FSA

A finite automaton M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is the finite set of states
- Σ is the finite alphabet of symbols
- $\delta: Q \times \Sigma \to Q$ is the state transition **function**
- q_0 is the start state
- F is the set of accept states

accepts a string $w = w_1 w_2 \cdots w_n$ $(w_i \in \Sigma)$ if there is a sequence $r_0, r_1, \ldots, r_n$ of states in Q such that:

1. $r_0 = q_0$, (start in the start state)
2. $r_{i+1} = \delta(r_i, w_{i+1})$ for $0 \le i < n$, and (every transition is legal)
3. $r_n \in F$. (the final state is an accept state)

NFA's

A nondeterministic finite automaton N is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is the finite set of states
- Σ is the finite alphabet of symbols
- $\delta: Q \times (\Sigma \cup \{\epsilon\}) \to \wp(Q)$ is the transition **function**
- q_0 is the start state
- $F \subseteq Q$ is the set of accept states

Every state of an NFA is allowed: outgoing ϵ transitions; multiple outgoing transitions with the same label; zero outgoing transitions for some labels

Notation: $\wp(Q)$ is the powerset (set of all subsets) of Q

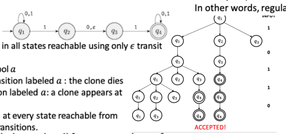
A nondeterministic finite automaton $N = (Q, \Sigma, \delta, q_0, F)$ **accepts** a string w if

i. w can be expressed as $w = y_1 y_2 \ldots y_n$, $\forall i: y_i \in \Sigma \cup \{\epsilon\}$, and

ii. there is a sequence $r_0, r_1, \ldots, r_n$ of states in Q such that:

1. $r_0 = q_0$, (start in the start state)
2. $r_{i+1} \in \delta(r_i, y_{i+1})$ for $0 \le i < n$, and (every transition is legal)
3. $r_n \in F$. (the final state is an accept state)

How an NFA Processes a String



Machine initially in start state

From the start state, clones appear in all states reachable using only ϵ transit

Each clone in its current state:

1. Reads the next input symbol a
2. If there is no outgoing transition labeled a : the clone dies
3. For each outgoing transition labeled a: a clone appears at the next state
4. Each clone spawns a clone at every state reachable from its state following only ϵ transitions.

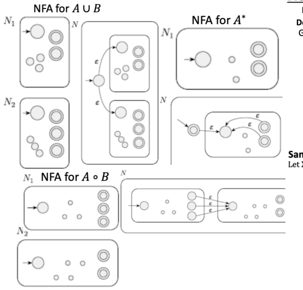
First process the input symbol, then take all free moves thereafter.

While processing the input string, the NFA can be in one of many different states

The NFA accepts the input string if one of its possible states is an accept state.

Given NFAs N_1, N_2 for A, B

NFA for $A \cup B$ — NFA for A^* — NFA for $A \circ B$



NFA recognizable languages are closed under regular operations.

Language

The **language recognized** by FSA M is $L(M) = \{w: M \text{ accepts } w\}$

A language is **regular** iff it is recognized by an FSA.

Let A, B be two languages.

UNION $A \cup B = \{w: w \in A \text{ or } w \in B\}$ $= \{oreo, ginger, cookie, icecream\}$

CONCATENATION $A \circ B = \{xy: x \in A \text{ and } y \in B\}$ $= \{oreocookie, oreoicecream, gingercookie, gingericecream\}$

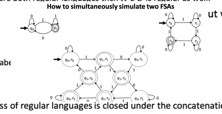
STAR $A^* = \{x_1 x_2 \ldots x_k: \ k \ge 0 \text{ and } \forall i\ x_i \in A\}$ $= \{\epsilon, oreo, ginger, oreooreo, oreoginger, gingerginger, gingeroreo, oreooreooreo, \ldots\}$

concatenate any number of strings in A

Theorem 1.25 The class of regular languages is closed under the union operation.

In other words, if A, B are both regular languages then $A \cup B$ is regular as well.

How to simultaneously simulate two FSAs



Theorem 1.26 The class of regular languages is closed under the concatenation operation.

Let A, B be regular languages recognized by FSAs M_1 and M_2.

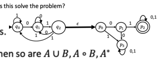
How do we construct M to recognize $A \circ B$?

Input string w should be recognized iff $w = xy$ where $x \in A$ and $y \in B$.

Insert a transition labeled ϵ from the M_1's accept state to M_2's start state.

The machine can magically guess when to take this transition (free move).

How does this solve the problem?



Regular Languages are those recognized by DFAs.

Every DFA is also an NFA

If A, B are each recognized by an NFA then so are $A \cup B, A \circ B, A^*$

every NFA has an equivalent DFA

Therefore, if A, B are each recognized by a DFA then so are $A \cup B, A \circ B, A^*$

In other words, regular languages are closed under regular operations

Regular Expressions

In Reg Expressions, * operation done first, then concatenation, then Union unless ().

Defining Regular Expressions Recursively

Given an alphabet Σ, a regular expression R over Σ is:

1. a, $a \in \Sigma$ — represents the language $\{a\}$
2. ϵ — single empty string $\{\epsilon\}$
3. ϕ — Lang doesn't contain any strings {}
4. $(R_1 \cup R_2)$, where R_1, R_2 are both regular expressions
5. $(R_1 \circ R_2)$
6. $(R_1)^*$

Rules to remember:

- $R \circ \phi = \phi$
- $R \circ \epsilon = R$
- $\phi^* = \{\epsilon\}$
- $R \cup \phi = R$
- $R \cup \epsilon = R$ iff $\epsilon \in R$

Regular Expressions are useful in defining patterns

NFAs are useful in designing automata

DFAs are a useful concept for designing pattern matching algorithms (and a whole lot more).

Sample Regular Expressions

Let $\Sigma = \{0,1\}$.

$L_1 = \{3rd\ last\ bit\ is\ 1\}$

$\Sigma^* 1 \Sigma\Sigma$

$L_2 = \{strings\ in\ which\ every\ 0\ is\ followed\ by\ at\ least\ one\ 1\}$

$1^*(011^*)^* = 1^*(01^+)^*$

$L_3 = \{strings\ starting\ with\ 0\ and\ followed\ by\ any\ number\ of\ 1s\ OR\ strings\ with\ no\ 0\}$

$01^* \cup 1^* = (0 \cup \epsilon)1^*$

$L_4 = \{strings\ with\ lengths\ that\ are\ multiples\ of\ 3\}$

$(\Sigma\Sigma\Sigma)^*$

Context-Free Grammars (CFG)

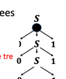
- A context-free grammar G is a 4-tuple (V, Σ, R, S), where
  1. V is a finite set of variables (non-terminal symbols),
  2. Σ is a finite set (disjoint from V) of terminal symbols,
  3. R is a finite set of rules, with the LHS being a variable and the RHS a string of variables and terminals, and
  4. $S \in V$ is the start variable
- If u, v, w are strings of variables and terminals, and $A \to w$ is a rule in R, we say that $uAv \Rightarrow uwv$ (uAv yields uwv).
- If $u \Rightarrow^* v$ we say that u derives v. ($\Rightarrow^*$ is the transitive closure of $\Rightarrow$)
- Finally, $L(G) = \{w \in \Sigma^*: S \Rightarrow^* w\}$

An Example CFG: The symbol S is a non-terminal (or variable).

Derivations as Parse Trees

$S \to 0S1 \to 00S11 \to 000111$

$S \to 0S1$, $S \to 01$



Derived string = pre-order traversal of leaves in the parse tree

CFG for Balanced Parentheses

Balanced Strings: (), ()(), (()), (()()), ...

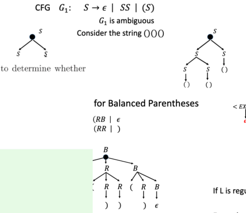
Not balanced: (, ()), ((()).

CFG G_1: $S \to \epsilon \mid SS \mid (S)$

G_1 is ambiguous

Consider the string ()()()

Why this CFG won't work



<EXPR> → <EXPR> + <EXPR>

<EXPR> → <EXPR> × <EXPR>

<EXPR> → (<EXPR>)

<EXPR> → a

a + a × a

for Balanced Parentheses

A grammar is ambiguous if any string has more than one parse tree.

Every Regular Language is Context-Free

If L is regular, there is a DFA M for it.

Example:

Create a CFG with variables Q_0, Q_1, Q_2, Q_3 with rules:

$Q_0 \to 1Q_0 \mid 0Q_1$

$Q_1 \to 1Q_0 \mid 0Q_2$

$Q_2 \to 1Q_3 \mid 0Q_2$

$Q_3 \to 1Q_3 \mid 0Q_3 \mid \epsilon$

This trick (one variable per state, start state variable is appears in LHS of first rule, one rule per transition, and each accept state variable yields the empty string) converts every DFA into an equivalent CFG.

Write this grammar in Chomsky Normal Form. Then use CYK algorithm to determine whether $ababb$ is in the language of this grammar. (S is the start symbol)

$S \to ASA \mid aB$

$A \to B \mid S$

$B \to b \mid \epsilon$

Solution:

- First, we add the new start variable S':

$S' \to S$

$S \to ASA \mid aB$

$A \to B \mid S$

$B \to b \mid \epsilon$

- Now we need to find all variables that can yield ϵ: A and B can go to ϵ, but not S. We need to eliminate all ϵ − *productions* and compensate them, by adding new productions.

$S' \to S$

$S \to ASA \mid aB \mid SA \mid AS \mid a$

$A \to B \mid S$

$B \to b$

- Next, we need to get rid of unit rules. We have these unit-rules: $S' \to S$, $A \to B$ and $A \to S$. Once again, we must compensate for the absense of these rules by adding new ones.

