1

Fill in the holes (boldfaced "WHY"s) in the posted proof of the Baker-Gill-Solovay Theorem (notes for Lecture 5).

**WHYNOT-1:** Had we done so, we would still have that $P^{\text{EXPCOM}} \subseteq NP^{\text{EXPCOM}}$ and that $EXP \subseteq P^{\text{EXPCOM}}$. However, the foregoing proof that $NP^{\text{EXPCOM}} \subseteq EXP$ would not go through. We have an input $x$ of length $n$ for which we are trying to determine membership in $L(W^{\text{EXPCOM}})$. In time $p_1(n)$, $W$ could construct a query $(M, y, t)$, where $t = 2^{p_1(n)}$, because writing down such a $t$ requires simply writing down one 1 followed by $p_1(n)$ zeroes. Simulating $M$ on input $y$ for $2^t$ steps would, in this case, require time $2^{2^t}$, which is doubly exponential in the length of $x$, not singly exponential in $|x|$ as we need it to be for this proof.

**WHY-2:** To determine whether $1^n$ is in $U_B$, an NP base machine can simply guess an $x$ of length $n$ and ask the oracle whether it is in $B$.

**WHY-3:** Although $B$ has not been completely defined by stage $i$, all of the answers to queries made by $M_i$ on input $1^n$ within $2^n/10$ steps have been determined. So, if $M_i$ halts within $2^n/10$ steps, it has all of the information it needs to make a decision about $1^n$.

**WHY-4:** Before stage $i$ begins, no strings of length $n$ have been put into $B$. So (*) cannot require the answer to a query of length $n$ to be “YES.” During the $(2^n/10)$-step simulation in stage $i$, the only queries (of any length) to which answers are fixed are those fixed in (**), and all of those answers are “NO.” So no strings of length $n$ are put into $B$ before stage $i$ gets to case 1.

**WHY-5:** Before stage $i$, membership in $B$ has not been determined for any strings of length $n$. During the $(2^n/10)$-step simulation in stage $i$, membership can be fixed for at most $2^n/10$ strings, of length $n$ or any other length, by (**). There are $2^n$ strings of length $n$; thus, membership in $B$ has not yet been decided for at least one of them (indeed for at least 90% of them) by the time stage $i$ gets to case 2.
WHY-6: At this point in the proof, membership has only been fixed for those $q$ that are considered in (**) for some stage $i$ and for (possibly not all) strings in $\{0,1\}^n$, where $n$ is the length used in some stage $i$. Clearly, this need not be all of $\{0,1\}^*$. To complete the definition of $B$, we can just take any string whose membership has not yet been determined to be a non-member of $B$. Of course, there are other ways to complete the definition; the point is that a membership bit must be defined for each string.

2

Prove that there is an oracle $A$ such that $\text{coNP}^A = \text{NP}^A$ and an oracle $B$ such that $\text{coNP}^B \neq \text{NP}^B$.

We begin by showing that $\exists A$ such that $\text{NP}^A = \text{coNP}^A$. Take $A = \text{TQBF}$. First we show that $\text{NP}^A = \text{P}^A$. Clearly, $\text{P}^A \subseteq \text{NP}^A$ for any oracle $A$. For the other direction, $\text{NP}^{\text{TQBF}} \subseteq \text{NPSPACE}^{\text{TQBF}} \subseteq \text{NPSPACE}$ since we can compute instances of $\text{TQBF}$ in polynomial space instead of asking the oracle. From Savitch’s theorem, we know that $\text{NPSPACE} \subseteq \text{PSPACE}$. Finally, Theorem 4.13 shows that $\text{TQBF}$ is $\text{PSPACE}$-complete, which gives us $\text{NP}^{\text{TQBF}} = \text{P}^{\text{TQBF}}$. Similarly, $\text{coNP}^A = \text{coP}^A$. Also, trivially, $\text{P}^A = \text{coP}^A$. Summing up, we have: $\text{NP}^A = \text{P}^A = \text{coP}^A = \text{coNP}^A$.

Next, we show that $\exists B$ such that $\text{NP}^B \neq \text{coNP}^B$. For any oracle $B$, let $L_B$ be the set of all strings $w$ for which no string of equal length appears in $B$. That is, $L_B = \{w | \exists x \in B, |x| = |w|\}$.

