If you have not yet turned in the Problem Set, you should not consult these solutions.

1. We need to define a language that has the same computational power as the function \( f \); that is, given \( f \) one should be able to “easily” compute \( L_f \), and vice versa. There are many ways to do this; here is one example:

Define

\[
L_f = \{(x,k,b) : x \in \Sigma^*, b \in \Sigma, k \text{ an integer and the } k\text{-th symbol of } f(x) \text{ is } b\}.
\]

The alphabet \( \Gamma \) thus includes \( \Sigma \) plus extra symbols to encode “(”, “)”, “,”, and integers (which can be written in binary, and thus encoded using two additional symbols).

To determine if a string \( z \) is in \( L_f \) using \( f \), we first determine if \( z \) is of the form \((x,k,b)\). If it is, we evaluate \( f(x) \), and accept iff the \( k \)-th symbol of \( f(x) \) is \( b \).

To compute \( f(x) \) using \( L_f \), we do the following: for \( k = 1, 2, \ldots \), we determine whether \((x,k,b) \in L_f \) for each \( b \in \Sigma \), and we add the \( b \) for which we get a positive answer to the output. We stop when we reach a value of \( k \) for which \( \forall b \ (x,k,b) \notin L_f \) (this marks the end of the string \( f(x) \)).

2. (a) This is an “if and only if” proof so it must be proven in both directions. That is, (1) given a regular language, show that it can be recognized by an all-paths-NFA, and (2) given an all-paths-NFA show that the language it recognizes is regular.

(1) If a language is regular, then by the equivalence theorems there is a FA that recognizes the language. Conveniently, a FA is an instance of an all-paths-NFA: since there is only one computation path for any given input, then strings are accepted only if all paths (the only path) ends in an accept state. So every regular language is recognized by an all-paths-NFA.

(2) To show that a language recognized by an all-paths-NFA is regular, we can construct a FA that recognizes the exact same language as a given all-paths-NFA. This can be done in the same way as the NFA-FA equivalence proof in the lecture slides (or Sipser, Theorem 1.39), with the following change: define the new machine’s accept states \( F' \) to be \( \{R \in Q' | \text{every state in } R \text{ is an accept state of } M\} \). This corresponds to the fact that for the all-paths-NFA to accept a string, every computation on that string must end in an accept state.

(b) If \( A \) and \( B \) are regular languages, then by part (a) there are all-paths-NFAs, \( M_a \) and \( M_b \), that recognize each respectively. Given this fact, we can construct a third all-paths-NFA, \( M_c \), that recognizes \( C = (A \cap B) \). Simply create a start state with epsilon-transitions that point to the start states of \( M_a \) and \( M_b \). A string is accepted by \( M_c \) if and only if all
computations end in accept states, which by the construction means all computations end in accept states in both \( M_a \) and \( M_b \). Therefore \( M_c \) recognizes the intersection of \( A \) and \( B \). By part (a) since \( C \) is recognized by an all-paths-NFA, it is a regular language.

(c) \( L_{\text{flip}} \) is the complement of \( L \). To see this, note that if \( x \) is a string in \( L \), then some computation of \( M \) on \( x \) ends up in an accept state. Since accept states of \( M \) are not accept states in \( M_{\text{flip}} \), and \( M_{\text{flip}} \) requires all computations to end in accept states, \( x \) cannot be a member of \( L_{\text{flip}} \). On the other hand, if \( x \) is not a string in \( L \), then all computations of \( M \) on \( x \) must end in reject states. Since reject states in \( M \) are accept states in \( M_{\text{flip}} \), then all computations of \( x \) on \( M_{\text{flip}} \) end in accept states, and \( x \) is a member of \( L_{\text{flip}} \).

3. Let \( L \) be the language consisting of all palindromes. Suppose \( L \) is a regular language. Then the Pumping Lemma must hold. Let \( p \) be the pumping length for \( L \). Consider \( w = \underbrace{aa \ldots a}_{p} b \underbrace{aa \ldots a}_{p} \in L \). Since \(|w| \geq p\), \( w \) can be written as \( w = xyz \) such that

(a) for every \( i \geq 0 \), \( xy^i z \in L \), and
(b) \(|y| > 0\), and
(c) \(|xy| \leq p\).