$S' \to ASA \mid aB \mid SA \mid AS \mid a$

$S \to ASA \mid aB \mid SA \mid AS \mid a$

$A \to b \mid ASA \mid aB \mid SA \mid AS \mid a$

$B \to b$

- Next step is to get rid of the rules of improper form. We achieve this by adding new variables.

$S' \to T_{AS}A \mid T_a B \mid SA \mid AS \mid a$

$S \to T_{AS}A \mid T_a B \mid SA \mid AS \mid a$

$A \to b \mid T_{AS}A \mid T_a B \mid SA \mid AS \mid a$

$B \to b$

$T_{AS} \to AS$

$T_a \to a$

DFA's and NFA's

Theorem 1.39. Every NFA has an equivalent DFA.

Proof Sketch: Let NFA $N = (Q, \Sigma, \delta, q_0, F)$.

Define DFA $M = (\wp(Q), \Sigma, \delta', q_0', F')$

$F' = \{R \subseteq Q: R \cap F \neq \Phi\}$ all subsets that contain a final state of N.

From state $r \in Q, a \in \Sigma$:

$\delta(r, a)$ = set of states in N directly following an a-transition from r

$\delta'(R, a)$ = set of states in N reached by following an a-transition from any state in $R \subseteq Q$

$= \cup_{r \in R} \delta(r, a)$

What about freebies: States reached by ϵ-transitions following the a-transition?

$E(R) = \{q \in N: q$ is reachable from $r \in R$ using a sequence of 0 or more ϵ-transitions$\}$

Now, $\delta'(R, a) = E(\cup_{r \in R} \delta(r, a))$

$= \cup_{r \in R} E(\delta(r, a))$

Finally, $q_0' = E(\{q_0\})$

Designing a DFA Equivalent to NFA N

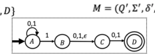
$N = (Q, \Sigma, \delta, q_0, F)$ where:

$Q = \{A, B, C, D\}$

$\Sigma = \{0,1\}$

$q_0 = A$

$F = \{D\}$



$M = (Q', \Sigma', \delta', q_0', F')$ where

$\Sigma' = \Sigma$

What states?

$\delta(A, 0) = \{A\}$	$\delta(C, 0) = \{D\}$
$\delta(A, 1) = \{A, B\}$	$\delta(C, 1) = \{D\}$
$\delta(A, \epsilon) = \Phi$	$\delta(C, \epsilon) = \Phi$
$\delta(B, 0) = \{C\}$	$\delta(D, 0) = \Phi$
$\delta(B, 1) = \{C\}$	$\delta(D, 1) = \Phi$
$\delta(B, \epsilon) = \{C\}$	$\delta(D, \epsilon) = \Phi$

Hint: Where are all the clones?

So, make one state in M for each subset of Q:

$Q' = \wp(Q)$

$q_0' = \{A\}$

$F' = \{R \in Q': R \cap F \neq \Phi\}$

Every NFA can be converted into an equivalent FSA (Their languages are the same.)

FSA vs. NFA

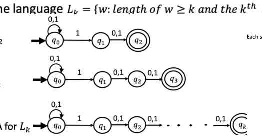
Is an exponential blowup necessary?

Consider the language $L_k = \{w: length\ of\ w \ge k\ and\ the\ k^{th}\ last\ symbol\ is\ 1\}$

3-state NFA for L_2

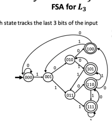
4-state NFA for L_3

$(k+1)$-state NFA for L_k



FSA for L_3

Each state tracks the last 3 bits of the input



The FSAs Grow Exponentially! This strategy uses

8 states for L_3

16 states for L_4

...

2^k states for L_k

The NFA for L_k needed only $k+1$ states while this FSA needs 2^k states!

Theorem. Every DFA for L_k has $\ge 2^k$ states.

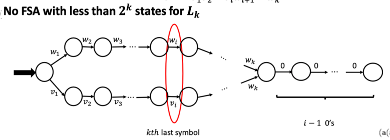
Proof: (by contradiction).

- Suppose that DFA M with fewer states recognizes L_k.
- There are 2^k strings of length k, but M has strictly less than 2^k states.
- By the pigeonhole principle, $\exists w, v$ both of length k that end up in the same state of M.
- Let i be the last (rightmost) bit that strings w, v differ in.

$w = w_1 w_2 \cdots w_i w_{i+1} \cdots w_k$

$v = v_1 v_2 \cdots v_i w_{i+1} \cdots w_k$

No FSA with less than 2^k states for L_k



$w' = w0^{i-1}$ and $v' = v0^{i-1}$ both end in the same final state. Both are either accepted or both are rejected.

But w', v' differ in the k^{th} last bit. So one is in L_k and the other is not.

This means the FSA with less than 2^k states does not recognize L_k.

Regular Expressions and FSAs

Theorem 1.54 A language is regular if and only if some regular expression describes it.

Proof in two parts:

- Given a regular expression, construct a DFA for it.
- Given a DFA, construct a regular expression for it.

Part I: From Regular Expressions to NFA

Part I: Given a regular expression R, construct a DFA (or an NFA) for it

Proof by induction:

Base Cases: $R = a\ (a \in \Sigma)$ $\quad R = \epsilon \quad R = \phi$

NFAs:

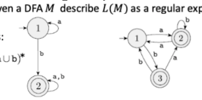
Inductive Step: $R = R_1 \cup R_2 \quad R = R_1 \circ R_2 \quad R = (R_1)^*$

NFAs — the NFA's pic's for U * concat

From DFA to Regular Expressions

Part II: Given a DFA M describe $L(M)$ as a regular expression.

Examples:



$a^*b(a \cup b)^*$

$(a(aa \cup b)^*ab \cup b)((ba \cup a)(aa \cup b)^*ab \cup bb)^*((ba \cup a)(aa \cup b)^* \cup \epsilon) \cup a(aa \cup b)^*$

Part II: Is there a systematic way to convert DFAs?

Converting DFA to REs Systematically

Part II: Given a DFA M describe $L(M)$ as a regular expression.

Proof Outline:

DFA M → GNFA G → Regular Expression $R = L(M)$

Step 1: Convert the DFA into a Generalized NFA

Step 2: Derive the regular expression from the GNFA

Step 1: Convert the DFA into a Generalized NFA

DFA M → GNFA G

Create a new start state with outgoing ϵ-transition to the start state of M

Create a new accept state with incoming ϵ-transitions from every accept state of M

If there are multiple labels on any transition, replace it with the union of the labels

Step 2: Derive the regular expression from the GNFA

Manipulate labels so that the label on (s, a) describes all strings accepted by the GNFA!

Eliminate states one-at-a-time: updating labels at each step

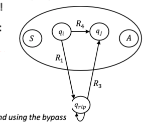
HOW EXACTLY?

To remove state q_{rip}

For every pair (q_i, q_j):

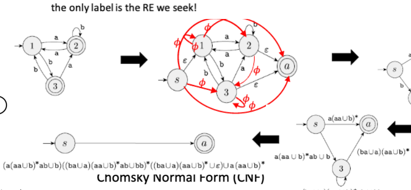
replace R_4 by $R_4 \cup R_1 R_2^* R_3$

consider all ways – old and using the bypass to go from q_i to q_j



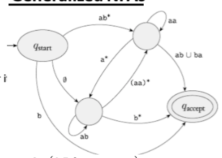
Must also separately consider (q_j, q_i) — Ignore the rest of the states and transitions

When only two states (start and accept) are left: the only label is the RE we seek!



$(a(aa \cup b)^*ab \cup b)((ba \cup a)(aa \cup b)^*ab \cup bb)^*((ba \cup a)(aa \cup b)^* \cup \epsilon) \cup a(aa \cup b)^*$

Generalized NFAs



Like an NFA but:

1. Start state:
   a. No incoming transition
   b. Outgoing transitions to every state
2. Accept state:
   a. No outgoing transition
   b. Incoming transitions from every state
   c. Distinct from start state
3. All other states:
   a. Outgoing transition to every state (except start), including itself.
4. Transitions labeled with regular expressions.

$G = (Q, \Sigma, \delta, q_{start}, q_{accept})$

$\delta: (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \to \mathcal{R}$ (the set of REs over Σ)

G accepts input string $w \in \Sigma^*$ if

$w = w_1 w_2 \ldots w_k$, $w_i \in \Sigma^*$ (the input can be divided into blocks)

and $\exists q_0, q_1, \ldots, q_k$ (and there is a path)

1. $q_0 = q_{start}$ (from the start state)
2. $q_k = q_{accept}$ (and ending in the accept state)
3. $w_i \in L(R_i)$, $R_i = \delta(q_{i-1}, q_i)$ (and each block matches the RE on the transition)

How does a GNFA computation proceed?

1. Take a transition by matching a block of input symbols to the RE label.
2. Accept input if in accept state and entire input has been processed.

The Conversion Algorithm

CONVERT(G): $(G = (Q, \Sigma, \delta, q_{start}, q_{accept})$ is the GNFA we start with)

1. Let k be the number of states of G.
2. If $k = 2$: return the label on transition (q_{start}, q_{accept})
3. If $k > 2$: select state $q_{rip} \in Q - \{q_{start}, q_{accept}\}$

   Define $G' = (Q', \Sigma, \delta', q_{start}, q_{accept})$

   where $Q' = Q - \{q_{rip}\}$

   and $\forall q_i \in Q - \{q_{accept}\}, \forall q_j \in Q - \{q_{start}\}$:

   if $R_1 = \delta(q_i, q_{rip}), R_2 = \delta(q_{rip}, q_{rip}), R_3 = \delta(q_{rip}, q_j), R_4 = \delta(q_i, q_j)$

   define $\delta'(q_i, q_j) = R_1 R_2^* R_3 \cup R_4$

   New label contains the previous label.