For any $B$, it holds that $L_B \in \text{coNP}^B$. We show this by proving that $L_B \in \text{NP}^B$. Given input $w$, a non-deterministic oracle $M$ guesses a string $x$ of length $|w|$ and checks if $x \in B$. If so, it outputs 1, else 0. If $w \notin L_B$, there exists an $x$ of the same length, so $M$ accepts. Similarly, if $w \notin L_B$, then $M$ always rejects.

We now define $B$ so that $L_B \notin \text{NP}^B$. Let $M_1, M_2, \ldots$ be the list of all NDTMs. We assume that $M_i$ runs in time at most $n^i$. We set $B = \emptyset$. Now, at step $i$, let $n$ be a number larger than any string that has been classified (we know whether it is in $B$ or not), and $2^n > n^i$. $M_i$ is run on input $1^n$. For every oracle query $y$ that $M_i$ makes: if $y \in B$ can be determined, then answer appropriately, else, answer “no” and set $y \notin L_B$.

Now, if $M_i$ rejects on every branch, then all strings $y$ of length $n$ are not in $B$ (so $\exists x \in B$ of length $n$ and $1^n \in L_B$). Otherwise, there must exist a branch where $M_i$ accepts $1^n$. We fix this branch and set $y \in \{0,1\}^n$ be a string that was not queried. Then, we define $y \in B$ and remove all other
strings of length $n$ from $B$. (Note that setting $y \in B$ does not change the
decision on the fixed path.)

Now that we have defined $B$, we claim that no $\text{NP}$-machine decides $B$.
Let $M$ be an $\text{NP}$-machine. Then, $M = M_i$ for some $i$. Now, if $M_i(1^n)$ is a
case where all branches reject, then by the construction of $B$, we have that
$1^n \in L_B$, so $M_i$ does not decide $B$. If not all branches reject, then there is at
least one accepting path even where $y \in B$ with $|y| = n$. Thus, $M_i$ accepts
even though $1^n \not\in B$ (as we just said, $\exists y \in B | |y| = n$). So, $M_i$ does not
decide $B$.

3 Problem 4.4

Show that the following language is $\text{NL}$-complete:

$$\{\langle G, s, t \rangle : G \text{ is a strongly connected digraph} \}$$

Denote this set by $\text{SCD}$ (for “strongly connected digraph”). First, we
must show that $\text{SCD}$ is in $\text{NL}$. Assume that $G$ is given as an adjacency
matrix and that $V(G) = \{1, \ldots, n\}$. Recall that, on September 18, 2012, we
gave the following $\text{NL}$ procedure to recognize the set $\text{PATH}$, $i.e.$, the set of
triples $(G, s, t)$ such that there is a path from $s$ to $t$ in the digraph $G$.

\begin{verbatim}
PATH(G, s, t)
{
    i ← 0;
    u ← s;
    WHILE(i ≤ n)
    {
        IF (u = t) THEN OUTPUT(ACCEPT) AND HALT;
        GUESS u′ ∈ V(G);
        IF ((u, u′) ∈ A(G)) THEN u ← u′;
        i ← i + 1;
    }
    OUTPUT(REJECT) AND HALT;
}
\end{verbatim}

To recognize $\text{SCD}$, we simply run $\text{PATH}$ on each pair of nodes in $G$.

\begin{verbatim}
SCD(G)
{
    FOR 1 ≤ s ≤ n
\end{verbatim}
FOR $1 \leq t \leq n$

IF (PATH$(s, t) = \text{REJECT}$) THEN OUTPUT(REJECT) AND HALT;
OUTPUT(ACCEPT) AND HALT;

}  

Note that this procedure SCD is nondeterministic, because PATH is nondeterministic. Moreover, because PATH runs in logspace, SCD requires only logspace, because each of $s$ and $t$ can be written down in $O(\log n)$ bits, and we can reuse the space we used for one $s$-$t$ pair when we do the computation for the next pair.

To show that every problem in NL is logspace-reducible to SCD, it suffices to show that PATH is logspace-reducible to SCD. Let $(G, s, t)$ be a PATH instance, where $G = (V, A)$. The corresponding instance $f(G, s, t)$ of SCD is $G' = (V, A')$, where $A'$ contains all arcs in $A$, plus all arcs of the form $(v, s)$, where $v \neq s$, and all arcs of the form $(t, v)$, where $v \neq t$.