By (c), we see that \( xy = aa \ldots a \). Thus

\[
xy^2z = \underbrace{aa \ldots a}_{p+|y|} b \underbrace{aa \ldots a}_{p} .
\]

Since \(|y| > 0\), \( xy^2z \) is not a palindrome, and thus it is not in \( L \). Contradiction. We conclude that the language consisting of all palindromes is not regular.

4. (a) The easiest way to show that \( L_n \) is regular is to show a Finite Automaton \( A \) that accepts it. \( A \) will have \( n \) states \( S_0, S_1, \ldots, S_{n-1} \) where the accept states are all the states except \( S_0 \), which is the start state. The transition function is \( \delta(S_j,0) = S_{(j+1) \pmod{n}} \). In other words, the machine is in state \( s_j \) iff the number of symbols it has read so far, modulo \( n \), is \( j \).

(b) Suppose \( \text{PRIMES} \) is a regular language. Then the Pumping Lemma must hold, so there is a pumping length \( p \) for this language. Let \( q > p \) be a prime, and consider the string \( w = \underbrace{00 \ldots 0}_{q} \), which is in \( \text{PRIMES} \).

By the Pumping Lemma, \( w \) can be written \( w = xyz \), with \(|y| = r > 0\), and for all \( i \geq 0 \), the string

\[
xy^iz = \underbrace{00 \ldots 0}_{q+(i-1)r}
\]

is in \( \text{PRIMES} \). This means that all the numbers \( q + (i-1)r \) must be primes for all \( i \geq 0 \), but for \( i = q+1 \) we have \( q + (i-1)r = q(1+r) \) with both \( q \) and \( r+1 \geq 1 \), which is a composite. Contradiction. We conclude that \( \text{PRIMES} \) is not a regular language.
5. We build a NPDA for $L$. There will be four states, labeled “match,” “add-one,” “add-two,” and “accept” plus three separate state $S$, $R$ and $R'$. State $S$ is the start state and “accept” is the (single) accept state. There is a transition labeled $\epsilon, \epsilon \rightarrow \$$ from $S$ to $R$ (this marks the bottom of the stack with $\$$, as is customary). Then there is a transition $d, \epsilon \rightarrow d$ from $R$ to $R'$ for each $d \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ (we disallow leading zeros, so the first digit read must be a non-zero). There is then a self-loop on $R'$ labeled with $d, \epsilon \rightarrow d$, for each $d \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and a transition labeled $#, \epsilon \rightarrow \epsilon$ from $R'$ to state “add-two”. These transitions cause the digits to the left of the first # to be pushed onto the stack, and the machine ends in state $R'$.

Now we have a transition from “add-two” to “match” labeled with $d + 2, d \rightarrow \epsilon$ for $d \in \{0, 1, 2, 3, 4, 5, 6, 7\}$. This handles the case in which there is no carry, so the least significant digit (which appears first in $N'(i + 2)$) is 2 larger than the least significant digit of $N(i)$, and the remaining digits must simply match.

We also have transitions from “add-two” to “add-one” labeled with $0, 8 \rightarrow \epsilon$ and $1, 9 \rightarrow \epsilon$. These handles the two cases in the least significant digit that generate carries.

We have a transition from “add-one” to “add-one” labeled with $0, 9 \rightarrow \epsilon$. These will match 0’s at the beginning of $N'(i + 1)$ to 9’s on the stack, for the zero or more 9’s leading up to the least significant digit of $N(i)$. We also have a transition from “add-one” to “match” labeled with $d + 1, d \rightarrow \epsilon$ for $d \in \{0, 1, 2, 3, 4, 5, 6, 8\}$, which will then transition to “match” upon matching the first non-9 digit with that digit plus one.

From this point on, we just need to match digits one by one. So, we have transitions from “match” to “match” labeled with $d, d \rightarrow \epsilon$ for $d \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Finally, we have transitions from “match” to “accept” labeled with $\epsilon, \$$ \rightarrow \epsilon$, from “add-one” to “accept” labeled with $1, \$$ \rightarrow \epsilon$. 