4. Return CONVERT(G')

G and CONVERT(G) are Equivalent

CLAIM: The RE CONVERT(G) describes $L(G)$.

Proof Sketch: By induction on the number of states.

Base Case: $k = 2$ All strings accepted state in G match the RE returned by CONVERT(G)

Inductive Hypothesis: Claim is true for all GNFAs with $k-1$ states.

Inductive Step: (from G to G'):

If G accepts w:

Case 1: q_{rip} is not on an accepting path. The same path exists in G' because the new label in G' contains the old ones in G.

Case 2: q_{rip} appears on an accepting path in G: $\ldots q_i q_{rip} q_{rip} q_{rip} q_j \ldots q_i q_{rip} q_k \ldots$

G' contains $\ldots q_i q_j \ldots q_i q_k \ldots$ which accepts w (use the bypass)

If G' accepts w:

every transition in G' is either a transition in G or a path in G. So an accepting path in G' implies an accepting path in G.

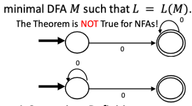
So, $L(G) = L(G')$

By the Inductive Hypothesis the RE CONVERT(G') describes $L(G')$ and therefore, $L(G)$.

DFA Minimization

For every regular language L, there exists a *unique*† minimal DFA M such that $L = L(M)$. († up to relabeling of states.)

The Theorem is NOT True for NFAs!



A Convenient Definition

Suppose we are given a DFA $M = (Q, \Sigma, \delta, q_0, F)$

In state q, if the next input symbol is a, the next state is $\delta(q, a)$.

What is the final state when we start in q and process string w?

Define: $\Delta: Q \times \Sigma^* \to Q$ recursively as follows:

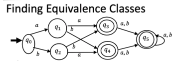
$\Delta(q, \epsilon) = q$

$\Delta(q, ax) = \Delta(\delta(q, a), x), \quad a \in \Sigma, x \in \Sigma^*$

To minimize a DFA we need to find all equivalence classes.

Easier to identify pairs of states that are distinguishable.

Finding Equivalence Classes



Step 1: Make a table of all pairs of states. Repeat until no new pairs are marked off.

Step 2: Mark off pairs of accept/reject states

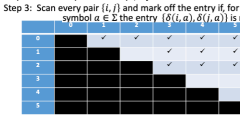
Step 3: Scan every pair $\{i, j\}$ and mark off the entry if, for some symbol $a \in \Sigma$ the entry $\{\delta(i, a), \delta(j, a)\}$ is marked.

Step 2: 0,1,2 distinguished from 3,4,5

Step 3: 0 distinguished from 1,2

No more pairs can be marked.

Unmarked pairs are equivalent!



Indistinguishable States

States p and q are indistinguishable: $p \sim q$

if, for every string w, the computation starting in state p reaches an accept state if and only if the computation from q reaches an accept state.

$p \sim q \overset{def}{=} \forall w \in \Sigma^*: (\Delta(p, w) \in F \Leftrightarrow \Delta(q, w) \in F)$

NOTE: If $p \in F, q \notin F$ then

$\Delta(p, \epsilon) = p \in F$, but $\Delta(q, \epsilon) = q \notin F$

A non-accepting state is NOT indistinguishable from an accepting state.

Indistinguishability and Equivalence

Lemma: The indistinguishability relation $\sim$ is an equivalence relation.

Proof:

- Reflexive: $p \sim p$ since $\forall w \in \Sigma^*: (\Delta(p, w) \in F \Leftrightarrow \Delta(p, w) \in F)$
- Symmetric: $p \sim q \Leftrightarrow q \sim p$

  $(\forall w \in \Sigma^*: \Delta(p, w) \in F \Leftrightarrow \Delta(q, w) \in F)$

  $\Leftrightarrow (\forall w \in \Sigma^*: \Delta(q, w) \in F \Leftrightarrow \Delta(p, w) \in F)$.

- Transitive: $(p \sim q \wedge q \sim r) \Rightarrow p \sim r$

  If $\forall w \in \Sigma^*: (\Delta(p, w) \in F \Leftrightarrow \Delta(q, w) \in F)$ and $\forall w \in \Sigma^*: (\Delta(q, w) \in F \Leftrightarrow \Delta(r, w) \in F)$

  it follows that $\forall w \in \Sigma^*: (\Delta(p, w) \in F \Leftrightarrow \Delta(r, w) \in F)$.

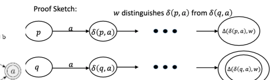
  Therefore, $p \sim q \wedge q \sim r \Rightarrow p \sim r$.

The Distinguishability Lemma

Lemma: If for some symbol $\delta(p, a) \nsim \delta(q, a)$ then $p \nsim q$

$\equiv$ If $p \sim q$ then $\delta(p, a) \sim \delta(q, a)$ for every symbol

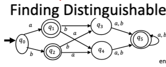
Proof Sketch: w distinguishes $\delta(p, a)$ from $\delta(q, a)$



aw distinguishes p from q

Recall: An equivalence relation over a set partitions the set into equivalence classes

Finding Distinguishable Pairs of States



Initialize: Make a table of all pairs of states.

Step 1: Mark off pairs of accept/reject states.

Repeat until no new pairs are marked off.

Step 2: Scan every pair $\{i, j\}$ and mark off the entry if, for some symbol $a \in \Sigma$ the entry $\{\delta(i, a), \delta(j, a)\}$ is marked.

Initialize: Create an empty set D which will store pairs of distinguishable states.

Step 0: Add all pairs (p, q) where exactly one is an accept state to D.

While True do:

Scan over all pairs (p, q) of states not yet distinguished

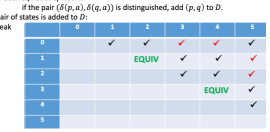
for each alphabet symbol a

if the pair $(\delta(p, a), \delta(q, a))$ is distinguished, add (p, q) to D.

If no new pair of states is added to D:

break

return D



The Minimal DFA

We started with DFA $M = (Q, \Sigma, \delta, q_0, F)$

So far, we have identified the equivalence classes.

Notation: If $p \in Q$ let $[p] \subseteq Q$ denote the equivalence class of state p

Next, we construct the reduced DFA $M_R = (Q_R, \Sigma, \delta_R, q_{0R}, F_R)$, where

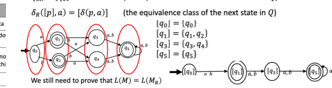
$Q_R = \{[p]: p \in Q\}$	(one state for each equivalence class)
$F_R = \{[p]: p \in F\}$	(equivalence classes containing accept states of Q)
$q_{0R} = [q_0]$	(the equivalence class of the start state of Q)
$\delta_R([p], a) = [\delta(p, a)]$	(the equivalence class of the next state in Q)

$[q_0] = \{q_0\}$

$[q_1] = \{q_1, q_2\}$

$[q_3] = \{q_3, q_4\}$

$[q_5] = \{q_5\}$



We still need to prove that $L(M) = L(M_R)$

Proving Correctness

Need to prove the following:

SOUNDNESS: Is every pair in D distinguishable?

If $(p, q) \in D$ then $p \nsim q$

COMPLETENESS: Does the algorithm find all distinguishable pairs of states?

If $p \nsim q$ then $(p, q) \in D$

Proving Soundness

SOUNDNESS: If $(p, q) \in D$ then $p \nsim q$

Proof Sketch: By induction on the iteration number k when the pair is added to D.

Base Case: $k = 0$ The pair is added in Step 0. In this case one state is an accept state and the other is not. Therefore, $p \nsim q$.

Inductive Hypothesis: If (p, q) is added to D in iteration k or earlier then $p \nsim q$.

Inductive Step: (p, q) is added to D in iteration $k+1$

Then, for some symbol a, it must be the case that the pair $(\delta(p, a), \delta(q, a)) \in D$ at the end of iteration k.

But this implies that $\delta(p, a) \nsim \delta(q, a)$.

By the distinguishability lemma it follows that $p \nsim q$.

Proving Completeness

COMPLETENESS: If $p \nsim q$ then $(p, q) \in D$

Proof Sketch: By induction on the length k of the shortest string that distinguishes (p, q).

Base Case: $k = 0$ If the empty string distinguishes the two states, one is an accept state and the other is not. Therefore (p, q) is added to D in the initial Step 0.

Inductive Hypothesis: If the shortest distinguishing string k then (p, q) is added to D in the k^{th} iteration.

Inductive Step: The shortest distinguishing string (sds) is length $k+1$

Then, for some symbol a, it must be the case that the sds for $(\delta(p, a), \delta(q, a))$ is length k; otherwise the sds for (p, q) would be less than $k+1$.

Now, by the inductive hypothesis, for $(\delta(p, a), \delta(q, a))$ is added to D in the k^{th} iteration.

But then (p, q) will be added to D in the $(k+1)^{st}$ iteration.

Chomsky Normal Form (CNF)

CFG is in Chomsky Normal Form if every rule is of the form

- $A \to BC$ where B, C are variables in $V - \{S\}$, or
- $A \to a$ where a is a terminal in Σ, or
- $S \to \epsilon$ where S is the start variable in V.

