Automata, Computability and Complexity with Applications

Additional Exercises

Elaine Rich

Part I: Introduction

1 Why Study Automata Theory?

2 Languages and Strings

1) Let L be the language {`Austin`, `Houston`, `Dallas`}. How many elements are there in L^*?

Countably infinitely many.

2) Let L = {w ∈ {`a`, `b`}* : every `a` in w is immediately followed by a `b`}. List the first six elements in a lexicographic enumeration of L.

ε, `b`, `ab`, `bb`, `abb`, `bab`

3) Let $L = \{w \in \{1, 2, 3\}^* : |w| \text{ is even}\}$. List the first twelve elements in a lexicographic enumeration of L.

ε, `11`, `12`, `13`, `21`, `22`, `23`, `31`, `32`, `33`, `1111`, `1112`

4) Are the following sets closed under the following operations? If not, what are their respective closures?
   a) The even length strings over the alphabet {`a`, `b`} under Kleene star.

   Closed.

   b) The odd length strings over the alphabet {`a`, `b`} under concatenation.

   Not closed. The closure is all strings over the alphabet {`a`, `b`} with length at least 1.

5) For each of the following statements, state whether it is *True* or *False*. Prove your answer.
   a) $\forall L\ ((L^+)^+ = L^+)$.

   True.

   b) $\forall L\ ((L^*)^+ = (L^+)^*)$.

   True.

   c) $\forall L\ (L^* = L^+ \cup \varnothing)$

   False.

   d) $\forall L_1, L_2, L_3\ ((L_1L_2L_3)^* = L_1^*L_2^*L_3^*)$.

   False.

   e) $\forall L_1, L_2\ ((L_1^* \cup L_2^*) = (L_1^* \cup L_2^*)^*)$.

   False.

   f) $\forall L\ (L^* L = L^+)$.

   True.

g) $\forall L_1, L_2, L_3 \; (L_1^* \; (L_2 \cup L_3)^+ = (L_1^* \; L_2^+ \cup L_1^* \; L_3^+))$.

False.

h) $\forall L \; (\varnothing \; L^* = \varnothing)$.

True.

i) $\forall L_1, L_2 \; ((L_1 - L_2) = (L_2 - L_1))$.

False. Counterexample:
Let $L_1 = \{\mathtt{a}\}$ and $L_2 = \varnothing$. Then $L_1 - L_2 = \{\mathtt{a}\}$, but $L_2 - L_1 = \varnothing$.

j) $\forall L_1, L_2, L_3 \; (((L_1 \; L_2) \cup (L_1 \; L_3))^* = (L_1 \; (L_2 \cup L_3))^*)$.

True.

k) $\forall L \; (((L \cup \varepsilon)^* \; (L \cup \varepsilon)^*)^* = L^*)$.

True.

l) $\forall L_1, L_2$ where $L_1 \neq L_2 \; ((L_1 \; L_2) \neq (L_2 \; L_1))$.

False. Counterexample:
Let $L_1 = \varnothing$ and $L_2 = \mathtt{a}^*$. Then $L_1 \; L_2 = \varnothing = L_2 \; L_1$.

m) If $L_1 - L_2$ is finite, then at least one of L_1 or L_2 must also be finite.

False. Counterexample:
Let $L_1 = \{\mathtt{a}\} \cup \mathtt{b}^*$. Let $L_2 = \mathtt{b}^*$. Then $L_1 - L_2 = \{\mathtt{a}\}$, which is finite. But neither L_1 or L_2 is.

n) The set of strings that correspond to binary encodings of positive integers is closed under concatenation.

True.

3 The Big Picture: A Language Hierarchy

4 Some Important Ideas Before We Start

1) Recall the function *chop*(*L*) defined in Example 4.10. *Chop*(*L*) = {*w* : ∃*x* ∈ *L* (*x* = x_1cx_2, $x_1 \in \Sigma_L^*$, $x_2 \in \Sigma_L^*$, $c \in \Sigma_L$, $|x_1| = |x_2|$, and $w = x_1x_2$)}. What is *chop*({$\mathtt{a}^n\mathtt{b}^{2n}$: $n \geq 0$})?

{$\mathtt{a}^n\mathtt{b}^{2n-1}$: $n \geq 1$ and odd}.

2) Are the following sets closed under the following operations? Prove your answer. If a set is not closed under the operation, what is its closure under the operation? Assume an alphabet Σ = {a, b}.
   a) FIN under complement.

   No, which we prove by counterexample. Let *L* = {a}. *L* is finite. But ¬*L* is infinite.

   b) INF under complement.

   No, which we prove by counterexample. Let *L* = {a, b}*. *L* is infinite. But ¬*L* = ∅ is infinite.

   c) FIN under reverse.

   Yes. For any language *L*, there is a one-to-one correspondence between the elements of *L* and L^R, so the cardinalities of the two sets are the same.

   d) INF under reverse.

   No, which we prove by counterexample. Let *L* = {a}. *L* is finite. But ¬*L* is infinite.

   e) FIN under Kleene star.

   No, which we prove by counterexample. Let *L* = {a}. *L* is finite. But *L** is infinite.

   f) INF under Kleene star.

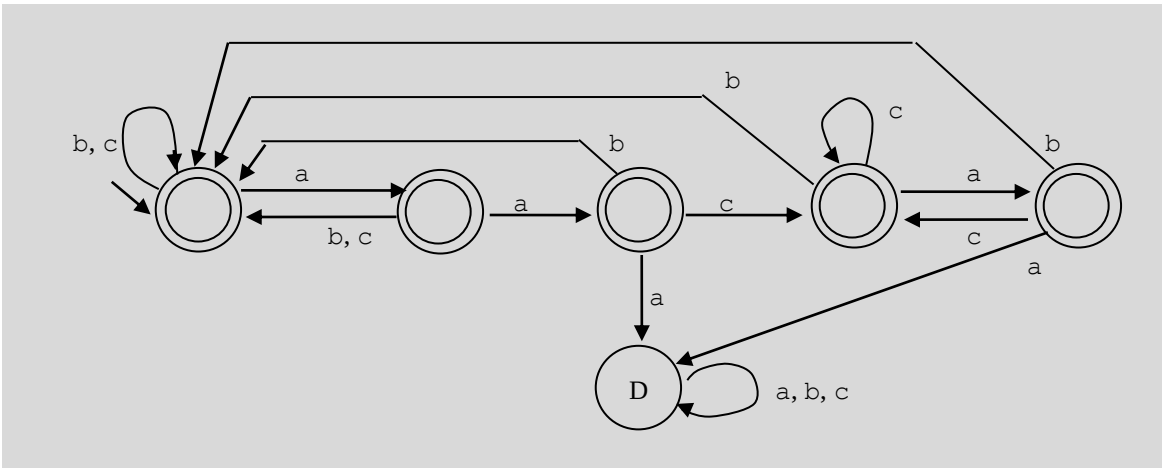
   Yes. For any language *L*, every string in *L* must also be in *L**. So *L** has at least as many elements as *L* does.

Part II: Regular Languages

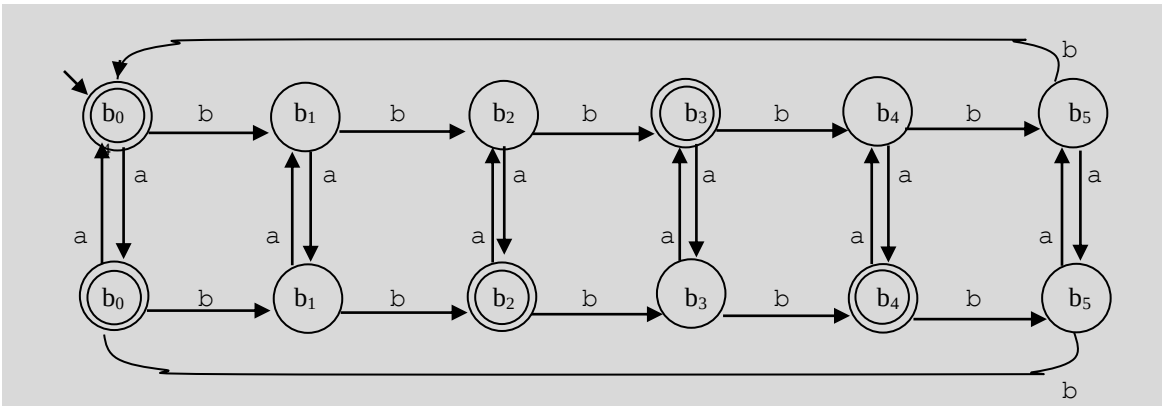
5 Finite State Machines

1) Show a DFSM to accept each of the following languages:
   a) {*w* ∈ {a, b, c}* : *w* does not end in a}.

   b) {*w* ∈ {a, b, c}* : every pair of aa sequences in *w* is separated by at least one b}.



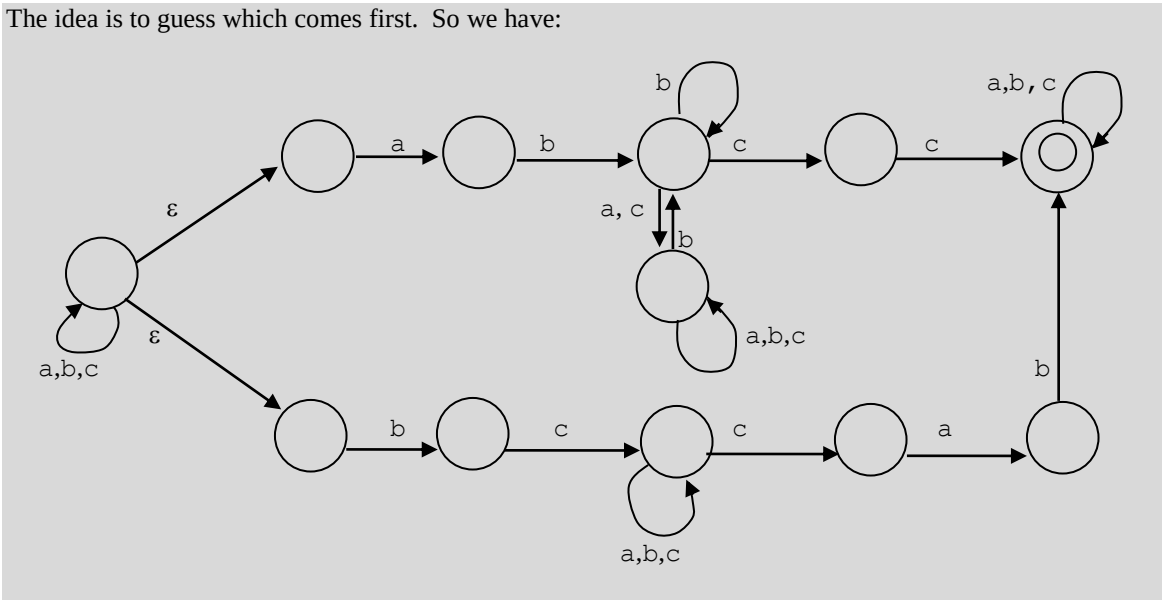
   c) {*w* ∈ {a, b}* : if $\#_a(w)$ is even, then $\#_b(w)$ is divisible by 3; if $\#_a(w)$ is odd, then $\#_b(w)$ is divisible by 2}



