Solution to Problem Set 3

April 3, 2013

1. (a) Let $Q_x$ be program in programming language $Q$ such that $Q_x() = x$ and the length of $Q_x$ is $K_Q(x)$. We will show that there exists a program $P_x$ in programming language $P$ such that $P_x() = x$ and the length of $P_x$ is at most $K_Q(x) + c_{PQ}$, where constant $c_{PQ}$ depends only on programming languages $P$ and $Q$ and not on $x$. We construct $P_x$ to do the following: it defines string containing $Q_x$ (source code for programming language $Q$) and then simulates programming language $Q$ on the string containing $Q_x$ (it can do that by assumption that $P$ and $Q$ are Turing-universal) and outputs whatever $Q_x()$ would output. Now notice that the simulation rules can be described in constant space and $Q_x$ is of length $K_Q(x)$.

(b) To compute $K(x)$ using a Halting Problem oracle, enumerate all the programs in increasing lexicographic order, and for each one call the oracle to check if it halts. For each program that does halt, run it and check whether it prints $x$; if it does, output its length and halt.

2. (a) We build a tree as follows. The root of the tree contains a single empty tiling of square of size $0 \times 0$. Now if a vertex of the tree contains a tiling $A$ of size $n \times n$ then it has a successor containing a tiling $B$ of size $(n + 2) \times (n + 2)$ if and only if the center of $B$ contains $A$. We notice that each vertex of the tree have finite degree. Now by König’s lemma and the fact that any finitely large square can tiled we get that there must be a path of infinite length (which gives a tiling for entire the plane).

(b) Consider the following tiling for a $1 \times 1$ square:

```
0
3 1
2
```

Now consider the following tiling for a $2 \times 2$ square:

```
<table>
<thead>
<tr>
<th>4</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>15</td>
</tr>
</tbody>
</table>
```
```
<table>
<thead>
<tr>
<th>8</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>13</td>
</tr>
</tbody>
</table>
```
```
<table>
<thead>
<tr>
<th>5</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>
```
We can use this basic pattern to generate a tiling for any \( n \times n \) square; beginning in the top left tile, we add 1 at the top of the tile, moving clockwise around the tile and moving around the tiles clockwise, making sure that we never break the constraint that adjacent edges have to match up. Notice that if we do this, the smallest number that appears in any tiling is going to be the top number of the top left tile. Thus, it’s clear that we can’t reuse the tiles from smaller tilings to make larger ones.

We can generate an infinite set of tiles this way and generate tilings for arbitrarily large \( n \times n \) squares. But for a given \( n \), we have to choose that top left square, which will “commit” us to that \( n \); once that square is chosen, we can’t tile a larger square with those tiles. Thus, we can’t tile the infinite plane with this system.

3. (a) Given \( P \), we build an oracle TM \( Q \) that enumerates all the strings and calls the oracle on \( \langle(P), x \rangle \) for each string \( x \). If the oracle rejects, \( Q \) accepts; if the oracle accepts, \( Q \) moves on to the next string. By construction, \( Q^{\text{HALT}}(\varepsilon) \) halts iff \( P \) runs forever on some input.

Now we feed \( \langle(Q), \varepsilon \rangle \) to the \( \text{SUPERHALT} \) oracle and accept \( P \) iff the oracle says that \( \langle(Q), \varepsilon \rangle \in \text{SUPERHALT} \).

(b) Given \( \langle(M), x \rangle \), we build a TM \( P \) that runs on input \( k \) as follows: \( P \) runs \( M \) on \( x \) for \( k \) steps and keeps a list \( L \) of oracle queries made by \( M \). Each time \( M \) makes an oracle query \( \langle(Q), y \rangle \), \( P \) adds the query to \( L \) and then tries to answer the query by running \( Q \) on \( y \) for \( k \) steps:

- If \( Q \) halts on \( y \) within those \( k \) steps, \( P \) writes the answer “yes” on the oracle tape of \( M \), takes \( \langle(Q), y \rangle \) off the list \( L \), and continues running \( M \) on \( x \).
- If \( Q \) does not halt on \( y \) within \( k \) steps, \( P \) writes “no” on the oracle tape of \( M \) and continues running \( M \) on \( x \). But because we don’t know for sure that \( Q \) doesn’t halt on \( y \), we keep \( \langle(Q), y \rangle \) in the list \( L \) so we can check it later.