$S_0 \to AA_1 \mid UB \mid a \mid SA \mid AS$

$S \to AA_1 \mid UB \mid a \mid SA \mid AS$

$A \to b \mid AA_1 \mid UB \mid a \mid SA \mid AS$

$A_1 \to SA$

$U \to a$

$B \to b$

Chomsky's Language Hierarchy

GRAMMAR	LANGUAGE	AUTOMATA
Type 3	Regular	Finite State Automata
Type 2	Context-Free	Non-deterministic Pushdown Automata
Type 1	Context-Sensitive	Linear-Space bounded non deterministic Turing Machine
Type 0	Recursively Enumerable	Turing Machines

Theorem 2.10: Every CFL is generated by a CFG in CNF.

Idea: Start with any CFG and remove violations to the CNF rules

1. Create a new start variable
2. Eliminate rules of the form $A \to \epsilon$.
3. Eliminate unit rules of the form $A \to B$.
4. Convert rules with long RHS by adding new variables.

Converting a CFG into Equivalent CNF:

$X \to YXY \mid Y \mid \epsilon$

$Y \to 00 \mid \epsilon$

1. Create a new start variable: $S \to X$; $X \to YXY \mid Y \mid \epsilon$; $Y \to 00 \mid \epsilon$
2. Eliminate rules of the form $A \to \epsilon$. For each occurrence of A on the RHS of any rule, add another rule without that occurrence of A.

   Eliminate $Y \to \epsilon$: $S \to X$; $X \to YXY \mid YX \mid XY \mid X \mid Y \mid \epsilon$; $Y \to 00$

   Eliminate $X \to \epsilon$: $S \to X \mid \epsilon$; $X \to YXY \mid YX \mid XY \mid X \mid Y \mid YY$; $Y \to 00$

3. Eliminate unit rules of the form $A \to B$.

   Eliminate $X \to Y$: $S \to X \mid \epsilon$; $X \to YXY \mid YY \mid 00 \mid XY \mid YX$; $Y \to 00$

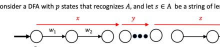
   Eliminate $S \to X$: $S \to YXY \mid YY \mid 00 \mid XY \mid YX \mid \epsilon$; $X \to YXY \mid YY \mid 00 \mid XY \mid YX$; $Y \to 00$

4. Convert rules with long RHS by adding new variables.

   $S \to YZ_1 \mid YY \mid Z_2 Z_3 \mid XY \mid YX \mid \epsilon$; $Z_1 \to XY$; $Z_2 \to 0$; $Z_3 \to 0$; $X \to YZ_1 \mid YY \mid Z_2 Z_3 \mid XY \mid YX$; $Y \to Z_2 Z_3$

Proof of The Pumping Lemma

Consider a DFA with p states that recognizes A, and let $s \in A$ be a string of length $l \ge p$.



The number of states on the accepting path is $l+1 \ge p+1$.

By the pigeonhole principle, some state must repeat within the first p transitions.

So, we can express $s = xyz$ such that $\Delta(q_0, x) = q$, $\Delta(q, y) = q$, $\Delta(q, z) = q_f$

and $|y| > 0$ and $|xy| \le p$.

Since the loop can be traversed any number of times, every string $\{xy^iz: i \ge 0\}$ will be accepted.

The Pumping Lemma for Context-Free Languages

$\forall$ context − free language A

$\exists p$ (the pumping length)

$\forall s \in A, |s| \ge p \Rightarrow \exists u, v, x, y, z: s = uvxyz$ where

1. $\forall i \ge 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \le p$

Proof idea: Choose p so that:

The parse tree for s ($|s| \ge p$) repeats a non-terminal R along a root-to-leaf path

Then (as we saw earlier) $S \Rightarrow^* s = uRz$, $R \Rightarrow^* vRy$

so that condition 1 is satisfied.

The question: how to choose p and what about conditions 2 and 3?

The Pumping Lemma for Regular Languages

$\forall$ regular language A

$\exists p$ (the pumping length)

$\forall s \in A, |s| \ge p \Rightarrow \exists x, y, z: s = xyz$ where

1. $\forall i \ge 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \le p$

Proof idea: Choose p to be the number of states of a DFA for A.

The accepting path for s repeats a state in the first p steps

So $s = xyz$, where y is the substring that goes around the loop

Traverse the loop any number of times after which z will reach the accept state.

Pumping Lemma: If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

2. $\{w: w$ contains at least two a's and at most one $b\}$

$aa^+ \cup aa^+ba^+ \cup aba^+ \cup baa^+$

Proving a Language Non-Regular

If A is regular then, according to the pumping lemma:

$\exists p$ (the pumping length)

$\forall s \in A, |s| \ge p \Rightarrow \exists x, y, z: s = xyz$ where

1. $\forall i \ge 0$, $xy^iz \in A$,
2. $|y| > 0$ and
3. $|xy| \le p$

Step 1. Carefully craft a string s, $|s| \ge p$

Step 2. Argue that if $s = xyz$ such that $|xy| \le p \wedge |y| > 0$ then $\exists i$: $xy^iz \notin A$.

Claim: The language $L = \{0^n1^n: n \ge 0\}$ is not regular.

Proof: We assume that it is regular.

Therefore it is recognized by a DFA M with p states, for some number p.

Consider the string $0^p1^p \in L$.

We can express $0^p1^p = xyz$, $|y| > 0$ $|xy| \le p$.

Note that y is a string of 0s.

The strings $xz, xyyz, xyyyz, \ldots$ will all be accepted by M.

But none of these is in L

So $L(M) \neq L$

Therefore, L is not regular!

Since the pumping lemma does not apply, A cannot be regular.

Proving a Language non-CFL

If A is a CFL then, according to the pumping lemma:

$\exists p$ (the pumping length)

$\forall s \in A, |s| \ge p \Rightarrow \exists u, v, x, y, z: s = uvxyz$ where

1. $\forall i \ge 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \le p$

Step 1. Carefully craft a string s, $|s| \ge p$

Step 2. Argue that if $s = uvxyz$ such that $|vxy| \le p \wedge |vy| > 0$ then $\exists i$: $uv^ixy^iz \notin A$.

Since the pumping lemma does not apply, A cannot be CFL.

Tricky part: we don't know where the substring vxy occurs!

Example 1: $L_1 = \{a^i b^i c^i\}$

Example 2: $L_2 = \{a^i b^i a^i b^i : i \ge 0\}$

Consider the string $a^pb^pa^pb^p$

Divide this into 5 pieces $uvxyz$ such that $|vxy| \le p$

Where does the segment vxy lie within the string?

The window always contains a's and b's from at most one block each.

Pumping the string makes the two blocks of a's or b's to become unequal in size.

Since condition 1 of the pumping lemma cannot be met, L_2 is not a CFL.

$\{w: w$ contains even numbers of a's and b's$\}$

$((aa \cup bb)(ab \cup ba)(aa \cup bb)^*(ab \cup ba) \cup (aa \cup bb))^*$

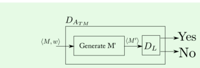
Problem 3: Reduction (10 points)

Prove that L is undecidable where:

$L = \{\langle M \rangle \mid M$ is a TM and accepts some string of odd length$\}$

You are not allowed to use Rice's Theorem in this problem (although you can adapt the proof of Rice's Theorem to this problem).

Solution:



We reduce A_{TM}, which we know is undecidable, to L. That is we give the code for a decider of A_{TM}, $D_{A_{TM}}$ assuming that we have access to a subroutine (a TM) that decides L.

Now let's investigate the language of the following TM M'.

Algorithm $M'(x)$

1. $r \leftarrow$Simulate $M(w)$
2. If $r =$"yes"
3. then return "yes"
4. else return "no"

If $\langle M, w \rangle \in A_{TM}$, then the simulation in line 1 always returns "yes" and therefore always line 3 executes which means $L(M') = \Sigma^*$ and therefore $\langle M' \rangle \in L$. If $\langle M, w \rangle \notin A_{TM}$, then either $M(w)$ in line 1 returns "no" in that case $M'(x)$ returns "no" in line 4, or $M(w)$ never halts, in that case $M'(x)$ never halts; so in both these two cases $L(M') = \emptyset$ and hence $\langle M' \rangle \notin L$.

So the idea for implementing $D_{A_{TM}}$ is to construct $\langle M' \rangle$ from M and w and pass it to D_L to answer the question $\langle M' \rangle \overset{?}{\in} L$ which as mentioned above will determine the answer to question $\langle M, w \rangle \overset{?}{\in} A_{TM}$. In pseudocode:

Algorithm $D_{A_{TM}}(\langle M, w \rangle)$

1. Write down $\langle M' \rangle$ using $\langle M, w \rangle$
2. return $D_L(\langle M' \rangle)$

$b \cup ((ab \cup ba)(aa \cup bb)^*a)$

Show that the language $L = \{<M, w, k> : TM\ M$ accepts input w and never moves its head beyond the first k tape cells$\}$ is decidable.

Let us suppose: Turing Machine(TM) M is not allowed to go beyond the first K-tape cells. The TM have a fixed tape size. In the machine the number of configuration is finite way in different ways such as $(k.ns.sa)^k$.

Here, k is the tape size, ns is the number of states, and sa is the size of the alphabet. Therefore, apply the computation on the $(k.ns.sa)^k$, the we find out of any computation on this device the machine halt else loops.

The reason behind for this computation is that after $(k.ns.sa)^k$ steps, at least one configuration would be repeated, and Turing machine M would be configuration repeatedly and that will never be halt.

Hence, Language L is decidable by the following decider is as shown below:

Decidable = "On input <M,w,k>": Turing machine M , w is a input string, and k is the integer.