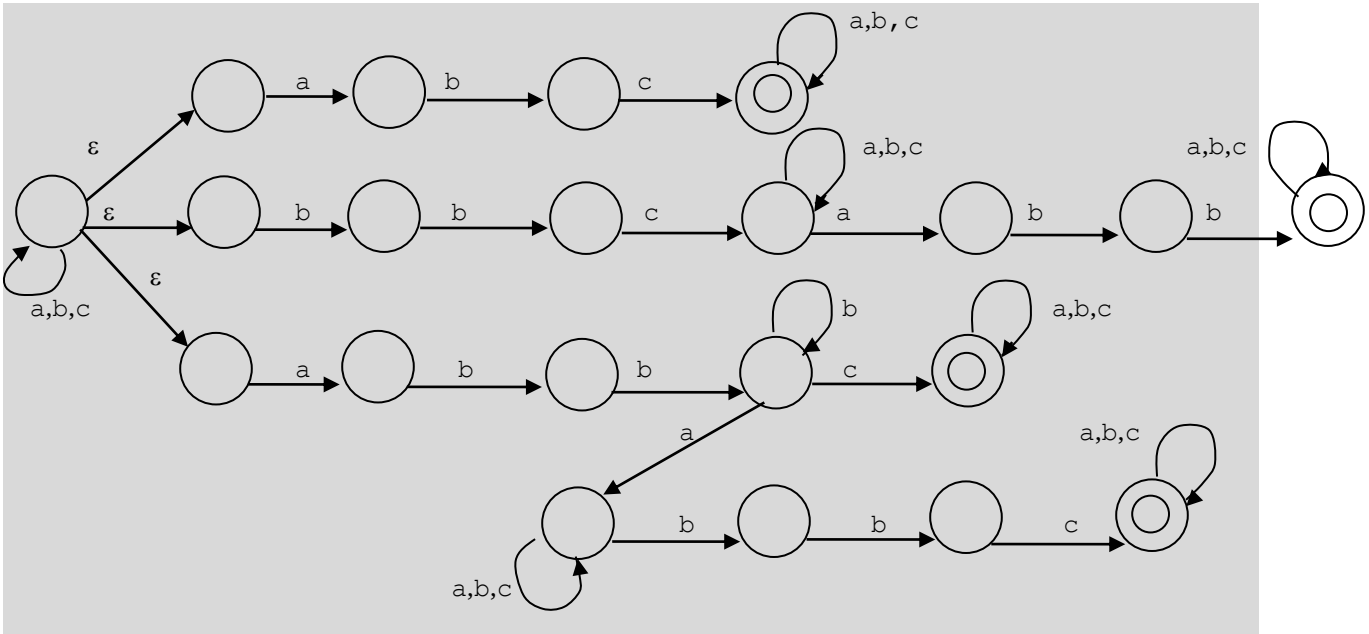
2) Show a possibly nondeterministic FSM to accept each of the following languages:
   a) {*w* ∈ {a, b}* : *w* contains at least one instance of babab or abbb or bbbbb}.

b) {*w* ∈ {a, b, c}*: *w* contains both ab and bcc as substrings}. (Note that they may overlap. So abcc ∈ *L*.)

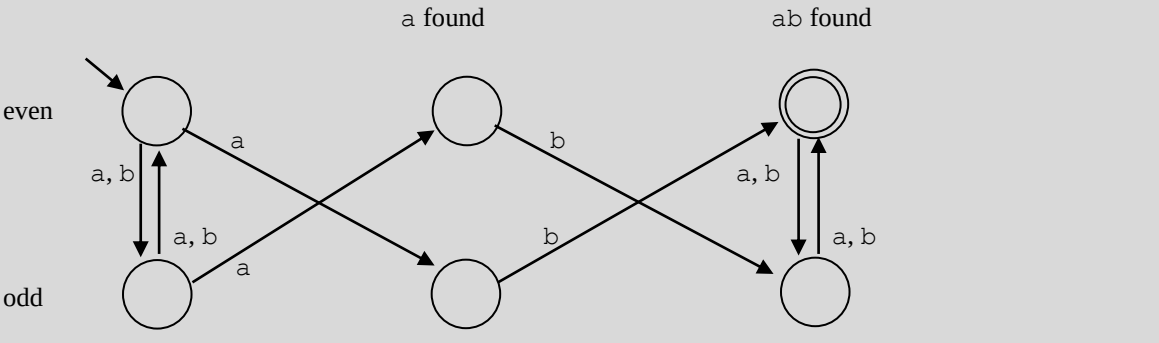
The idea is to guess which comes first. So we have:



c) {*w* ∈ {a, b, c}*: *w* contains either or both of: (1) both the substring abb and the substring bbc, or (2) the substring abc}.



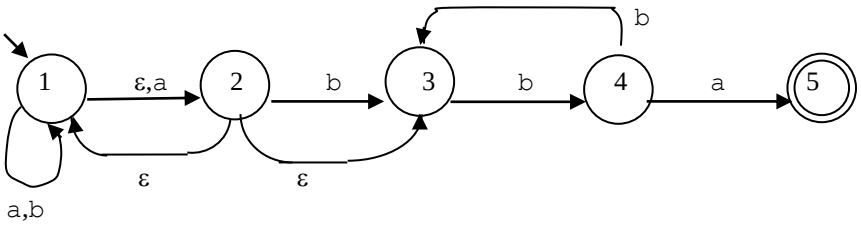
d) {*w* ∈ {a, b}* : *w* has even length and contains the substring ab}.



3) Suppose that a language *L* can be accepted by a four-state NDFSM *M*. Can we guarantee that it is possible to construct a DFSM with fewer than 10 states that accepts *L*? Why or why not?

No. If we construct a DFSM *M′* to accept *L*, it is possible that *M′* will have one state for each set of states that *M* could be in. So the possibilities include each element of $\wp(K_M)$ except ∅. $|\wp(K_M)| = 2^4 = 16$. So *M′* may need as many as 15 states.

4) For each of the following NDFSMs, use *ndfsmtodfsm* to construct an equivalent DFSM. Begin by showing the value of *eps*(*q*) for each state *q*:
a)

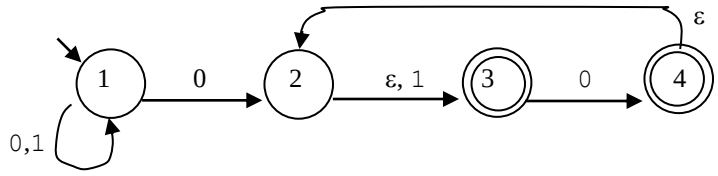


eps(1) = {1, 2, 3}
eps(2) = {2, 3, 1}
eps(3) = {3}
eps(4) = {4}
eps(5) = {5}

{1, 2, 3}, a, {1, 2, 3}
, b, {1, 2, 3, 4}
{1, 2, 3, 4}, a, {1, 2, 3, 5}
, b, {1, 2, 3, 4}
{1, 2, 3, 5}, a, {1, 2, 3}
, b, {1, 2, 3, 4}

All states are accepting.

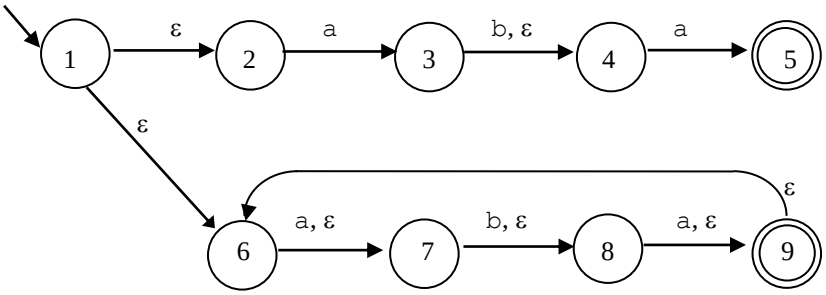
b)



{1}, 0, {1, 2, 3}
1, {1}
{1, 2, 3}, 0, {1, 2, 3, 4}
1, {1, 3}
{1, 2, 3, 4}, 0, {1, 2, 3, 4}
1, {1, 3}
{1, 3}, 0, {1, 2, 3, 4}
1, {1}
The start state is {1}. All but {1} are accepting.

c)

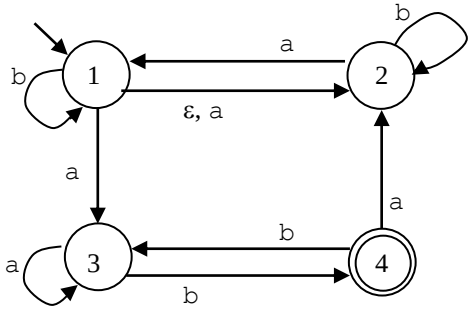
.



eps(1) = {1, 2, 6, 7, 8, 9}
eps(2) = {2}
eps(3) = {3, 4}
eps(4) = {4}
eps(5) = {5}
eps(6) = {6, 7, 8, 9}
eps(7) = {7, 8, 9, 6}
eps(8) = {8, 9, 6, 7}
eps(9) = {9, 6, 7, 8}

Start state is {1, 2, 6, 7, 8, 9}.
{1, 2, 6, 7, 8, 9}, a, {3, 4, 6, 7, 8, 9}
b, {6, 7, 8, 9}
{3, 4, 6, 7, 8, 9}, a, {5, 6, 7, 8, 9}
b, {4, 6, 7, 8, 9}
{6, 7, 8, 9}, a, {6, 7, 8, 9}
b, {6, 7, 8, 9}
{5, 6, 7, 8, 9}, a, {6, 7, 8, 9}
b, {6, 7, 8, 9}
{4, 6, 7, 8, 9}, a, {5, 6, 7, 8, 9}
b, {6, 7, 8, 9}
All states are accepting.

d)



eps(1) = {1, 2}
eps(2) = {2}
eps(3) = {3}
eps(4) = {4}

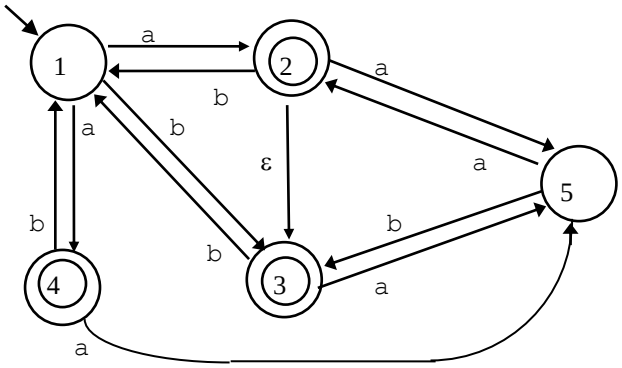
Start state is {1, 2}.

{1, 2},	a, {1, 2, 3}
	b, {1, 2}
{1, 2, 3},	a, {1, 2, 3}
	b, {1, 2, 4}
{1, 2, 4},	a, {1, 2, 3}
	b, {1, 2, 3}

{1, 2, 4} is the only accepting state.

e)

.



eps(1) = {1}
eps(2) = {2, 3}
eps(3) = {3}
eps(4) = {4}
eps(5) = {5}
eps(6) = {6}

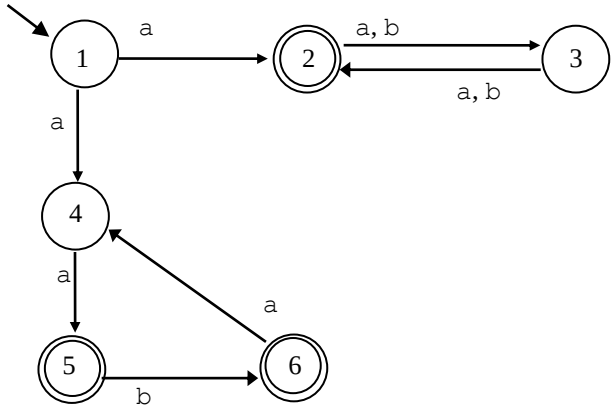
Start state is {1}.

