinite set less a finite set, which is finite.

(d) *Some language is infinite and has an infinite complement.* True. Let $L = \{w \in \{0,1\}^* : \ |w| \text{ is odd}\}$.

(e) *The set of real numbers is a language.* False. A language is a set of strings. A real number is not a string. Some real numbers can be *represented* as strings, but not every real number can be represented by a string, because strings are finite.

(f) *There is a language that is a subset of every language.* True. The empty set $\emptyset$ is such a language.

(g) *The Kleene closure (the star) of a language is always infinite.* False. There are two counterexamples: $\emptyset$ and $\{\epsilon\}$, both of which have the finite set $\{\epsilon\}$ as their Kleene-star.

(h) *The concatenation of an infinite language and a finite language is always infinite.* False. The only exception is concatenating an infinite language to the empty set. The concatenation of an infinite language and a nonempty language is indeed infinite.

(i) *There is an infinite language L containing the emptystring and such that L^i is a proper subset of L^* for all $i \geq 0$.* True. An example is $L = \{1^i : \ i \text{ is zero or a power of } 2\}$. Then $L^* = \{1\}^*$ but L^i contains only strings whose of length m where m, when written in binary, contains i or fewer 1's.