Case 1: Simulate TM M on the input w during o more than $(k.ns.sa)^k$ steps. IF TM M heads ever goes beyond the k-th position, it is rejects. Otherwise accepts.

Case 2: If the configuration is ended without any simulation. Machine M didn't halt, reject.

If there is accepting computation path on k cells, then we found a successful split and the string is in w. If all the computation path rejects, then the string is not $(k.ns.sa)^k$. In other case it is easy to see machine M is halts.

Hence, language L is decidable on the first k tape cells.

To show that this is decidable, construct a TM S that will decide it.

S: on input <M,w,k>
1. Compute the number of states of M, denoted n
2. Simulate M on input W for at most length (W)+n+(k-1) steps.
   If M goes head beyond the k, does not accept
3. If M go head beyond the k tape cells, reject.

For example if we take k = 10, when TM M visits the 10th cell of its tape while processing input string '01' , this language is decidable.
An idea for a decider:
Given <M, 01>, simulate M on 01, i.e., simulate two steps of the machine. While simulating M on the universal Turing machine we keep track of the cell position currently being visited by the head of machine M. We essentially keep a counter of the times we move to the right and decrease the number if we move to the left, until we reach zero in which case we know we are on the first position. If we ever find the counter to indicate that we are in cell 10, then accept, else we reject.

S decides L because, it is bounded by a finite computation length. Note that the above length is sufficient, for after passing through length(w) states and moving, in the worst case beyond k at each time, then all further computations will spent reading blank symbols. Now because, the input symbol will be constant here after during the simulation while reading beyond k. we consider what will happen in the following (n+(k-1)) states. The (n+(k-1)) states incur the longest possible transistion sequence the machine traverse, and still go to unique states. Thus if the machine does not move beyond the first k during the transistion, it will then loop through the states on the same input, and will loop forever moving beyond k.

Let A be a TM-recognizable language consisting of TM descriptions of deciders. Prove that some decidable language D is not decided by any machine that appears in A.

Answer: Since A is Turing-recognizable, there exists an enumerator E that enumerates it. In particular, we let $\langle M_i \rangle$ be the i^{th} output of E (note: $\langle M_i \rangle$ may not be distinct).

Let $s_1, s_2, s_3 \ldots$ be the list of all possible strings in $\{0,1\}^*$. Now, we define a TM D as follows:

$D =$ "On input w:
1. If $w \notin \{0,1\}^*$, **reject**.
2. Else, w is equal to s_i for a specific i.
3. Use E to enumerate $\langle M_1\rangle, \langle M_2\rangle, \ldots$ until $\langle M_i \rangle$.
4. Run M_i on input w.
5. If M_i accepts, **reject**. Otherwise, **accept**."

Clearly, D is a decider (why??). However, D is different from any M_i (why??), so that $\langle D\rangle$ is not in A.

(a) Prove that there exists a Turing machine M whose language L is decidable, but M is not a decider. This shows that just because a Turing machine's language is decidable, it's not necessarily the case that the Turing machine itself must be a decider.
   Let M be the Turing machine that loops indefinitely on all inputs, which has language $L = \emptyset$. L is decidable (the Turing machine that rejects all inputs is a decider for L), but M is not a decider.

(b) Prove that for every language L, there is a decider M that accepts every string in L and another decider M' that rejects every string not in L. Explain why this doesn't prove that every language is decidable.
   We can set M as the TM that accepts all inputs and M' as the TM that rejects all inputs; then M accepts all strings in L and M' rejects all strings in L, and both are deciders. This doesn't show that L is decidable since the definition of decidability requires the same TM to accept all strings in L and reject all strings not in L, not two different TMs.

(c) Find a pair of languages A and B such that A is a subset of B, B is decidable, but A is not decidable. This shows that a subset of a decidable language is not necessarily decidable, i.e. bigger languages are not necessarily 'harder.'
   Let A be any undecidable language (e.g. $HALT$), and B be Σ^*. Then B is decidable, and A is a subset of B.

a. For any two sets A and B, if A is uncountably infinite and B is countably infinite, then $A \cap B$ is countably infinite .... T F
Why/Why not?

FALSE. $A \cap B$ can be empty which is finite. E.g., A=[-2,-1] (set of all real numbers between -2 and -1) and B= N, the set of natural numbers.

b. If R is any regular language and L is any context free language, then L∘R is context-free .... T F
Why/Why not?

TRUE. Every regular language is a CFL and CFLs are closed under concatenation as you showed in your homework #5.

c. The language $\{a^m b^n c^n d^m \mid m, n \geq 0\}$ over $\Sigma = \{a,b,c,d\}$ is not context free... T F
Why/Why not?

FALSE. Here's a CFG for the language:
S → aSd | B
B → bBc | ε

d. For any two languages A and B, if $A \subseteq B$, then A is reducible to B........ T F
Why/Why not?

FALSE. A hard problem (language) can be a subset of an easy problem and not be reducible to the easy problem. E.g., let $A = A_{TM}$ and $B = \Sigma^*$. Then, $A \subseteq B$ but A is not reducible to B (because Σ^* is decidable and A_{TM} is not).

e. If language A is reducible to language B and B is undecidable, then A must be undecidable........ T F
Why/Why not?

FALSE. The reduction only proves $A \leq B$, and B's undecidability does not tell us anything about A's hardness. E.g., $A = \emptyset$ (the empty language) is decidable (via a TM that rejects all strings). A can be reduced to $B = A_{TM}$ simply by not using B's decider at all in the reduction and just rejecting all strings.

b. Show that the class of decidable languages is closed under complementation.

Let L be a decidable language and let M be the decider for L. Then, the following TM M' decides the complement of L:
'On input w:
1. **Simulate M on w.**
2. **If M accepts w, then REJECT; otherwise, ACCEPT."**
(M is a decider, so M always halts, either in accept state or reject state)
M' accepts w iff M rejects w i.e. M' accepts complement of L(M) = L.

6. Let $L = \{a^m b^n c^p \mid 0 < m < n < p\}$. Prove that L is not a context-free language.
Pick $s = a^p b^{p+1} c^{p+2}$. In the case analysis, using $|vxy| \leq p$, conclude that at least one symbol from $\{a,b,c\}$ is missing. Pump up or down according to different cases.

a. (2) If A is a language in **NP**, A is Turing-decidable.
Answer: True

b. (2) If D is a regular language, $D \in$ **P**.
Answer: True (we can always simulate a DFA in $O(n)$ steps)

c. (2) The language
LISKOV = $\{w | w$ is a string Barbara Liskov said during her talk at UVa $\}$
is context-free.
Answer: True (the language is finite, so it must be context-free and regular also)

d. (2) There exists some deterministic TM that can decide 3*SAT* in polynomial time.
Answer: Unknown. This depends on whether **P** = **NP**, which is unknown.

e. (2) The language $ADD = \{1^x + 1^y = 1^z | x + y = z\}$ is in **NP-Hard**.
Answer: Unknown. If **P** = **NP** then ADD is **NP-Hard**; otherwise ADD is not **NP-Hard** since we know it is in **P**.

a. (5) Is the language $\{a^n b a^n | n \geq 0\}$ regular? (A full credit answer *must include a clear and convincing proof* supporting your answer.)
Answer: We prove by contradiction using the pumping lemma.
Assume the language $L = \{a^n b a^n | n \geq 0\}$ is regular. Let p be its pumping length. Choose $s = a^p b a^p$. Since $|s| = 2p + 1 \geq p$, by the pumping lemma, there exists strings x, y, z in L such that $s = xyz$, $|xy| \leq p$, $|y| > 0$, and for all $i \geq 0$ xy^iz is in L. Since $|xy| \leq p$ then both x and y contain only a's. Since $|y| > 0$ then xy^2z has more a's to the left of b than to its right. Thus xy^2z is not in the language. We reached a contradiction. Thus, the assumption that the language is regular is false and we conclude that L is not regular.

b. (5) Is the language $\{a^n b a^n | n \geq 0\}$ context-free? (A full credit answer *must include a clear and convincing proof* supporting your answer.)
Answer: The language $\{a^n b a^n | n \geq 0\}$ is context-free. To prove a language is context-free, we can show a NDPDA or CFG that generates the language. The easiest way is a CFG: $S \rightarrow aSa|b$ generates the language.

a. (3.8 / 5) A *Right-Only Turing Machine* which is identical to a standard Turing Machine except the tape head can only move right. Describe precisely the class of languages that can be recognized by a *Right-Only Turing Machine.*
Answer: The regular languages.

b. (3.6 / 5) A *Right-Only Cyclic Turing Machine* is like a *Right-Only Turing Machine* except instead of having a one-way infinite tape, the tape is arranged in a cycle. So, when the TM moves right from the last input square, it ends up on the leftmost input square. Prove that a *Right-Only Cyclic Turing Machine* can recognize some languages that are not context free but is *less* powerful than a standard Turing Machine.
Answer: (1) A ROC-TM can recognize $a^n b^n c^n$ which is not context free. It can do this by crossing off one a, one b, and one c each time it cycles, and then check all symbols are marked off. One subltety is that the ROC-TM can't tell when it gets to the end of the input, so might not know the difference between abc (in the language) and $abcabc$ (not in the language). There is an easy solution, though — the ROC-TM knows it starts at the beginning of the input, so add a mark to that square. Then, each time that mark is reached, it knows it has returned to the start.
(2) A ROC-TM cannot recognize $HALTS_{TM}$, which can be recognized by a regular TM. The reason is simulating a TM can require much more tape than the size of its description. For example, consider simulating the TM that is the best currently known BB(6,2) candidate. Its description is short (only six states and the transition function), but its execution writes 3.18710^{10566} symbols on the tape!