{1},	a, {2, 3, 4}
	b, {3}
{2, 3, 4},	a, {5}
	b, {1}
{3},	a, {5}
	b, {1}
{5},	a, {2, 3}
	b, {3}
{2, 3},	a, {5}
	b, {1}

{2, 3, 4}, {3}, {2, 3} are accepting state.

5) Let *M* be the following NDFSM. Construct (using *ndfsmtodfsm*), a DFSM that accepts $\neg L(M)$.

.



1) Convert to deterministic: *eps*(1) = {1}, so the start state is {1}.

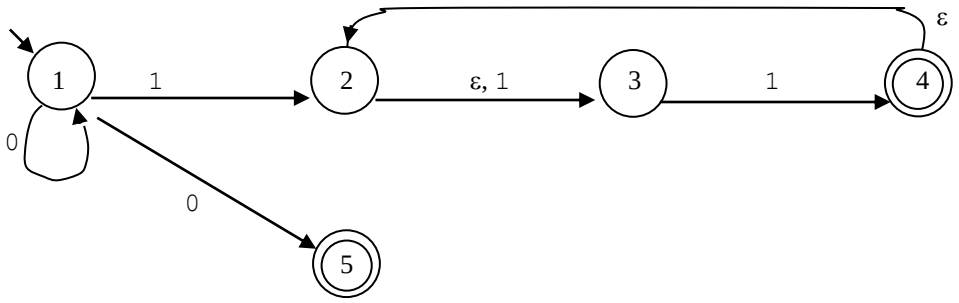
{1},	a, {2, 4}
{2, 4},	a, {3, 5}
	b, {3}
{3, 5},	a, {2}
	b, {2, 6}
{3},	a, {2}
	b, {2}
{2},	a, {3}
	b, {3}
{2, 6},	a, {3, 4}
	b, {3}
{3, 4},	a, {2, 5}
	b, {2}
{2, 5},	a, {3}
	b, {3, 6}
{3, 6},	a, {2, 4}
	b, {2}

All states except {1}, {3}, and {3, 4}are accepting states.

2) Add the dead state, D, and the transitions:
{1}, b, D
D, a, D
D, b, D

3) Swap accepting and nonaccepting states, making all states except D, {1}, {3}, and {3, 4} nonaccepting.

6) Let *M* be the following NDFSM. Construct (using *ndfsmtodfsm*), a DFSM that accepts $\neg L(M)$.



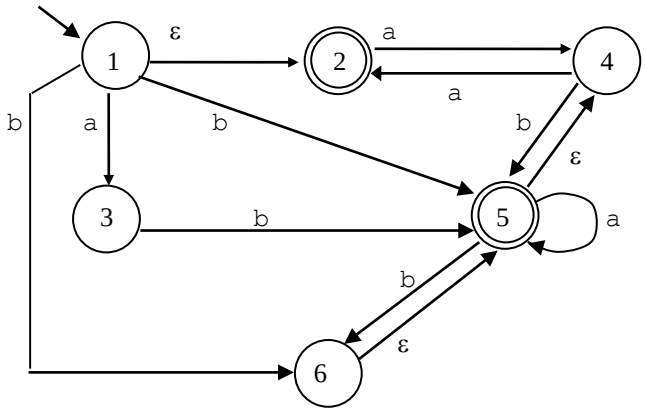
1) Convert to deterministic: *eps*(1) = {1}, so the start state is {1}.
{1}, 0, {1, 5}
1, {2, 3}
{1, 5}, 0, {1, 5}
1, {2, 3}
{2, 3}, 0, ∅
1, {2, 3, 4}
{2, 3, 4}, 0, ∅
1, {2, 3, 4}
All states except {1} are accepting states.

2) Add the dead state, D, and the transitions:
{2, 3}, 0, D
{2, 3, 4}, 0, D
D, 0, D
D, 1, D

3) Swap accepting and nonaccepting states, making all states except D and {1} nonaccepting.

7) Let *M* be the following NDFSM. Construct (using *ndfsmtodfsm*), a DFSM that accepts $\neg L(M)$.

.



1) Convert to deterministic:

eps(1) = {1,2} *eps*(5) = {5,4} *eps*(6) = {6,5,4}
The start state is {1, 2}.

{1, 2},	a, {3, 4}
	b, {6, 5, 4}
{3, 4},	a, {2}
	b, {5, 4}
{6, 5, 4},	a, {5, 4, 2}
	b, {6, 5, 4}
{2},	a, {4}
	b, {}
{5, 4},	a, {5, 4, 2}
	b, {6, 5, 4}
{5, 4, 2},	a, {5, 4, 2}
	b, {6, 5, 4}
{4},	a, {2}
	b, {5, 4}

All states except {6, 5, 4}, {5, 4} and {5, 4, 4} are accepting states.

2) Add the dead state, D, and the transitions:

{2},	b, D
D,	a, D
D,	b, D

3) Swap accepting and nonaccepting states, making D, {1, 2}, {3, 4}, {2} and {4} accepting.

8) We consider the relationship between and FSMs.

a) Give an example of a regular language *L* such that has precisely three equivalence classes.

One simple example is (a ∪ b ∪ c)*.

b) Show a minimal DFSM that accepts *L*.

Whatever *L* is chosen in a), the machine shown here will have exactly three states.

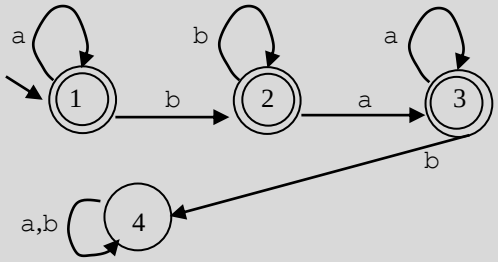
9) (Note: Some of these problems should be given after regular expressions have been introduced.) For each of the following languages L:

(i) Describe the equivalence classes of $\approx_L$.

(ii) If the number of equivalence classes of $\approx_L$ is finite, construct the minimal DFSM that accepts L.

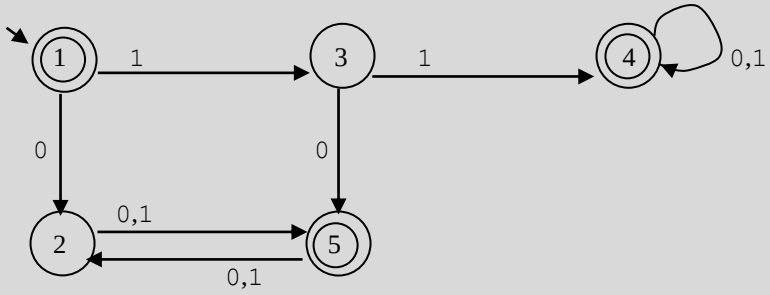
a) $(a \cup \varepsilon)a^*b^*a^*$.

[1] [a^*]
[2] [a^*bb^*]
[3] [$a^*bb^*aa^*$]
[4] [everything else]



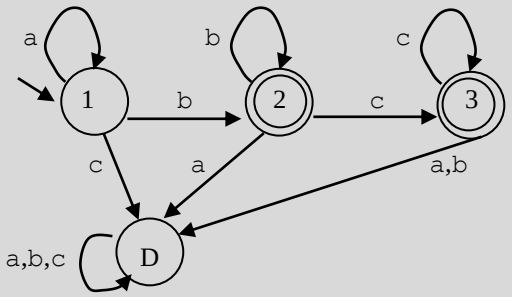
b) $\{w \in \{0, 1\}^* :$ every odd length string in L begins with $11\}$

[1] [ε]
[2] [0 and odd length strings of length greater than 1 that do not begin 11]
[3] [1]
[4] [$11(0 \cup 1)^*$]
[5] [even length strings that do not begin 11]



c) $\{a^*b^nc^* : n \geq 1\}$

[1] [a*]
[2] [a*bb*]
[3] [a*bb*cc*]
[4] [everything else]



d) $\{a^*b^nc^m : n \geq 1 \text{ and } m = 2n\}$

[1] [a*]
[2] [a*b]
[3] [a*bb]
[4] [a*bbb]

and so forth, counting just b's

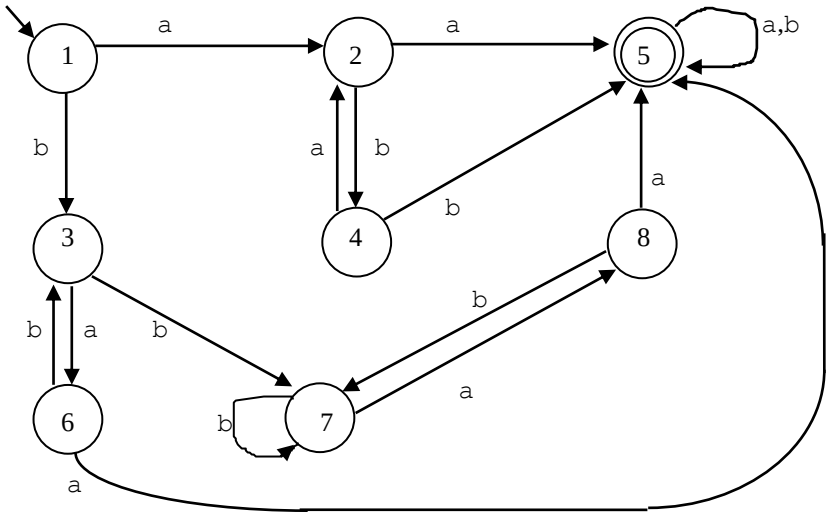
[strings in L]
[strings with letters out of order]

[a*ba]
[a*bba]
[a*bbaa]

and so forth, counting a's and b's

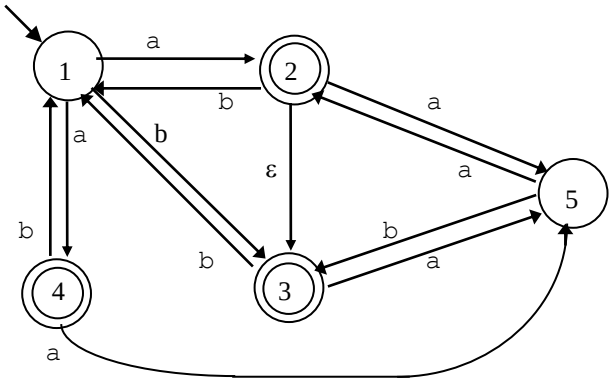
The number of equivalence classes of $\approx_L$ is infinite. L is not regular.

10) Minimize the following FSM M:



11) Minimize the following FSM *M*:

.



Step 1: Convert to deterministic (and name the states for later convenience):
eps(1) = {1}, so the start state is {1}.

[A] {1}, a, {2, 3, 4}
b, {3}

[B] {2, 3, 4}, a, {5}
b, {1}

[C] {3}, a, {5}
b, {1}

[D] {5}, a, {2, 3}
b, {3}

[E] {2, 3}, a, {5}
b, {1}

Accepting states are B, C, and E.