First, we must show that $(G, s, t) \in \text{PATH}$ if and only if $G' \in \text{SCD}$. If $(G, s, t) \in \text{PATH}$, then there is a path $s \rightarrow \cdots \rightarrow t$ in $G$. Thus, for any $v$ and $w$ in $V$, there is a path $v \rightarrow s \rightarrow \cdots \rightarrow t \rightarrow w$ in $G'$; this means that $G' \in \text{SCD}$. On the other hand, if $G' \in \text{SCD}$, then there is a path from $v$ to $w$ for any pair of nodes $v$ and $w$ in $V$, and, in particular, there is one from $s$ to $t$. But what is a path $v_0 = s \rightarrow v_2 \rightarrow \cdots \rightarrow v_\ell = t$ in $G'$ from $s$ to $t$ used one or more arcs that were not in the original instance’s graph $G$. If it did, then it would not be a simple path, i.e., it would visit $s$ or $t$ or both more than once, because all of the arcs in $G'$ that are not in $G$ have $s$ or $t$ as an endpoint. Such a path must contain a subpath $v_i \rightarrow \cdots \rightarrow v_j$ that starts at $v_i = s$, ends at $v_j = t$, and has no occurrence of $s$ or $t$ between $v_i$ and $v_j$. That subpath is a path from $s$ to $t$ in the original instance.

Next, we must show that the reduction $f$ is implicitly logspace-computable. If $x$ is a PATH instance that contains an $n$-node graph $G$, then $|x| = n^2 + 2 \log n + c$ for some constant $c$ — $n^2$ bits for the adjacency matrix of $G$, $2 \log n$ for $s$ and $t$, and $c$ bits for “delimiters” between the adjacency matrix and $s$ and between $s$ and $t$. The length of SCD instance $f(x)$ is just $n^2$ — all one needs is the adjacency matrix of $G'$. Thus, the set $L'_f = \{(x, i) \text{ such that } i \leq |f(x)|\}$ is clearly computable in logspace. Similarly, the set $L_f$ of pairs $(x, i)$ such that the $i^{th}$ bit of $f(x)$ is 1 is also computable in logspace: If $i$ is the index of a pair $(v, s)$ with $v \neq s$ or of a pair $(t, w)$ with $t \neq w$, then $(x, i) \in L_f$; otherwise, it is part of the input to the PATH instance and can simply be read off the input tape.
4 Problem 4.8

Define a function \( f : \{0, 1\}^* \to \{0, 1\}^* \) to be write-once logspace computable if it can be computed by an \( O(\log n) \)-space TM \( M \) whose output tape is “write-once” in the sense that, in each step, \( M \) can either keep its head in the same position on that tape or write to it a symbol and move one location to the right. The used cells of the output tape are not counted against \( M \)’s space bound.

Prove that \( f \) is write-once logspace computable if and only if it is implicitly logspace computable in the sense of Definition 4.16.

\((\Rightarrow)\) Suppose \( f \) is write-once logspace computable. Let \( M \) be a write-once TM computing \( f \) in logspace. For any input \((x, i)\), we can compute bit \( i \) of \( f(x) \) in logspace by simply simulating \( M \) without its output tape and keeping a count of how many times \( M \) has written to its write-once output tape. After \( M \) writes for the \( i^{th} \) time, we output the bit it just wrote. We can also check whether \( i \leq |f(x)| \) by simply checking whether \( M \) halts before our counter reaches \( i \). Hence \( f \) is implicitly logspace computable.

\((\Leftarrow)\) Suppose \( f \) is implicitly logspace computable. Let \( M \) be a logspace TM recognizing \( \{(x, i) | f(x) = 1 \} \) and \( M' \) be a logspace TM recognizing \( \{(x, i) | i \leq |f(x)| \} \). Given an input \( x \), we can output \( f(x) \) in a write-once fashion by simply running \( M \) on \((x, 1)\) and outputting the result, and then running \( M \) on \((x, 2)\) and outputting the result, etc. We continue doing this until we reach \((x, i)\), where \( i \not\leq |f(x)| \) (which is checked each time using \( M' \)), at which point we halt with precisely \( f(x) \) on the output tape. Note that this requires a counter for the current bit – this counter is guaranteed to take only log space since \( |f(x)| \) is polynomial in \( |x| \).