(bonus +5, average +0.8) Is a Right-Only Cyclic Turing Machine *more powerful* than a NDPDA? (A convincing argument is required for credit.)
Answer: From part b, we know that a ROC-TM can recognize some language that cannot be recognized by an NDPDA. To prove it is more powerful, we also need to show it can recognize every language that can be recognized by an NDPDA. This is non-obvious, since the NDPDA has a stack of infinite depth, thus seems to have more memory than a ROC-TM. The ROC-TM is equivalent to a Linear Bounded Automaton. The restriction of only moving right doesn't really matter, since we can simulate moving left by cycling around to reach the square to the left. A LBA can recognized all context-free languages (in fact, the class of languages recognized by a LBA is the context-sensitive languages, which includes the context-free language).

A *k*-PDA is a pushdown automaton with *k* stacks. Show that 2-PDAs are more powerful than conventional PDAs with only 1 stack. (Hint: Give a language that you can show is recognizable by a 2-PDA but not by 1-PDAs. An implementation level description is sufficient – no need to formally define the 2-PDA).

The language $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context free as proved in class. Therefore, it cannot be recognized by a 1-PDA. L can however be recognized by the following 2-PDA: Push all initial a's on stack #1. Push the following b's on stack #2. Reject if there is an "a" following any "b". When a "c" is read, pop one a and one b for each c. Reject if an a or b is read. If the input ends and both stacks are empty, accept, otherwise reject.

Alternate solution:
The language $L = \{0^n 1^n 0^n \mid n \geq 0\}$ is not context free as proved in class (see lecture slides). Therefore, it cannot be recognized by a 1-PDA. L is however decidable as shown in class (see lecture slides), i.e., L is accepted by a TM. Problem 5 proves that a 2-PDA can simulate an arbitary TM. Therefore, L can be recognized by a 2-PDA.

A language is prefix-closed if the prefix of any string in the language is also in the language. Show that every *infinite prefix-closed context free language* contains an *infinite regular* subset. (Hint: Go over the proof of the pumping lemma for context free languages and see what it implies if the language is also prefix-closed).

Let L be an infinite prefix-closed context free language. Since L is a CFL, the pumping lemma holds. Let p be the pumping length and let s be a string in L longer than p. Then s can be split into uvxyz such that $uv^ixy^iz \in L$ for all $i \geq 0$ and $|vy| \geq 1$. Since L is prefix-closed, all prefixes of s are also in L, therefore, $uv^i \in L$ for all $i \geq 0$. Thus, the regular language $uv^* \subseteq L$. If $v \neq \varepsilon$, uv^* is an infinite regular subset of L, which proves the statement. If $v = \varepsilon$, then $y \neq \varepsilon$ (since we know $|vy| \geq 1$). In this case, uxy^* is an infinite regular subset of L.

a. Let L_1 and L_2 be any two decidable languages, decided by TMs M_1 and M_2 respectively. Construct a decider TM M for the language $L_1 - L_2$. Give a high level description, starting with M = "On input w: …".

On input w, our decider M does the following:
1. **Run M_1 on input w. The computation is guaranteed to halt, since M_1 is a decider for the language L_1.**
2. **Run M_2 on input w. Again, the computation is guaranteed to halt.**
3. **If M_1 accepted, and M_2 rejected, then accept the string w, else reject.**

b. Let L_1 and L_2 be any two Turing-recognizable languages, recognized by TMs M_1 and M_2 respectively. Prove or Disprove: $L_1 - L_2$ is Turing-recognizable. Either sketch a proof or give a specific counterexample.

L_1-L_2 is not necessarily Turing-recognizable.

Counterexample: Consider $L_1 = \Sigma^*$, and $L_2 = A_{TM}$. We know that L_2 is Turing-recognizable but not decidable. Now L_1-L_2 is the *complement of* the language L_2. If we have a recognizer for a language and its complement, then we have a decider for the language. This is a contradiction, since A_{TM} is undecidable. Hence we cannot always build a recognizer for the language L_1-L_2.

After proposing a toast to "Touring machines," an already toasted CS major from [name-deleted] university claims that the problem of figuring out whether a given TM accepts a finite number of strings is simple enough to be decidable. Having taken CSE 322, you know better, so you define the language $FINITE_{TM} = \{ \langle M\rangle \mid M$ is a TM and $L(M)$ is finite$\}$. Show that $FINITE_{TM}$ is undecidable by giving a reduction from a known undecidable language to $FINITE_{TM}$. For your reduction, you may use any of the languages shown to be undecidable in Section 5.1 in the textbook (up to Theorem 5.4 and its proof). (Hint: Use a reduction roughly along the lines of the one used in Theorem 5.2 and discussed in class.)

We use a construction similar to Theorem 5.2, and reduce A_{TM} to the problem $FINITE_{TM}$. We need to show that given a decider for $FINITE_{TM}$, we can use it to solve A_{TM}, making A_{TM} decidable (a contradiction). Therefore, $FINITE_{TM}$ must be undecidable. We can build the decider for A_{TM} as follows:
On input $< M, w >$,

(a) Construct a new machine M' to do the following: on any input x, it ignores the input and instead simulates M on w on a separate tape. If M accepts w then M' accepts x, otherwise M' rejects or loops on x.
(b) Feed the input $< M' >$ to the decider for $FINITE_{TM}$. If it rejects $< M' >$, then we accept the input $< M, w >$, else we reject.

Here is what is happening: If it is the case that M accepts w, then our constructed machine M' will accept any input x given to it. As a result, its language is Σ^*, an infinite language. Otherwise its language is ϕ (finite). Our decider for $FINITE_{TM}$ will answer the question of whether M''s language is finite or infinite, and we can therefore use this answer to decide whether M accepts or rejects w. Thus a decider for $FINITE_{TM}$ yields a decider for A_{TM} which is a contradiction.

Use the pumping lemma to prove that $\{www: w \in \{a, b\}^*\}$ is not regular.

(b) $L = \{www \mid w \in \{a,b\}^*\}$
(1) Assume given language L is Regular
(2) W = ababab
ab a bab
for i = 2
uv^iw = ab(a)²bab
= abaabab ∉ L
∴ it is not Regular

a) Given:- $L = \{www \mid w \in (0,1)^*\}$
(i) Assume given language 'L' is regular and there exists a finite automata with "2k" States.
(ii) Select a valid string from 'L'. Let z = 010101.
(iii) Divide the string 'z' into 3 parts u,v,w such that $|v| \geq 1$ and $|uv| \leq 2k$
z = 010101
010 (u) 1 (v) 01 (w)
(iv) For any value of i, if $uv^iw \in L$ then L is regular otherwise L is not regular.
say i=2, z = uv^iw become 0101101 ∉ L
010 1²01 ∉ L
This is contradiction to our statement made.
∴ $L = \{www \mid w \in (0,1)^*\}$ is not regular.
b) Given:- $L = \{1^n 0^m 1^n \mid m,n \geq 0\}$

Show that $\{x\#y: x, y \in \Sigma^*$ and $x \neq y\}$ is context free.

{x#y / x != y} is context-free
let Σ = {0,1}
CFG:
S → 0S0 | 1S1 | 0A1 | 1A0
A → S | #
Since, we built CFG, the given language is context-free.

b). {xy | |x| = |y| but x != y}
let Σ = {0,1}
S → 0S0 | 1S1 | 0A1 | 1A0
A → S | ε

Show that the language $L1 = \{0^n\#0^{2n}\#0^{3n} : n \geq 0\}$ is not context free.

Solution Assume L_1 is context-free. Let p be its pumping length. and $s = 0^p\#0^{2p}\#0^{3p}$. By the pumping lemma for context-free languages, we have $s = uvxyz$ satisfying the following.

- $uv^ixy^iz \in L_1$ for $i = 0, 1, 2, \ldots$, and
- $|vy| > 0$, and
- $|vxy| \leq p$.

We do a case analysis here.

- Either v or y contains #. Then $uvvxyyz$ contains more than two #'s, which contradicts $uvvxyyz \in L_1$.
- $v = 0^{k_1}$ and $y = 0^{k_2}$ for some $k_1, k_2 \in \mathcal{N}$. We have $k_1 + k_2 > 0$ since $|vy| > 0$.
  - vxy occurs before the second # in s. Then there are $3p + k_1 + k_2$ 0's in $uvvxyyz$ before the second # and only $3p$ $0's$ after, implying $uvvxyyz \notin L_1$.
  - vxy occurs after the first # in s. Then there are $5p + k_1 + k_2$ 0's in $uvvxyyz$ after the first # and only p $0's$ before, implying $uvvxyyz \notin L_1$.

Therefore, L_1 is not context-free.

Here is the pumping lemma for regular languages:

If A is a regular language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into 3 pieces, $s = xyz$, satisfying the following conditions:
1. For each $i \geq 0$, $xy^iz \in$ A,
2. $|y| > 0$, and
3. $|xy| \leq p$

Problem 7. [40 points]
Consider the following two languages over the alphabet $\Sigma = \{0, 1\}$.

$$L_U = \{<M, w> \mid \text{TM } M \text{ accepts input } w\}$$

$$L_{15} = \{<\widehat{M}> \mid |L(\widehat{M})| = 15\}$$

The language L_U is the universal language, while the language L_{15} corresponds to the following decision problem: given a Turing machine $\widehat{M}$, does its language have size exactly 15?
Show that L_{15} is undecidable via a reduction from L_U, assuming that L_U is undecidable.
(**Hint:** The number of strings of length at most 3 is exactly 15.)
Solution: We propose the following reduction from L_U to L_{15}.