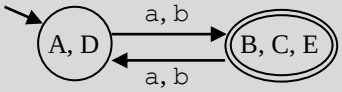
Step 2: Minimize:

$\equiv_0$ = [B, C, E]
[A, D]

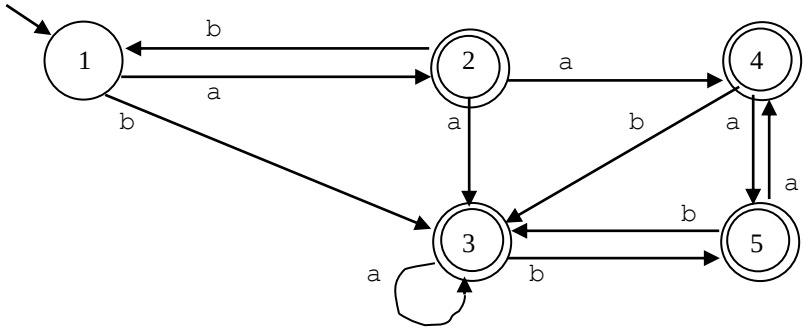
$\equiv_1$ = B, a, D C, a, D E, a, D
b, A b, A b, A

A, a, B D, a, E
b, C b, C

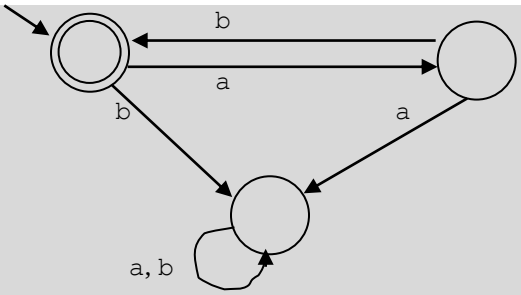
So we have two states: [B, C, E] and [A, D]. No further splitting is required. So a minimal DFSM *M** that accepts *L* is:



12) Consider the following FSM *M*:



a) Show a mimimal FSM *M′* that accepts ¬*L*(*M*).

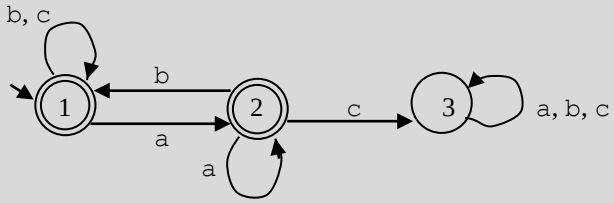


b) Write a regular expression for ¬*L*(*M*).

(ab)*

13) Construct a Büchi automaton to accept each of the following languages of infinite length strings:

a) {*w* ∈ {a, b, c}$^{\omega}$: no a is immediately followed by a c}.



6 Regular Expressions

1) [Luay Nakhleh] What is the cardinality of L = a*b* - ab?

The language defined by a*b* is countably infinite. When we subtract a single element, we still have a countably infinite set.

2) Describe in English, as briefly as possible, the language described by each of these regular expressions:
   a) a*(ba)a*(ba ∪ a*)*a*.

The set of strings over the alphabet {a, b} that contain at least one b and where every b is immediately followed by at least one a.

3) Write a regular expression to describe each of the following languages:
   a) {$w \in$ {a, b}* :w contains exactly one occurrence of the substring aaa}.

(ab ∪ aab ∪ b)* aaa (ba ∪ baa ∪ b)*

   b) {$w \in$ {a, b}* : w contains bba as a substring that starts in an even numbered position in the string (where numbering starts at 1)}.

(aa ∪ ab ∪ ba ∪ bb)* bba (a ∪ b)*

   c) {$w \in$ {a, b, c}* : w ends with a character that occurs no where else in it}.

(a ∪ b)*c ∪ (b ∪ c)*a ∪ (a ∪ c)*b

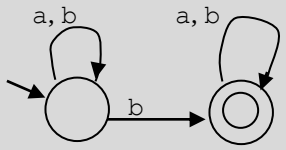
   d) {$w \in$ {a, b}$^+$: w starts and ends with the same character}.

a ∪ b ∪ a(a ∪ b)*a ∪ b(a ∪ b)*b

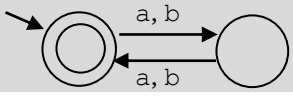
   e) {$w \in$ {a, b}* : w does not contain any occurrences of an a immediately followed by another a}

(b*ab)*b*(a ∪ ε) Or, alternatively: (b*(ab)*)*(a ∪ ε)

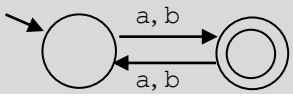
4) Draw an FSM to accept each of the following languages:
   a) (a ∪ b)*a*b(a ∪ b)*



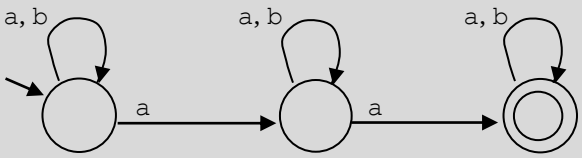
   b) ((a ∪ b) (a ∪ b))*



c) (a ∪ b) (a (a ∪ b) ∪ b (a ∪ b))*

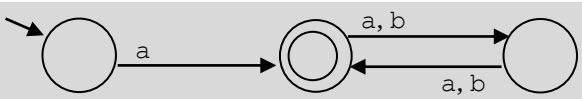


d) (a ∪ ε) (a ∪ b)* a (a ∪ b)* a (a ∪ b)*(a ∪ b)*

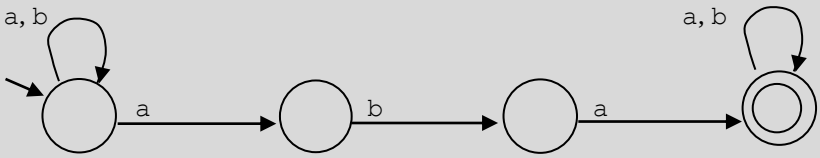


5) For each of the following regular expressions α, draw the smallest FSM that accepts $L(\alpha)$.

a) a ((a ∪ b) (b ∪ a))* ∪ a ((a ∪ b) a)* ∪ a ((b ∪ a) b)*



b) (a ∪ b)* abaa*b* ∪ (a ∪ b)* aba (b ∪ aba ∪ a ∪ bab)*

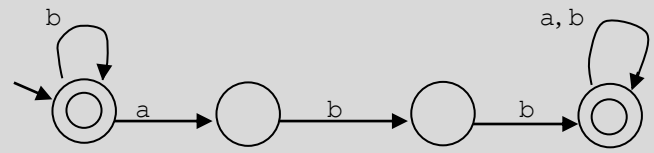


6) Let $L = \{w \in \{a, b\}^* :$ the first a in w is immediately followed by two b's$\}$.

a) Show a regular expression for L.

b*(ε ∪ abb(a ∪ b)*)

b) Show an FSM that accepts L.



7) Let $L = \{w = xcy: x, y \in \{a, b\}^*$ and $|x|$ is even and $|y|$ is odd$\}$. Indicate whether each of the following regular expressions correctly describes L:

a) (a ∪ b)* c (a ∪ b) (a ∪ b)*

No. $|x|$ is not constrainted to be even.

b) (aa ∪ ab ∪ bb ∪ ba)* c (a ∪ b) (a ∪ b)*

No. $|y|$ is not constrainted to be odd.

c) (aa ∪ ab ∪ bb ∪ ba)* c (aa ∪ ab ∪ bb ∪ ba)*(a ∪ b) (a ∪ b)*

No. |*y*| is not constrainted to be odd.

d) (aa ∪ ab ∪ bb ∪ ba)* c (aa ∪ ab ∪ bb ∪ ba)*(a ∪ b)

Yes.

e) (aa ∪ ab ∪ bb ∪ ba)* c (aa ∪ bb)*(ab ∪ ba)*(a ∪ b)

No. Both length requirements must be met, but there are odd length strings, such as abaaa, that cannot be generated as |*y*|.

f) (aa ∪ ab ∪ bb ∪ ba)* c (aa ∪ ab ∪ bb ∪ ba)* a ∪ b

No. Since concatenation has higher precedence than ∪, this expression can generate the string b, which is not in *L*.

8) Let *L* = {*w* ∈ {a, b}* : *w* ends in a}. Indicate whether each of the following regular expressions correctly describes *L*:

a) (a*b* ∪ b*a*)a

No. It cannot generate ababa.

b) (a*b* ∪ b*a*)*a

Yes.

c) (b*a*b*)*a

Yes.

d) (b*a)*b*a*

No. It can generate b.

9) Let *L* = {*w* ∈ {a, b}* : *w* contains at least two a's}. Indicate whether each of the following regular expressions correctly describes *L*:

a) a(b*a)*(a ∪ b)*

No. All the strings it generates start with a. So, for example, it cannot generate baa.

b) (aa ∪ baa)*(a ∪ b)*

No. It can generate ε.

c) b*ab*a(a ∪ b)*

Yes.

d) b*(ab*a)*(a ∪ b)*

Yes.

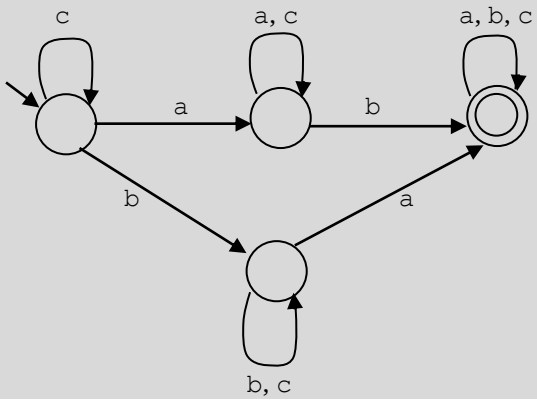
10) Let $L = \{w \in \{a, b, c\}^* : w$ contains at least one a and at least one b$\}$.

a) Show a regular expression for *L*.

The regular expression has two parts: One generates the case where a comes first; the other allows b to come first.

((a ∪ b ∪ c)* a (a ∪ b ∪ c)* b (a ∪ b ∪ c)*) ∪ ((a ∪ b ∪ c)* b (a ∪ b ∪ c)* a (a ∪ b ∪ c)*)

b) Show a DFSM that accepts *L*.

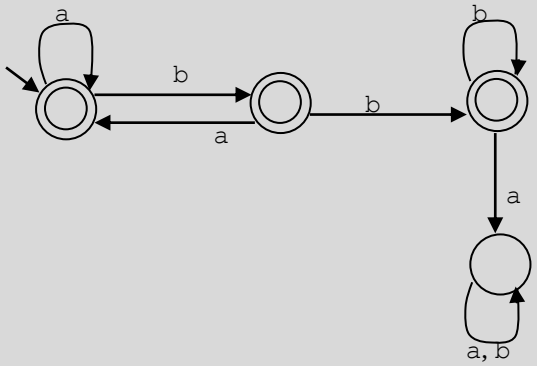


11) Let $L = \{w \in \{a, b, c\}^* : w$ contains no instance of the string bba$\}$.

a) Show a regular expression for *L*.

(a ∪ ba)* b*

b) Show a DFSM that accepts *L*.