If after \( k \) such simulated steps \( M \) has not yet halted on \( x \), \( P \) halts. If \( M \) did halt on \( x \), \( P \) continues running all the oracle queries it saved for later, in round-robin: if \( L = \{\langle(Q_1), y_1\rangle, \ldots, \langle(Q_m), y_m\rangle\} \), then \( P \) simulates one step of \( Q_i \) on \( y_i \) for each \( i = 1, \ldots, m \) and repeats. If some \( Q_i \) halts, \( P \) halts too. Otherwise \( P \) continues this simulation forever. (If \( L \) is empty when \( M \) halts, \( P \) simply loops.)

Now we take \( P \) and feed it to the \( L \)-oracle, and accept the original input \( \langle(M), x \rangle \) iff the \( L \)-oracle accepts \( P \).

The correctness of this reduction is based on the fact that if \( M \) halts on \( x \) it does so in finitely many steps and makes finitely many oracle queries, and if \( \langle(Q), y \rangle \) is an oracle query that returns “yes” then again \( Q \) halts on \( y \) in finitely many steps. Take \( k \) to be the maximum number of steps required for \( M \) to halt on \( x \) or, for any query \( \langle(Q), y \rangle \in \text{HALT} \) that \( M \) makes, for \( Q \) to halt on \( y \). Then \( P \) loops on \( k \) and on any number greater than \( k \).

4. (a) No, \( M_F \) doesn’t halt: if it halts then \( G(F) \) is provable in \( F \), which means (since \( F \) is sound) that \( G(F) \) is true. But that means \( G(F) \) is not provable in \( F \), contradiction...
(b) Neither. If $F$ proves that $M_F$ halts, then because $F$ is sound, $M_F$ really halts – but we saw in (a) that it doesn’t halt. On the other hand, if $F$ proves that $M_F$ doesn’t halt, then $F$ also proves that there’s no proof of $G(F)$ (since, had there been a proof, $M_F$ would have found it and halted). But this means that $F$ proves $G(F)$, which is again a contradiction!

(Technical note: Here, we made one implicit assumption, that it’s possible to prove in $F$ that $M_F$ halts if and only if there’s a proof of $G(F)$ in $F$. Fortunately, such an equivalence will be easy to prove, in just about any proof system strong enough for Gödel’s Theorem to apply to it. For, as we saw in class, we can encode the computation of a Turing machine as a number, and write logical statements like $P(x) =$ “after the computation represented by number x, the TMs tape contains a valid proof in $F$”, etc.)

(c) If $n \geq k_F$, we can build a “Gödelian machine” with $n$ states that is equivalent to $M_F$ (we can just not use the extra states if $n > k_F$). Call this machine $M'$. Suppose that $F$ proves “$BB(n) = c$” for some constant $c$. Then $F$ proves that

(*) If $M'$ halts, it halts in at most $c$ steps.

Any reasonable proof system is also able to prove that a given machine halts or does not halt in $c$ steps. So $F$ can prove that $M'$ halts or does not halt ( whichever statement is true) by combining (*) with the first $c$ steps of $M'$.

(d) Here is a 2-state TM that runs for 6 steps:

- State 0: if the tape reads 0, write 1 and move right. If the tape reads 1, write 1 and move left. Go to state 1 in both cases.
- State 1: if the tape reads 0, write 1 and move left. If the tape reads 1, halt.

On the all-zero tape the machine runs as follows (alternating between state 0 and state 1):

- (State 0, $\hat{0}$): Write 1, move right.
- (State 1, $\hat{0}$): Write 1, move left.
- (State 0, $\hat{1}$): Write 1, move left.
- (State 1, $\hat{0}$): Write 1, move right.
- (State 0, $\hat{1}$): Write 1, move left.
- (State 1, $\hat{1}$): halt.

5. (a)

\[
\begin{align*}
2^{n^{o(1)}} & \rightarrow 2^{o(n)} \rightarrow O(2^{n}) \rightarrow 2^{O(n)} \rightarrow 2^{n^{O(1)}} \\
\downarrow & \uparrow \downarrow \uparrow \downarrow \uparrow \downarrow \uparrow \downarrow \uparrow \\
3^{n^{o(1)}} & \rightarrow 3^{o(n)} \rightarrow O(3^{n}) \rightarrow 3^{O(n)} \rightarrow 3^{n^{O(1)}}
\end{align*}
\]

(b) Solving for $t$, we obtain $n = \Omega(2^{\sqrt[4]{\log(t)}})$. (A derivation is presented as an appendix at the end of the solutions.)