Given a Turing machine M and input w, construct a Turing machine $\widehat{M}$ which behaves as follows on being given input $\widehat{w}$.

1. $\widehat{M}$ simulates the behavior of M on input w;
2. if M halts on w and accepts, then $\widehat{M}$ examines its own input $\widehat{w}$, halting in a final state if $|\widehat{w}| \leq 3$ and halting in a non-final state otherwise;
3. if M halts on w and rejects, then $\widehat{M}$ rejects its own input $\widehat{w}$.

This reduction can be computed by a halting Turing machine.
Observe that in the case where $w \in L(M)$, the language $L(\widehat{M})$ is exactly the set of strings over $\Sigma = \{0,1\}$ of length at most 3, which implies that $|L(\widehat{M})| = 15$. In the case where $w \notin L(M)$, we have that $L(\widehat{M})$ is the empty language and hence $|L(\widehat{M})| = 15$. Thus, we have established that: $<M, w> \in L_U$ if and only if $<\widehat{M}> \in L_{15}$.
Thus, since L_U reduces to L_{15}, we obtain that L_{15} is undecidable.

Problem 3. [30 points]
Show that the following language is not context-free using the Pumping Lemma.

$$L = \{(0^k 1^k)^k \mid k \geq 1\}.$$

Solution: Assume that L is context-free and apply the pumping lemma. Let n be the pumping lemma constant and define $m \geq \max\{2, n\}$ (basically we need m to be greater than 1 for the following proof). Choose $z = (0^m 1^m)^m$. Let the pumping lemma decompose z into $uvwxy$ with $|vwx| \leq n$ and $|vx| > 0$. Consider the pumped string $z' = uv^0wx^0y = uwy$
Suppose that vwx contains only one symbol, say 0's without loss of generality (a similar argument applies for the case of 1's). It is clear that vwx must belong to a single block of 0's. Then, the string z' will have a block of 0's of length less than n, while all other blocks of 0's and 1's will have length exactly n, giving us a string that does not belong to L — a contradiction.
Otherwise, vwx can straddle at most two consecutive blocks since $|vwx| \leq n \leq m$, say the first block is of 1's and the second block is of 0's (a similar argument applies to the case where the 1's and 0's are switched). Now, in the string z' we would have reduced the size of at least one of these two blocks to below m, while all other blocks of 0's or 1's will stay at length m. Again, this gives us a string that does not belong to L – a contradiction.

A Both Regular and Context-Free
B Context-Free, but not regular,
C Neither context-free nor regular but membership in the language can be decided by an algorithm.
D Undecidable

(a) $\{a^n b^m \mid m = 2n + 1\}$........
(b) $\{a^n a^m \mid m = 2n + 1\}$........
(c) $\{a^n b^m \mid m \equiv n \pmod 3\}$........
(d) $\{xcx^R \in \{a, b\}^* \mid x$ has an even number of a's$\}$.
(e) $\{a^i b^j c^k \mid k = i + j\}$........
(f) $\{a^i b^j a^k \mid k \neq i$ or $k \neq j\}$........
(g) $\{a^i b^j a^k \mid k \neq i$ and $k \neq j\}$........
(h) $\{a^n b^n a^m b^m \mid m, n \geq 0\}$........
(i) $\{a^n b^m a^n b^m \mid m, n \geq 0\}$........
(j) $\{xy \in \{a, b\}^* \mid x \neq y\}$........

(a) **B**. Here is a CFG for the language $S \rightarrow aSbb \mid b$
To show it is not regular, use pumping lemma with $s = a^p b^{2p+1}$.
(b) **A**. The language is the same as $\{a^m \mid m \bmod 3 \equiv 1\}$.
(c) **A**. One can express this language as $L_0^a \circ L_0^b \cup L_1^a \circ L_1^b \cup L_2^a \circ L_2^b$, where $L_i^a = \{a^n \mid n \equiv i \bmod 3\}$, $L_i^b = \{b^n \mid n \equiv i \bmod 3\}$ $(i = 0, 1, 2)$.
(d) **B**. The language is the intersection of a CFL language $(\{xcx^R \mid x \in \{a,b\}^*\})$ and a regular language $((b^*ab^*ab^*)^*c(b^*ab^*ab^*)^*$. To prove this language is not regular use the pumping lemma with $s = a^{2p}ca^{2p}$.
(e) **B**. The PDA for this language works as follows. Push in all the a's and b's that come in and match each of them with the c's. To prove that this language is not regular use the pumping lemma with $s = a^p b^p c^{2p}$.
(f) **B**. The PDA for this language non-deterministically decides which of $k \neq i$ or $k \neq j$ is true and then goes on to check that part. To prove that this language is not regular, use the pumping lemma with $s = a^p b^{p!+p} a^{p!+p}$ and pump $i = p!/|y| + 1$ times.
(g) **C**. An intuitive argument is that to check both conditions, one needs two stacks.
(h) **B**. This language is $L \circ L$, where $L = \{0^n 1^n \mid n \geq 0\}$. For non-regularity use pumping lemma with $s = 0^p 1^p$.
(i) **C**. Again one needs two stacks to do this. Consider the following natural one stack PDA candidate (that does not work). First push in all the a's and then the b's. However, to match the a's one has to pop off all the b's, which will not allow the matching of b's later.
(j) **A**. This language is the set of all non-empty strings. To see this note that any $z \in \{a,b\}^*$, with $z \neq \epsilon$ can be written as xy, where $x = \epsilon$ and $y = z$.

a. (2) *language*
Answer: A set of strings.

b. (2) *algorithm*
Answer: A procedure that is guaranteed to complete. (A procedure is a precise sequence of steps that can be followed without thought.)

c. (3) **P** (the complexity class)
Answer: The set of languages that can be decided by a deterministic Turing Machine in a number of steps that is polynomial in the size of the input.

d. (3) **NP-Complete**
Answer: The set of languages that are in NP (they have polynomial time verifiers) and NP-Hard (every problem in NP is polynomial-time reducible to every problem in NP-Complete).

That is, a string $\langle M, w\rangle$ is in *RECIPROCAL* if either (1) M accepts w and M accepts w^R or (2) M rejects w and M rejects w^R or (3) M does not halt on w and M does not halt on w^R. (A full credit answer *must include a clear and convincing proof* supporting your answer.)

Answer: We prove by contradiction using a reduction from A_{TM}.

Assume the language *RECIPROCAL* is decidable. Then, there exists a TM, M_R, that decides *RECIPROCAL*. We show how to use it to construct a TM which decides the language $A_{TM} = \{\langle M, w\rangle | M \text{ accepts } w\}$, which we know is undecidable.

$M_A(\langle M, w\rangle) =$
a. Construct a TM machine M', with input denoted by x that:
   (a) If the input x is w^R, accept.
   (b) Otherwise, simulate M on w and output the result.
b. Run M_R with input $\langle M', w\rangle$ and output the result.

Since M' always accepts w^R then M_R accepts $\langle M', w\rangle$ only when M accepts w. Therefore M_A is a TM that decides A_{TM} which we know is impossible. Therefore, the assumption that *RECIPROCAL* is decidable is false and *RECIPROCAL* is undecidable.

ution We present the context-free grammar rules for each case.
1. *The set of strings over the alphabet $\Sigma = \{a, b\}$ with twice as many a's as*

$$S \rightarrow SS \mid aaSb \mid abSa \mid baSa \mid$$

2. *The complement of the language $\{a^n b^n \mid n \geq 0\}$.*

$$S \rightarrow T \mid aU \mid Vb$$
$$T \rightarrow aT \mid Ta \mid bT \mid Tb \mid ba$$
$$U \rightarrow aU \mid W$$
$$V \rightarrow Vb \mid W$$
$$W \rightarrow aWb \mid \epsilon$$

3. $\{w\#x \mid w^R$ *is a substring of x for $w, x \in \{0,1\}$*$\}$.

$$S \rightarrow S0 \mid S1$$
$$S \rightarrow T$$
$$T \rightarrow 0T0 \mid 1T1$$
$$T \rightarrow U$$
$$U \rightarrow U0 \mid U1 \mid \#$$

4. $\{w \mid w = w^R$ *for* $w \in \{0,1\}^*\}$.

$$S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \epsilon$$

Problem 5. [20 points]
This problem refers to languages over some finite alphabet Σ. We suggest that you use closure properties to answer the following questions.
(a). [10 points] Show that if L is a recursive language and F is a finite language, then $L - F$ is recursive. (The notation $L - F$ denotes all strings in L that do not belong to F.)

Solution: Clearly, $L - F = L \cap \overline{F}$. All finite languages are regular and hence recursive. Since recursive languages are closed under complement, $\overline{F}$ must be recursive. Since recursive languages are closed under intersection, $L \cap \overline{F}$ must be recursive.
(b). [10 points] Show that if L is a non-recursive language and F is a finite language, then $L \cup F$ is non-recursive.

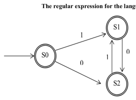
Solution: Define $F_1 = F \cap \overline{L}$ – these are the strings in F that do not belong to L. Clearly, F_1 is finite since it is a subset of a finite language F.
Assume that $L_1 = L \cup F$ is recursive, for the sake of contradiction. Now, observe that $L = L_1 - F_1$. By part (a), if L_1 is recursive then $L_1 - F_1$ must also be recursive. This implies that L is recursive which contradicts the given fact that L is non-recursive. If follows that $L_1 = L \cup F$ must also be non-recursive.

Let L = {w | $w \in \{0,1\}^*$ and w contains neither 00 nor 11 as a substring}.

(a) The state diagram of a finite automaton (DFA or NFA) accepting L.