12) Let *L* be the language defined by the following regular expression:

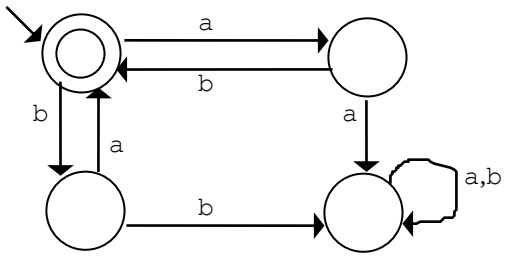
((ab)* c (ab)* c (ab)*)* ∪ ((a ∪ b)* c (a ∪ b)* c (a ∪ b)*)*

Show an FSM that accepts *L*.

The trick here is that the second half of the expression generates all the strings that the first one does. So we can ignore the first one to build a simple FSM.

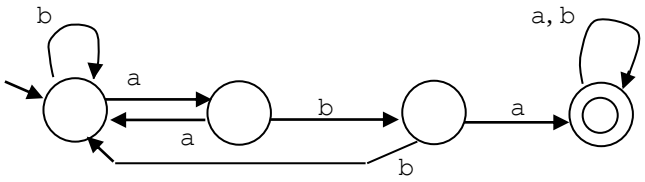
13) Write a regular expression for the language recognized by the each of the following FSMs:

a)



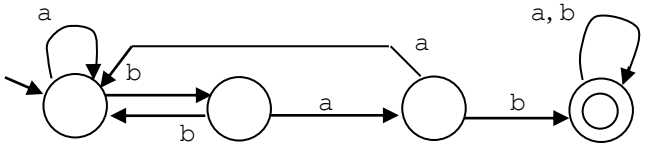
(ab ∪ ba)*

b)



(aa ∪ b ∪ abb)* aba (a ∪ b)*

c)



(a ∪ bb ∪ baa)* bab (a ∪ b)*

14) Simplify each of the following regular expressions:

a) a*b*(a ∪ b)* (a ∪ ε) b*a*

(a ∪ b)*

b) (bb)*(a ∪ b)(ab)*(b ∪ a)(ba)*)*(aa)*

((a ∪ b)(a ∪ b))*

15) [Luay Nakhleh] A language $L \subseteq \Sigma^*$ is called *normal* if and only if at least one of the following holds:

- $L = \varnothing$.
- $L = \{\varepsilon\}$.
- $L = \{a\}$, where $a \in \Sigma$.
- $L = L_1 \cup L_2$, where L_1 and L_2 are normal languages.
- $L = L_1 L_2$, where L_1 and L_2 are normal languages.
- $L = \{ww: w \in L_1\}$, where L_1 is a normal language.

For each of the following assertions, state whether it is *True* or *False* and prove your answer:

a) Every normal language is regular.

True. Notice that every normal language is finite. The definition does not include the Kleene star operator, or any other operator that can build an infinite language from one or more primitive (and finite) ones. Every finite language is regular.

b) Every regular language is normal.

False. By the same argument, no infinite language is normal. Yet there are infinite languages, for example a*, that are regular.

16) For each of the following statements, state whether it is *True* or *False*. Prove your answer.

a) $(a \cup b)^*(a^* \cup \varepsilon)a(a \cup b)^*a = aa(a \cup b)^*$.

False. The string baa can be generated by the first expression, but all strings generated by the second expression must start with aa.

b) $(a \cup b)^* = (a \cup b \cup ba \cup ab)^*$

True.

c) $b^* (a \cup b)^* a^* = b^* (a \cup b)^*$.

True. In fact, both regular expressions are equivalent to $(a \cup b)^*$.

d) $(a^* \cup b^*a^*)^* = (b \cup a)^*$.

True.

e) $(\varepsilon \cup (a \cup b)^* b (a \cup b)^* \cup (a \cup b)^* a (a \cup b)^*) = (a \cup b)^*$.

True.

f) $(a^* \cup b)^* = (a \cup b)$.

False. The second expression can only generate two strings: a and b. The first expression can generate many others, including, for example aa.

g) $(a \cup b)^* \cup ab = (a \cup b \cup ba \cup ab)^+$.

False. The second expression cannot generate ε but the first one can.

h) $(a \cup b)\, a^* = (\varepsilon \cup b)\, a^*$.

False. The empty string ε is in the language defined by the expression on the right but it is not in the language defined by the expression on the left.

i) $(a \cup b \cup \varepsilon)^+ (a \cup b) = (a \cup b)^+$.

True.

j) $(a \cup bc)^* - (a \cup bc) = (a \cup bc)^+$.

False. $\varepsilon \in (a \cup bc)^* - (a \cup bc)$, but it is not in $(a \cup bc)^+$.

k) [Luay Nakhleh] For any regular expressions α and β, $(\alpha \cup \beta)^* = \alpha^*(\beta\alpha^*)$.

True. Any string generated by either of these expressions contains 0 or more substrings generated by β, preceeded and/or followed and/or separated by zero or more substrings generated by α.

7 Regular Gramamars

1) Show a regular grammar for each of the following languages:

   a) $\{w \in \{\texttt{a}, \texttt{b}\}^* : w \text{ ends in } \texttt{aa}\}$.

$S \rightarrow \texttt{a}S$
$S \rightarrow \texttt{b}S$
$S \rightarrow \texttt{a}T$
$T \rightarrow \texttt{a}$

   b) $\{w \in \{\texttt{a}, \texttt{b}\}^* : \text{every } \texttt{a} \text{ in } w \text{ is immediately followed by at least one } \texttt{b}\}$.

$S \rightarrow \varepsilon$
$S \rightarrow \texttt{b}S$
$S \rightarrow \texttt{a}T$
$T \rightarrow \texttt{a}$

   c) $\{w \in \{\texttt{a}, \texttt{b}\}^* : w \text{ contains no two sequential characters that are the same}\}$.

$S \rightarrow \varepsilon$
$S \rightarrow \texttt{a}A$
$S \rightarrow \texttt{b}B$
$A \rightarrow \varepsilon$
$A \rightarrow \texttt{b}B$
$B \rightarrow \varepsilon$
$B \rightarrow \texttt{a}A$

   d) $\{w \in \{\texttt{a}, \texttt{b}\}^* : |w| \geq 1$ and:
   - if $|w|$ is even, then w starts with an $\texttt{a}$, and
   - if $|w|$ is odd, then w starts with a $\texttt{b}\}$.

$S \rightarrow A \mid B$
$A \rightarrow \texttt{a}O$
$B \rightarrow \texttt{b}E$
$O \rightarrow \texttt{a} \mid \texttt{b} \mid \texttt{a}E \mid \texttt{b}E$
$E \rightarrow \varepsilon \mid \texttt{a}O \mid \texttt{b}O \mid$

2) Let L be the language described by the following regular grammar:

$S \rightarrow \varepsilon$
$S \rightarrow \texttt{0} \mid \texttt{2} \mid \texttt{4} \mid \texttt{6} \mid \texttt{8}$
$S \rightarrow \texttt{0}S \mid \texttt{2}S \mid \texttt{4}S \mid \texttt{6}S \mid \texttt{8}S$
$S \rightarrow \texttt{1}T \mid \texttt{3}T \mid \texttt{5}T \mid \texttt{7}T \mid \texttt{9}T$
$T \rightarrow \texttt{0} \mid \texttt{2} \mid \texttt{4} \mid \texttt{6} \mid \texttt{8}$
$T \rightarrow \texttt{0}S \mid \texttt{2}S \mid \texttt{4}S \mid \texttt{6}S \mid \texttt{8}S$

   a) For each of the following strings, indicate whether or not it is in L:

(i)	`46`	Yes
(ii)	`3377`	No
(iii)	`567890`	Yes
(iv)	`8787777`	No

b) Give a simple English description of *L*.

Decimal strings that contain no consecutive odd digits.

3) Let *L* be the language described by the following regular grammar:

$S \rightarrow \texttt{a}V \mid \texttt{b}W$
$V \rightarrow \texttt{a}W \mid \texttt{b}W \mid \varepsilon$
$W \rightarrow \texttt{a} \mid \texttt{b} \mid \texttt{a}V \mid \texttt{b}V$

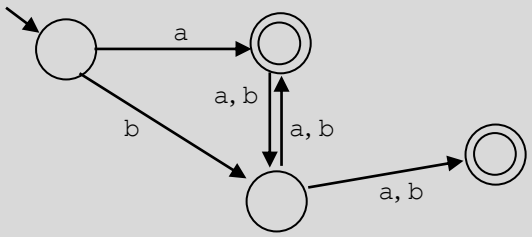
a) For each of the following strings, indicate whether or not it is in *L*:

(i)	ε	No
(ii)	aaaa	No
(iii)	bbbb	Yes
(iv)	ababa	Yes

b) Give a simple English description of *L*.

Strings over the alphabet {a, b} that either:
- Start with a and have odd length, or
- Start with b and have even length.

c) Use *grammartofsm* to construct an FSM that accepts *L*.



d) How many states are there in the mimimal FSM that accepts *L*?

3.

8 Regular and Nonregular Languages

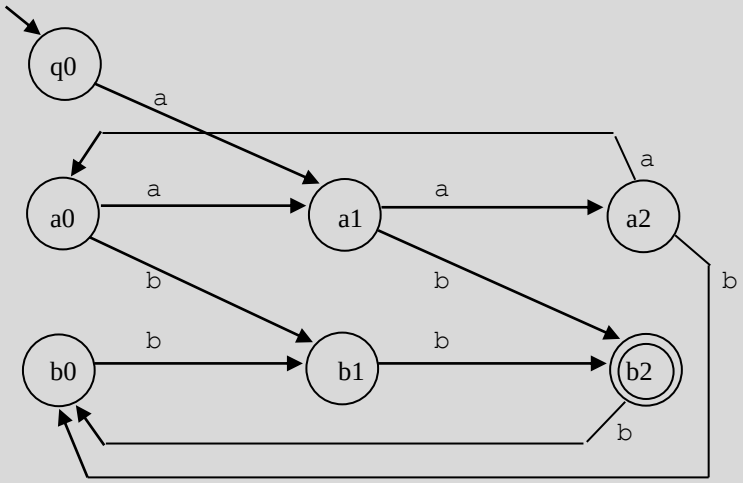
1) For each of the following languages L, state whether or not L is regular. Prove your answer.

a) $\{a^n b^n : n \leq 5\}$.

Regular. L is finite. The following regular expression defines it: $\varepsilon \cup ab \cup aabb \cup aaabbb \cup aaaabbbb \cup aaaaabbbbb$.

b) $\{a^i b^j : i, j \geq 1 \text{ and } i + j \equiv_3 2\}$.

Regular. We can build an FSM to accept L. We need the special start state since there must be at least one a. There is only one accepting state since there must also be at least one b.



c) [Luay Nakhleh] $\{a^{i-j} : 0 \leq j \leq i\}$.

Regular. It's just a*.

d) $\{w \in \{a, b, c\}^* :$ every c has an a immediately to its left and a b immediately to its right$\}$.

Regular. The following regular expression defines it: $(a \cup b \cup acb)^*$.

e) $\{w \in \{0, 1, 2\}^* :$ every 1 is immediately preceded by 0 and immediately followed by 2$\}$.

Regular. The following regular expression defines it: $(0 \cup 2 \cup 012)^*$.

f) $\{w \in \{0, 1\}^* :$ w contains exactly one pair of consecutive 0's.