6. (a) The set of finite sets of integers is countable set. We can order finite set of integers and obtain a finite sequence of integers. There are only countably many ordered finite sequences
of integers. (We can encode any finite sequence of integers uniquely as a finite binary string and a subset of finite binary strings is countable.)

(b) The set of languages $L \subseteq \{0, 1\}^*$ in the complexity class P is countable set. For each language in P there must be a corresponding Turing machine. Now the answer follows from the fact that the set of Turing machines is countable set.

(c) The set of languages $L \subseteq \{0, 1\}^*$ not in the complexity class P is uncountable set because the set of all languages is uncountable set (Cantor’s diagonal argument) and the set of all languages in P is countable set.

(d) The set of languages to which $HALT$ is Turing-reducible is uncountable set. Take any language $L$. It suffices to construct a unique language $L'$ corresponding to $L$ such that $HALT$ is Turing reducible to $L'$ because the set of of possible languages $L$ is uncountable set. We construct $L' = \{0x|x \in L\} \cup \{1x|x \in HALT\}$. Clearly, $HALT$ is Turing-reducible to $L'$ (append 1 in front of the input string and ask whether the resulting string is in $L'$ and output the answer).

(e) The set of languages Turing-reducible to $HALT$ is countable because for every such language there must be a corresponding oracle Turing machine (with access to oracle $HALT$) and the set of oracle Turing machines is countable set.

7. By definition $EXP \subseteq NEXP$. Suppose that $L \in NEXP$ and NDTM $M$ decides it. We claim that then language $L' = \{(x, 1^{2|x|^z})|x \in L\} \in NP$. NDTM $M'$ decides $L'$: given input $y$, first check if there is a string $z$ such that $y = (z, 1^{2|x|^z})$. If not, reject. If $y$ is of this form, then simulate $M$ on $z$ for $2^{|z|^z}$ steps and output its answer. Clearly, the running time is polynomial in $|y|$, and hence $L' \in NP$. Hence if $P = NP$, then $L'$ is in P. But if $L'$ is in P then $L$ in in EXP: To determine whether an input $x$ in in $L$, we just pad the input and decide whether it is in $L'$ using the polynomial-time machine for $L'$. (This proof is from [1].)

8. Suppose for the sake of contradiction that $PSPACE = \text{TIME}(2^n)$. We will show that $PSPACE = \text{TIME}(2^{n^2})$, which violates time hierarchy. Clearly, $\text{TIME}(2^n) \subseteq \text{TIME}(2^{n^2})$. Let $L \in \text{TIME}(2^{n^2})$. We will show that $L \in \text{TIME}(2^n)$. We construct $L' = \{x\#0|x|^2-|x|-1|x \in L\}$. Now $L' \in \text{TIME}(2^n)$ because we can first check that the input is of correct form (it can be written as $z\#0|x|^2-|z|-1$ for some $z$) and then run TM of language $L$ on $x$. By our assumption $L' \in PSPACE$. Let TM $M$ be such that $L(M) = L'$. We get that $L \in PSPACE$. (Given input $x$ pad it to get $x\#0|x|^2-|x|-1$ and run $M$ on the resulting string.)
Appendix: Derivation of solution to 5b:

By definition, \( t = O(n^{\log n}) \) is the same as saying

\[
\exists c, n_0 \text{ such that } \forall n > n_0, t \leq c \times n^{\log n}
\]

we can rewrite this condition:

\[
\exists c, n_0 \text{ such that } \forall n > n_0, \frac{t}{c} \leq n^{\log n}
\]

\[
\exists c, n_0 \text{ such that } \forall n > n_0, \log \frac{t}{c} \leq \log n^{\log n}
\]

\[
\exists c, n_0 \text{ such that } \forall n > n_0, \log \frac{t}{c} \leq \log n \times \log n
\]

\[
\exists c, n_0 \text{ such that } \forall n > n_0, \log \frac{t}{c} \leq (\log n)^2
\]

\[
\exists c, n_0 \text{ such that } \forall n > n_0, \sqrt{\log \frac{t}{c}} \leq \log n
\]

\[
\exists c, n_0 \text{ such that } \forall n > n_0, n \geq 2^{\sqrt{\log \frac{t}{c}}}
\]

Notice that for any \( c_0 > 0 \) there will be some constant \( n_0 \) for which the original inequality will hold. Thus, if we set \( c = 1 \), we get one potential solution: \( f(t) = 2^{\sqrt{\log t}} \).

References