(b) A regular expression denoting L.

The regular expression for the language L is (1 ∪ε) (01)* (0 ∪ε).



(c) A context-free grammar

S → A T B
T → 01T | ε
A → 1 | ε
B → 0 | ε

(a) If a PDA M actually pushes a symbol on its stack then $L(M)$ is not regular.
(b) For every infinite regular language L over Σ there are strings $x, y, z \in \Sigma^*$ such that $|y| \geq 1$ and for every $k \geq 0$, $xy^kz \in L$.
(c) There is no algorithm to tell, given an arbitrary string P whether or not P is the ASCII for a syntactically correct C program.
(d) If L is a CFL and $L = K \cap R$ for a regular language R then K is a CFL.
(e) If L is not a CFL and $L = K \cap R$ for a regular language R then K is not a CFL.
(f) There is no algorithm to tell, given an arbitrary program P and an input x, whether or not P runs forever on input x.
(g) If the stack in a PDA M only has capacity for 100 characters on any input then $L(M)$ is regular.
(h) If L is accepted by some PDA M then L^R is accepted by some PDA M'.
(i) There is no algorithm to tell, given an arbitrary CFG G and input x whether or not $x \in L(G)$.
(j) Context-free languages are more interesting than regular languages.

(a) **False**. The PDA could just push a "dummy" symbol onto the stack and never really use it in its' computation.
(b) **True**. Use the pumping lemma for regular languages (as the language is infinite, it will have an s in the language such that $|s| \geq p$, where p is the pumping length).
(c) **False**. The set of syntactically correct programs can be generated by a context-free grammar. Checking whether an input w can be generated by a CFG G is exactly the same as deciding A_{CFG}, which is decidable.
(d) **False**. Let $R = \emptyset$. Then $L = \emptyset$ which is regular (and hence, a CFL). Now K can be any arbitrary language.
(e) **True**. This is just the contra-positive of the closure of intersection of a CFL with a regular language.
(f) **True**. The statement is "equivalent" to the halting problem. In particular, if the statement is false, then there would exist a decider (call it D) that can tell given an arbitrary TM P and w, whether P loops on w or not. Given such a decider, one can construct a decider for A_{TM} as follows. On input $\langle P, w\rangle$, first run the input on D. If D says yes, then simulate P on w and return the simulation's answer , otherwise reject. Of course, A_{TM} is undecidable and hence, we have a contradiction.
(g) **True**. The number of possible stack arrangements is finite (in fact, bounded by $(|\Sigma| + 1)^{100}$). Thus, one can "encode" all the possibilities in the states of a (huge) DFA.
(h) **True**. Just "reverse" the transitions.
(i) **False**. A_{CFG} is decidable.
(j) **True**. There are languages that are context-free but not regular.

1. ______ The union of a regular language with a context-free language must be context-free.
TRUE: Since context-free languages are closed under union.
2. ______ The language $\{a^i b^j c^k d^l \mid i = j$ and $k = l\}$ is context-free.
TRUE: this is just the concatenation of two context-free languages.
3. ______ The language $\{a^i b^j c^k d^l \mid i + k = j + l\}$ is context-free.
TRUE: a PDA can easily add and subtract the number of a's, b's, c's, and d's on stack.
4. ______ The class of non-regular languages is closed under complementation.
TRUE: If the complement of a language L is regular, then L itself must be regular to begin with; therefore, the complement of a non-regular language must be non-regular.
5. ______ The class of non-context-free languages is closed under complementation.
FALSE: For example, $L_{ww} = \{ww \mid w \in \{0,1\}^*\}$ is non-context-free, but its complement is context-free.
6. ______ If L_1 and L_2 are context-free, then the language $L_1 - L_2$ must be context-free.
FALSE: consider the case where $L_1 = \Sigma^*$; then the difference is like complementation and CFLs are not closed under complementation.
7. ______ If L_1 is context-free and L_2 is regular, then the language $L_1 - L_2$ must be context-free.
TRUE: Since $L_1 - L_2 = L_1 \cap \overline{L_2}$ and the context-free languages are closed under intersection with regular languages.
8. ______ The context-free grammar $S \rightarrow a \mid ab \mid SS \mid Sb$ is ambiguous.
TRUE: Since $S \Rightarrow ab$ and $S \Rightarrow Sb \Rightarrow ab$ shows that ab has two parse trees.

3. (25 points)
Let G = (V,Σ,R,S) be a context-free grammar, where Σ = {w, c, {, }, ;, a}, V = Σ ∪ {S} and R is given by:
S → wcS
S → {S}
S → S;S
S → a

(This grammar models some statements in C-like languages, where w represents **while**, c represents a Boolean condition, {…} represents a compound statement, and a represents a non-**while** statement.)

(a) Show two distinct leftmost derivations, and the corresponding parse trees, for the string: wca;a

Derivation 1: S ⇒ wcS ⇒ wcS;S ⇒ wca;S ⇒ wca;a
Derivation 2: S ⇒ S;S ⇒ wcS;S ⇒ wca;S ⇒ wca;a

(Draw the two parse trees according to the above two derivations (each level of the tree corresponds to 1 step in the derivation). See page 98 in the text for an example

(b) Is *G* ambiguous? Why or why not?

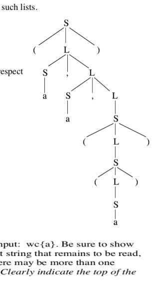
Yes, because there exists a string in L(G) that has 2 distinct parse trees (2 distinct leftmost derivations).

Define a well-formed list to be either the single "atom" a, or a sequence of one or more well-formed lists separated by commas and enclosed in parentheses. For example, "a" is a list, as are "(a)" and "(a,a,((a)))" but "((),aa)" is not (for two reasons).

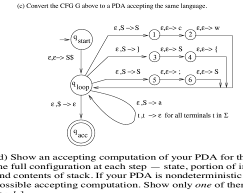
(a) Give a context-free grammar that generates all such lists.

S → a | (L)
L → S | S, L

(b) Give a parse tree for the list "(a,a,((a)))" with respect



(c) Convert the CFG G above to a PDA accepting the same language.



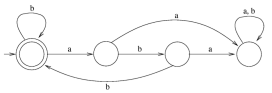
(d) Show an accepting computation of your PDA for the input: wc{a}. Be sure to show the full configuration at each step — state, portion of input string that remains to be read, and contents of stack. If your PDA is nondeterministic, there may be more than one possible accepting computation. Show only *one* of them. [*Clearly indicate the top of the stack.*]

(Hint: Might be helpful to construct the parse tree first)

	State	Input to be read	Stack
1.	q_{start}	wc{a}	ε
2.	q_{loop}	wc{a}	$S [top]
3.	1	wc{a}	$S
4.	2	wc{a}	$Sc
5.	q_{loop}	wc{a}	$Scw
6.	q_{loop}	c{a}	$Sc
7.	q_{loop}	{a}	$S
8.	3	{a}	$}
9.	4	{a}	$}S
10.	q_{loop}	{a}	$}S{
11.	q_{loop}	a}	$}S
12.	q_{loop}	a}	$}a
13.	q_{loop}	}	$}
14.	q_{loop}	ε	$
15.	q_{acc}	ε	ε

Let L be the set of strings in {a,b}* such that each a, if any, has two b's immediately to its right. Give:

(a) a finite automaton accepting L.



(b) a regular expression denoting L.

(b ∪ abb)*

(c) a context-free grammar generating L.

S → bS | abbS | ε

(a) [10 points] It is not known whether the complement of every NP language is also in NP. The following is a fallacious proof that this is always the case. Find the error in the proof.

Theorem. If $L \in NP$, then $\overline{L} \in NP$.

Proof: Given any polynomial-time NTM N for the language L, make its final states into non-final states and vice versa. Clearly, the resulting NTM N' accepts exactly the language $\overline{L}$ and has a polynomial running time. Hence, $\overline{L} \in NP$.

Solution: The machine N' will accept all those strings for which N behaves as follows: *at least one execution path of N leads to a non-final state of N*. But this could include many strings which are accepted by N. In fact, the set of strings rejected by N' are only those strings which are rejected by N along *all possible* execution paths of N. Thus, it is clear that the machine N' does not accept exactly the language $\overline{L}$ but in fact some superset of that language.

(b) [10 points] Let $L_1, L_2 \subseteq \Sigma^*$ be such that $L_1 <_{poly} L_2$. Prove that $\overline{L}_1 <_{poly} \overline{L}_2$.

Solution: We can use exactly the same reduction as in $L_1 <_{poly} L_2$. Let f be the reduction function which gives $L_1 <_{poly} L_2$. Clearly, it has the following properties: f is computable in polynomial time, and $w \in L_1$ if and only if $f(w) \in L_2$. Observe that the latter property is equivalent to saying that $w \notin L_1$ if and only if $f(w) \notin L_2$. This implies that $\overline{L}_1 <_{poly} \overline{L}_2$.

(c) [10 points] Suppose that L_1 is NP-complete and $\overline{L}_1 \in NP$. Prove that for all $L \in NP$, it must be the case that $\overline{L} \in NP$.

Solution: Since L_1 is NP-complete, it must be NP-hard which means that for all languages $L \in NP$ it must be the case that $L <_{poly} L_1$. By part (b) of this problem, it follows that for all languages $L \in NP$ it must be the case that $\overline{L} <_{poly} \overline{L}_1$. Now, since $\overline{L}_1 \in NP$, it must be the case that for all languages $L \in NP$, $\overline{L} \in NP$. (This is based on the fact that: if $X <_{poly} Y$ and $Y \in NP$, then $X \in NP$.)