Regular. L can be described by the regular expression:

$$(1 \cup 01)^* 00 (1 \cup 10)^*$$

g) $\{w \in \{a, b, c, d\}^* :$ if there are any a's in w, then there is at least one c, and if there are any b's, then there is at least on d$\}$.

Regular. Easy to build an NDFSM.

h) $\{w \in \{a, b, c\}^* : w$ contains at least two occurrences of the substring ac or w contains no more than 3 c's$\}$.

Regular. $L = L_1 \cup L_2$, where: $L_1 = \{w \in \{a, b, c\}^* : w$ contains at least two occurrences of the substring $ac\}$ and $L_2 = \{w \in \{a, b, c\}^* : w$ contains no more than 3 c's$\}$. L_1 is regular because it can be accepted by a straightforward NDFSM (must show it). L_2 is regular because it is defined by the regular expression: $(a \cup b)^*(c \cup \varepsilon)(a \cup b)^*(c \cup \varepsilon)(a \cup b)^*(c \cup \varepsilon)(a \cup b)^*$. The regular languages are closed under union, so L must also be regular.

i) $\{w \in \{a, b, c\}^* : \#_a(w)$ is even or $\#_b(w)$ is divisible by 3 or w contains at least one $c\}$.

Regular. There exists a straightforward NDFSM that accepts L. It has three branches, one of which accepts all strings w where $\#_a(w)$ is even, one of which accepts all strings w where $\#_b(w)$ is divisible by 3, and the last of which accepts all strings w where w contains at least one c. Alternatively, L can be described by the regular expression:

$$\begin{aligned}&(b \cup c)^*(a(b \cup c)^*a(b \cup c)^*)^* \\ \cup\ &(a \cup c)^*(b(a \cup c)^*b(a \cup c)^*b(a \cup c)^*)^* \\ \cup\ &(a \cup b)^*c(a \cup b \cup c)^*\end{aligned}$$

j) $\{a^n b^m : 0 \le n, 0 \le m$, and $n + m$ is even$\}$.

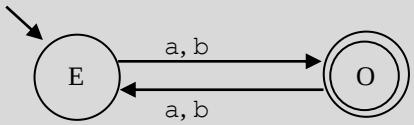
Regular. $L = \{w \in \{a^*b^* : |w|$ is even$\}$.

k) $\{0^n 1^m : n, m \ge 1\}$.

Regular. L can be described by the regular expression 00^*11^*.

l) $\{w \in \{a, b\}^* : w$ has an even number of a's and an odd number of b's or w has an odd number of a's and an even number of b's$\}$.

Regular.



m) $\{w \in \{a, b, c\}^* : (|w|$ is even$) \rightarrow (w$ contains an even number of a's$\}$.

Regular. $L = \{w \in \{a, b, c\}^* : |w|$ is odd or w contains an even number of a's$\}$. Build an FSM for $\{w \in \{a, b, c\}^* : |w|$ is odd$\}$ and one for $\{w \in \{a, b, c\}^* : w$ contains an even number of a's$\}$. The regular languages are closed under union.

n) $\{a^n b^* a^n : n \ge 0\} \cap \{b^n a^* b^n : n \ge 0\}$.

Regular. Note that:

- $\{a^n b^* a^n : n \ge 0\} = (aa)^* \cup b^*$ $\cup$ {some strings that start with a and contain both a's and b's$\}$.
- $\{b^n a^* b^n : n \ge 0\} = a^* \cup (bb)^*$ $\cup$ {some strings that start with b and contain both a's and b's$\}$.

L contains all strings that are in both of those languages. So $L = (aa)^* \cup (bb)^*$.

o) $\{w \in \{a, b\}^* : w$ contains at least one instance of the substring aabaab and one instance of the substring bb$\}$.

Regular. There exists a straightforward NDFSM that accepts L. It has two branches, one of which accepts aabaab followed (eventually) by bb and the other of which accepts bb followed by aabaab.

p) $\{xy : x, y \in \{a, b\}^*$ and $\exists z$ (z is a prefix of y and $z = as$ for some $s \in \{a, b\}^*$)$\}$.

Regular. A string w is in L iff it contains at least one a. To see why this is so, call the part of w that comes before that a x. Call the part that starts with that a y. That a can then be the beginning of z, which is thus a prefix of y. So L is defined by the following regular expression: $(a \cup b)^*a(a \cup b)^*$.

q) Let $\Sigma = \{a, b\}$. $L = \{w \in \Sigma^*$: (w contains the substring ab) $\rightarrow$ (w contains the substring ba)$\}$.

Regular. It helps to rewrite L as:

$L = \{w \in \Sigma^*$: $\neg$(w contains the substring ab) $\vee$ (w contains the substring ba)$\}$

$L =$ b^*a^* $\cup$ $(a \cup b)^*$ ba $(a \cup b)^*$

You can also do this with an FSM.

r) $\{x\, y^R$: $x, y \in \{0, 1\}^*$ and y is a prefix of $x\}$.

Regular. Since ε is a prefix of everything, $L = (0 \cup 1)^*$.

s) $\{(a \cup b)^* - (ab)^n : n \geq 1\}$.

Regular. Let L_1 be $(ab)^n$. We can describe L_1 with the regular expression $(ab)(ab)^*$. Thus L_1 must be regular. L is just the complement of L_1. Since the regular languages are closed under complement, L must be regular.

t) $L_1 \cup L_2$, where $L_1 = \{a^n b^m : n \leq m\}$ and $L_2 = \{a^n b^m : m \leq n\}$.

Regular. $L = a^*b^*$.

u) $(a \cup b)^* L_1^+ (a \cup b)^*$, where $L_1 = \{ww^R : w \in (a \cup b)^*\}$.

Regular. The key to understanding L is to observe that, while every string in L must contain at least string from L_1, $\varepsilon \in L_1$. So we can simply ignore L_1^+ and form L's strings from $(a \cup b)^*$. So $L = (a \cup b)^*$. We prove this by showing that any string s in $(a \cup b)^*$ can be generated as follows:
1. Generate x with the first $(a \cup b)^*$.
2. Choose ε from L_1^+.
3. Choose ε from $(a \cup b)^*$.

v) $(a \cup b)^* L_1^+ (a \cup b)^*$, where $L_1 = \{ww^R : w \in (a \cup b)^+\}$.

Regular. But now it's a bit more difficult because $\varepsilon \notin L_1$. But we can choose strings in L_1 that are formed by letting w be just a single character. So all that is required is that there be a double letter somewhere. $L = (a \cup b)^* (aa \cup bb) (a \cup b)^*$.

w) $\{w \in \{0, 1\}^* : \exists k \geq 0$ and w is a binary encoding (leading zeros allowed) of $2^k+1\}$.

Regular. $L = 0^* (10 \cup 10^*1)$.

x) $\{w : w$ is, for some $n \geq 1$, the decimal notation, without leading zeros) for the number $10^n\}$.

Regular. L can be described by the regular expression 100*.

y) $\{w \in \{1\}^* : w$ is the unary notation for a natural number that is a multiple of 7$\}$.

Regular. L can be described by the regular expression (1111111)*.

z) $\{w \in \{0\text{-}9\}^*: w$ is the decimal notation for a natural number that is a multiple of 7$\}$.

Regular. We can build a deterministic FSM M to accept L. M is based on the standard algorithm for long division. The states represent the remainders we have seen so far (so there are 7 of them, corresponding to 0 – 6). The start state, of course, is 0, corresponding to a remainder of 0. So is the final state. The transitions of M are as follows:

$\forall s_i \in \{0 - 6\}$ and $\forall c_j \in \{0 - 9\}$ $(\delta(s_i, c_j) = (10s_i + c_j) \pmod 7)$.

So, for example, on the input 962, M would first read 9. When you divide 7 into 9 you get 1 (which we don't care about since we don't actually care about the answer – we just care whether the remainder is 0) with a remainder of 2. So M will enter state 2. Next it reads 6. Since it is in state 2, it must divide 7 into 2*10 +6 (i. e., 26). It gets a remainder of 5, so it goes to state 5. Next it reads 2. Since it is in state 5, it must divide 7 into 5*10 + 5 (i. e., 52), producing a remainder of 3. Since 3 is not zero, we know that 862 is not divisible by 7, so M rejects.

aa) $\{w \in \{a, b\}^*: w = (a \cup b)^* a^n b^n (a \cup b)^*, n \geq 0\}$.

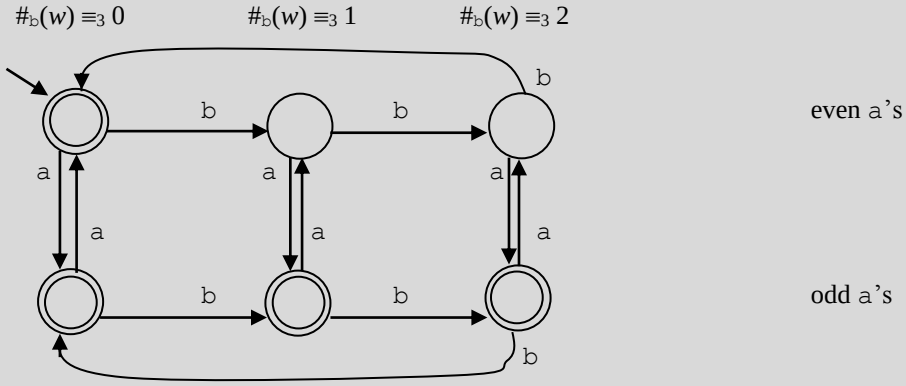
Regular. We can let n be 0. So $L = (a \cup b)^*$.

bb) $\{w \in \{a, b\}^*: (\#_a(w) \equiv_2 0) \rightarrow (\#_b(w) \equiv_3 0)\}$.

Regular. A string w is in L iff at least one of the following conditions holds:

- $\neg(\#_a(w) \equiv_2 0)$: in other words, $\#_a(w)$ is odd, or
- $(\#_b(w) \equiv_3 0)$: in other words, $\#_b(w)$ is divisible by 3.

We can build an FSM to check for these conditions:



cc) $\{w \in \{a, b\}^*: \exists y$ (y is a nonempty prefix of w and y occurs at least twice in w)$\}$. Example: aabbaab ∈ L.

Regular. $L = \{w \in \{a, b\}^*:$ if x is the first character of w then x occurs again at least once$\}$. So L is:

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

dd) $(L')^*$, where $L' = \{w \in \{a, b\}^* : \#_a(w) \neq \#_b(w)\}$.

Regular. $L = (a \cup b)^*$. To see why that is so, observe that ε, a and b are all in L'.

ee) $\{w \in \{a, b\}^* : |w|$ is a leap year in the modern Western calendar$\}$.

Regular. A year is a leap year if and only if it is divisible by 4, unless it is divisible by 100, in which case it is not a leap year, unless it is divisible by 400, in which case it is. An FSM can check these constraints.

ff) $\{w \in \{0\text{-}9\}^*$: w corresponds to the Social Security number of the oldest person ever to have a United States Social Security number$\}$.

Regular. This set is finite.

gg) Assume that we represent dates as strings of the form *mm*/*dd*/*yyyy*, where each *m*, *d*, and *y* is drawn from {0-9}. We will accept that, with this representation, we can only describe dates starting at 00/00/0000 and going up to 12/31/9999. Given this representation, let L = {strings that occur in years that are prime numbers}.

Regular. This set is finite.

hh) $\{w \in \{0\text{-}9\}^*$: all 0's precede all 1's precede all 2's, etc.$\}$.

Regular. L can be described by the following regular expression: 0*1*2*3*4*5*6*7*8*9*.

ii) PartialBal = $\{w \in \{), (\}^*$: all ('s in w occur before all)'s$\}$.

Regular. PartialBal can be accepted by a three-state FSM that stays in its initial state as long as it is reading left parens. As soon as it sees a right paren, it goes to state 2. It stays there as long as it continues to read right parens. If it sees a left paren in state 2, it goes to state 3, its dead state. States 1 and 2 are accepting. Alternatively, it can be described by the regular expression (*)*.

jj) $\{w = xyz, x, y, z \in \{a, b\}^*$ and $|x| = |y|\}$.

Regular. The key is that both x and y may be empty. So $L = (a \cup b)^*$.

kk) $\{w \in \{a, b\}^*$: exactly two prefixes of w contain one or more a's$\}$

Regular. L can be described by the regular expression b*a(b ∪ a). To see why this is true, observe that any prefixes that contain only b's don't count. As soon as an a appears in the string, the first prefix to contain it counts as one. To get a second one, there must be one more character. To prevent there being more than two, there can be no more characters.

ll) $\{rs^{2k}t$: r, s and t are in $\{a, b, c\}^*$ and $k \geq 1\}$

Regular. $L = \{a \cup b \cup c\}^*$. To see why, note that we can let both s and t be ε. Then, to produce any string w in $\{a \cup b \cup c\}^*$, we let r be w.

mm) $\{w = xy, x \in a^*b, y \in a^*b, |x| = |y|\}$.

Not regular. $L = \{a^n b a^n b, n \geq 0\}$, which we show is not regular by pumping. Let $w = a^k b a^k b$. y must occur in the first a region and be equal to a^p for some nonzero p. Let $q = 2$. The resulting string is $a^{k+p} b a^k b$, which is not in L.

nn) $\{w \in \{a, b\}^* : \#_a(w)$ is evenly divisible by 4 and $\#_b(w)$ is evenly divisible by $7\}$.

Regular. $L = L_1 \cap L_2$, where:

$L_1 = \{w \in \{a, b\}^* : \#_a(w)$ is evenly divisible by $4\}$, and

$L_2 = \{w \in \{a, b\}^* : \#_b(w)$ is evenly divisible by $7\}$

There exists a four-state DFSM that accepts L_1 and a seven-state DFSM that accepts L_2. So both L_1 and L_2 are regular. Since the regular languages are closed under intersection, L must also be regular.

oo) $\{(abc)^n a^n : n \geq 0\}$.

Not regular. It is possible to do this directly using the Pumping Theorem, but it is tedious. Instead, note that if L were regular, then L^R would also be regular. $L^R = \{a^n(cba)^n : n \geq 0\}$. We show that this is not regular by using the Pumping Theorem. Let $w = a^k(cba)^k$. y must occur in the initial a region and be equal to a^p for some nonzero p. Let $q = 0$ (i.e., pump out). The resulting string is: $a^{k-p}(cba)^k$. This string is not in L because the number of initial a's no longer equals the number of occurrences of cba.

pp) $\{w \in \{a, b\}^* : \#_a(w) = 3 \cdot \#_b(w)\}$.

Not regular, which we show by pumping. Let $w = a^{3k}b^k$. y must occur in the a region and be equal to a^p for some nonzero p. Let $q = 0$. The resulting string is $a^{3k-p}b^k$, which is not in L, since $3k\text{-}p \neq 3k$.

qq) $\{w \in \{a, b\}^* : \#_a(w) > 2 \cdot \#_b(w)\}$.

Not regular, which we show by pumping. Let $w = a^{2k+1}b^k$. y must occur in the a region and be equal to a^p for some nonzero p. Let $q = 0$. The resulting string is $a^{2k+1-p}b^k$, which is not in L, since $2k+1\text{-}p$ is not greater than $2k$.

rr) $\{w \in \{a, b\}^* :$ for each suffix x of w, $\#_a(x) \geq \#_b(x)\}$.

Not regular. We use the Pumping Theorem to show this. Let $w = b^k a^k$. $y = b^p$, for some nonzero p. Let q be 2. The resulting string is $b^{k+p}a^k$. It is a suffix of itself and the number of b's is greater than the number of a's. So it is not in L.

ss) $\{w \in \{a, b, c\}^* : \#_c(w) > \#_a(w)\}$.

Not regular. We use the Pumping Theorem to show this. Let $w = c^{k+1}a^k$. $y = c^p$, for some nonzero p. Pump out. The resulting string is $c^{k-p}a^k$. There are no longer more c's than a's. So it is not in L.

tt) $\{w \in \{a, b, c\}^*$: the number of occurrences of the substring aba in w equals the number of occurrences of the substring bab in $w\}$.

Not regular. If L were regular, then $L' = L \cap (abaa)^*c(babb)^*$ would also be regular, but it is not, as we can show using the Pumping Theorem. Let $w = (abaa)^k c(babb)^k$. y must occur in the first k characters, so it must be in the $(abaa)^k$ region. If $y \neq (abaa)^p$ for some p then pump in once. The resulting string will not be in $(abaa)^*c(babb)^*$ and so is not in L'. If $y = (abaa)^p$ for some p then pump in once. The resulting string will have more aba's than bab's and so will not be in L'.

uu) $\{w \in \{0 - 9\}^*$: the number of odd digits in w equals the number of even digits in $w\}$.

Not regular. We use the Pumping Theorem to show this. Let $w = 0^k1^k$. y must occur in the first k characters, so it must be in the 0 region. Pump in once. The number of 1's (and thus odd digits) no longer equals the number of 0's (and thus even digits).

vv) $\{w = x\mathtt{a}^n: x \in \{\mathtt{a} \cup \mathtt{b} \cup \mathtt{c}\}^*$ and $|x| = n$ and $\#_{\mathtt{b}}(x) = n/2\}$.

Not regular, which we can show using the Pumping Theorem. Let $w = \mathtt{a}^k\mathtt{b}^k\mathtt{a}^k$. y must occur in the first k characters, so it must be a^p for some nonzero p. Pump in once. If p is odd, then the resulting string is not in L since every string in L has even length. If p is even, then there is at least one b in the second half of the new string. No string in L has that property.

ww) $\{w \in \{\mathtt{a}, \mathtt{b}\}^*$: for each prefix x of w, $\#_{\mathtt{a}}(x) \geq \#_{\mathtt{b}}(x)\}$.

Not regular, which we can show using the Pumping Theorem. Let $w = \mathtt{a}^k\mathtt{b}^k$. y must occur in the first k characters, so it must be a^p for some nonzero p. Pump out once. The resulting string is not in L because it is a prefix of itself but it has more b's than a's.

xx) $\{w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* : \#_{\mathtt{a}}(w) \neq max(\#_{\mathtt{b}}(w), \#_{\mathtt{c}}(w))\}$.

Not regular. If L were regular, then $\neg L = \{w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* : \#_{\mathtt{a}}(w) = max(\#_{\mathtt{b}}(w), \#_{\mathtt{c}}(w))\}$ would also be regular. But we show that it isn't by pumping. Let $w = \mathtt{a}^k\mathtt{b}^k\mathtt{c}^k$. Note that $max(\#_{\mathtt{b}}(w), \#_{\mathtt{c}}(w)) = k$. $y = \mathtt{a}^p$, for some nonzero p. Pump out once. The number of a's in w is now less than $max(\#_{\mathtt{b}}(w), \#_{\mathtt{c}}(w))$, so this string is not in $\neg L$. Thus neither $\neg L$ nor L is regular.

yy) $\{w = st : s \in \{\mathtt{a}, \mathtt{b}\}^*$ and $t \in \{\mathtt{b}, \mathtt{c}\}^*$ and $\#_{\mathtt{b}}(s) = 2 \cdot \#_{\mathtt{a}}(s)$ and $\#_{\mathtt{c}}(t) = 3 \cdot \#_{\mathtt{b}}(t)\}$

Not regular, which we show by pumping. Let $w = \mathtt{b}^{2k}\mathtt{a}^k\mathtt{c}^{3k}\mathtt{b}^k$. y must occur in the first b region. It is $\mathtt{b}^p$, for some nonzero p. Note that, when we pump, the boundary between the s and t regions cannot move because there can be no a's in s or c's in t. Let $q = 0$ (i.e., pump out). The resulting string is $\mathtt{b}^{2k-p}\mathtt{a}^k\mathtt{c}^{3k}\mathtt{b}^k$. The s region is $\mathtt{b}^{2k-p}\mathtt{a}^k$. It doesn't have twice as many b's as a's. So this string is not in L.

zz) $\{w$ is of the form $x\#y$, where $x, y \in \{\mathtt{1}\}^+$ and $y = x+1$ when x and y are interpreted as unary numbers$\}$ (For example, 11#111 and 1111#11111 $\in L$, while 11#11, 1#111, and 1111 $\notin L$.).

Not regular. Intuitively, L isn't regular because any machine to accept it must count the 1's before the # and then compare that number to the number of 1's after the #. We can prove that this is true using the Pumping Theorem: Let $w = \mathtt{1}^k\#\mathtt{1}^{k+1}$. Since $|xy| \leq k$, y must occur in the region before the #. Thus when we pump (either in or out) we will change x but not make the corresponding change to y, so y will no longer equal $x+1$. The resulting string is thus not in L.

aaa) $\{x\#y : x, y \in \{\mathtt{0}, \mathtt{1}\}^*$, and, when viewed as binary numbers, $y = 2x\}$. For example, 100#1000 $\in L$.

Not regular. If L were regular, then $L' = L \cap \mathtt{10}^*\#\mathtt{10}^*$ would also be regular. But it is not. We use the Pumping Theorem to show this. Let $w = \mathtt{10}^k\#\mathtt{10}^{k+1}$.

- If y includes the initial 1, then set q to 0 (i.e., pump out). The resulting string will no longer start with 1, as required by L'.
- If y does not include the initial 1 then it is $\mathtt{0}^p$, for some nonzero p. Let $q = 3$ (i.e., pump in twice). The resulting string has more 0's in the initial region than in the second one and so $x > y$. So $y \neq 2x$. So the resulting string is not in L'.

bbb) $\{x\#y$: $x, y \in \{\mathtt{0}, \mathtt{1}\}^*$, when viewed as binary numbers, $x+y = 3y\}$

Not regular, which we show using the Pumping Theorem. We must start by choosing a string that is in fact in L. Let $w = \mathtt{100}^k\#\mathtt{10}^k$. Then $w \in L$ since $3y$ is $\mathtt{110}^k$. We must consider three cases for where y can fall:
$y = \mathtt{1}$ Pump out. Arithmetic is wrong. The left side is 0 but right side isn't.
$y = \mathtt{10}^*$ Pump out. Arithmetic is wrong. "
$y = \mathtt{0}^p$ pump out. Arithmetic wrong. Decreased left side but not right.

ccc) $\{a^i b^j : 0 \le i, 0 \le j$, and $i+j$ is prime$\}$.

Not regular. The regular languages are closed under letter substitution. So if L were regular, so would be $L' = L$ with every b replaced by an a. But $L' = \text{Prime}_a$, which we have already shown is not regular. So L is not regular either.

ddd) [Luay Nakhleh] $\{0^n 1^m 2^j : n, m, j \ge 0$ and $m \le max(n, j)\}$.

Not regular, which we show using the Pumping Theorem. Let $w = 1^k 2^k$. So $n = 0$, $m = j = k$, and $m = max(n, j)$, since $k = max(0, k)$. Since y must occur in the first k characters, it must be 1^p for some nonzero p. Set q to 2. The resulting string is $1^{k+p}2^k$. $max(n, j)$ has not changed. But m has grown. $k+p > max(0, k)$. So the new string is not in L.

2) Let $L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$. Answer each of the following questions and prove your answer:

a) Is L regular?

No. We proved this in Example 8.14.

b) Is L^* regular?

No, which we show using the Pumping Theorem. Note that every string in L^* also has an equal number of a's and b's. Let $w = a^k b^k$. y must occur in the first k characters, so it must be a^p for some nonzero p. Pump out once. The resulting string is not in L because it does not have an equal number of a's and b's.

3) Let $L = \{a^p : p$ is prime$\}$. Answer each of the following questions and prove your answer:

a) Is L regular?

No. We proved this in Example 8.13.

b) Is L^* regular?

Yes. $L = \{aa, aaa, aaaaa, \ldots\}$. So L^* contains ε plus every string that consists of only a's except for the single string a. L^* can thus be described the regular expression $\varepsilon \cup a^*$.

4) Let *mid* be a function on strings, defined as follows:
For any string s in some language L over Σ:

- If $|s| \le 2$ then $mid(s) = \varepsilon$.
- If $|s| > 2$ then let x and z be single characters in Σ. Then we can rewrite s as xwz for some $w \in \Sigma^*$. $mid(s) = w$.

For any language L over Σ, we can define the function $midL(L)$ as follows:
$midL(L) = \{t \in \Sigma^* : t = mid(s)$ for some $s \in L\}$

a) What is $midL(a^*ba^*)$?

a^*ba^*

b) Is FIN (the set of finite languages) closed under *midL*?

Yes. Each string in L can produce no more than one string in $midL(L)$, so $|midL(L)| \le |L|$.

c) Is INF (the set of infinite languages) closed under *midL*?

Yes. Every string s in L of length greater than 2 produces at least one string in $midL(L)$ of length $|s|$ -2. If L is infinite, then there is no upper bound on the length of its strings. Thus there is no upper bound on the length of the strings in $midL(L)$. So $midL(L)$ is infinite.

d) Are the regular languages closed under *midL*?

Yes. Note that *midL*(*L*) contains all strings that can be derived by taking some string in *L* and erasing the first and last characters. If *L* is regular, then it is accepted by some DFSM $M = (K, \Sigma, \delta, s, A)$. From *M*, we construct a new FSM M' that accepts *midL*(*L*). Initially, let M' be *M*. Create a new start state s' and a new accepting state a'. Make a' the only accepting state of M'. For every transition $((s, c), q)$ in δ, add to M' the transition $((s', \varepsilon), q)$. (So, for every first move that *M* can make on some input character, M' can make the same move without that character.) Similarly, for every transition $((q, c), a)$ in δ, where $a \in A$, add to M' the transition $((q, \varepsilon), a')$. (So, for every last move that *M* can make on some input character, M' can make the same move without that character.)

e) Are the nonregular languages closed under *midL*?

No. Let $L = \{1a^p1, p > 0 \text{ and prime}\} \cup \{0a^p0, p > 0 \text{ and not prime}\}$. *L* is not regular. But *midL*(*L*) = $\{a^p, p > 0\}$, which is regular.

5) Let *edges* be a function on languages such that:
edges(*L*) = $\{w: \exists x \in L\ (x = c_1yc_2, c_1 \in \Sigma, c_2 \in \Sigma, y \in \Sigma^*, w = c_1c_2)\}$.
Prove your answers to parts (b) - (e).

a) What is *edges*(a*ba*)?

{aa, ab, ba}

b) Is FIN (the set of finite languages) closed under *edges*?

Yes. Each string in *L* can produce no more than one string in *edges*(*L*), so $|edges(L)| \leq |L|$.

c) Is INF (the set of infinite languages) closed under *edges*?

No. Let *L* = a*. *L* is infinite. But *edges*(*L*) = {aa}, which is finite.

d) Are the regular languages closed under *edges*?

Yes. For all *L*, every string in *edges*(*L*) is of length 2. Since $|\Sigma|$ is finite, for all *L*, *edges*(*L*) is finite. Every finite language is regular.

e) Are the nonregular languages closed under *edges*?

No. By the argument in part d, for all *L*, *edges*(*L*) is regular. A specific counterexample: Let $L = \{a^nba^n : n \geq 0\}$. *L* is not regular. *edges*(*L*) = {aa}, which is regular.

6) [Luay Nakhleh] Let *min* be a function on languages such that:
min(*L*) = $\{w \in L$: no proper prefix of *x* is in *L*$\}$.
Prove or disprove each of the following claims:

a) If *L* is regular, *min*(*L*) is regular.

True. If L is regular, then it is accepted by some DFSM *M*. From *M* we construct a new DFSM *M´* that accepts *min*(*L*). *M´* is just *M´* except that we remove every outgoing edge from every accepting state.

b) If *min*(*L*) is regular, *L* is regular.

Fasle, which we show by counterexample. Let $L = \{a^mb^n : n, m > 0 \text{ and } m \geq n\}$, which isn't regular. But *min*(*L*) = {ab, aab, aaab, aaaab} = aa*b, which is regular.

7) Define the binary function • on pairs of languages with alphabet Σ, as follows:

$L_1 \bullet L_2 = \{w: \exists x,y \in \Sigma^* \ (wx \in L_1 \text{ and } wy \in L_2)\}$.

Are the regular languages closed under •? Prove your answer.

Yes. Note that $L_1 \bullet L_2 = pref(L_1) \cap pref(L_2)$. We've shown that the regular languages are closed under *pref*. So both $pref(L_1)$ and $pref(L_2)$ are regular. We've also shown that the regular languages are closed under intersection. So $pref(L_1) \cap pref(L_2)$ is regular.

8) [Luay Nakhleh] Let A and B be two regular languages over some alphabet Σ. Which of the following languages are necessarily regular? Prove your answers.

a) $L_1 = \{x : x^R \in A\}$.

Necessarily regular. The regular languages are closed under reverse.

b) $L_2 = \{x : x \in A \text{ and } x^R \in B\}$.

Necessarily regular. $L_2 = L(A) \cap L(B)^R$. The regular languages are closed under both reverse and intersection.

c) $L_3 = \{x : x \in A \text{ and } x^R \notin B\}$.

Necessarily regular. $L_3 = L(A) \cap \neg L(B)^R$. The regular languages are closed under reverse, complement, and intersection.

d) $L_4 = \{x : x \in A \text{ and } x = x^R\}$.

Not necessarily regular. Let Σ = {a, b}. Let $A = \Sigma^*$. Then $L_4 = \{x = x^R : x \in \{a, b\}^*\}$, which we can show, using the Pumping Theorem, isn't regular. Let $w = a^k b a^k$. Then y is a^p, for some nonzero p, and it must occur in the initial a region. Pump in once. The resulting string is $a^{k+p} b a^k$. It is not in L because there are $k+p$ a's before the b reading from the left but only k a's before the b, reading from the right.

9) [Luay Nakhleh] Let Σ be an alphabet containing symbols a and b and possibly others. Then:

- For any string $y \in \Sigma^*$, let $y_{a \to b}$ be y with every occurrence of a replaced by b.
- For any language $L \subseteq \Sigma^*$, let $L_{a \to b}$ be the language $\{y_{a \to b} : y \in L\}$.

For example, if $L = \{aa, ab, abc\}$, then $L_{a \to b} = \{bb, bbc\}$.

Prove or disprove each of the following assertions:

a) If L is regular, then $L_{a \to b}$ is regular.

This claim is true. If L is regular, then there exists some DFSM M that accepts it. We modify M in the following way so that it accepts $L_{a \to b}$: Change every transition labeled a to one labeled b.

b) If $L_{a \to b}$ is regular, then L is regular.

This claim is false, which we prove with a counterexample. Let $L = \{a^n b^n : n \geq 0\}$. $L_{a \to b}(L) = \{b^n b^n : n \geq 0\} = (bb)^*$. So $L_{a \to b}(L)$ is regular, but L is not.

10) [Luay Nakhleh] For any two languages L_1 and L_2, define:

- *split*(L_1, L_2) = {x_1yx_2 : x_1, $x_2 \in L_1$ and $y \in L_2$}.
- *symsplit*(L_1, L_2) = {x_1yx_2 : x_1, $x_2 \in L_1$ and $y \in L_2$ and $0 \leq |x_1| - |x_2| \leq 1$}.

a) Let L_1 = 0* and let L_2 = 1*. Give precise descriptions of *split*(L_1, L_2) and *symsplit*(L_1, L_2).

split(L_1, L_2) is 0*1*0*. *symsplit*(L_1, L_2) is $\{0^n1^*0^n : n \geq 0\} \cup \{0^{n+1}1^*0^n : n \geq 0\}$.

b) Are the regular languages closed under *split*? Prove your answer.

This claim is true. *split*(L_1, L_2) is simply $L_1L_2L_1$ and the regular languages are closed under concatenation.

c) Are the regular languages closed under *symsplit*? Prove your answer.

This claim is False The languages described in part (a) give a counterexample. Both L_1 and L_2 are regular, but *symsplit*(L_1, L_2) is not.

9 Decision Procedures for Regular Languages

1) Define a decision procedure for each of the following. Argue that each of your decision procedures gives the correct answer and terminates.
   a) Given two regular expressions α_1 and α_2, are there at least three strings that can be generated by both α_1 and α_2?

1. From α_1 and α_2 construct M_1 and M_2 so that $L(\alpha_1) = L(M_1)$ and $L(\alpha_2) = L(M_2)$.
2. Construct M^* to accept $L(M_1) \cap L(M_2)$.
3. Apply *ndfsmtodfsm* to M^* to construct M^{**}.
4. Let Σ be the alphabet of M^{**} and let k be the number of states in M^{**}.
5. Run all strings in Σ^* of length $< k$ through M^{**}, keeping track of how many are accepted.
6. If none were accepted, halt and return *False*.
7. If 3 or more strings were accepted, halt and return *True*.
8. If $L(M^{**})$ is infinite, return *True*, else return *False*.

   b) Given two regular expressions, α and β, and a string w, is $w \in L(\alpha) \cap L(\beta)$?

1. From α, build FSM M_α, such that $L(M_\alpha) = L(\alpha)$.
2. From β, build FSM M_β, such that $L(M_\beta) = L(\beta)$.
3. Build M_F, such that $L(M_F) = L(M_\alpha) \cap L(M_\beta)$.
4. Run M_F on w. If it accepts, return *True*. Else return *False*.

   c) Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let α be a regular expression. Does the language generated by α contain an infinite number of strings that start with `a`?

1. Use *regextofsm*(E) to build an FSM M such that $L(M) = L(E)$.
2. Build M^* such that $L(M^*) = \mathtt{a}(\mathtt{a} \cup \mathtt{b})^*$. In other words, all strings over Σ that start with `a`.
3. Build M^{**} such that $L(M^*) = L(M) \cap L(M^*)$. In other words, M^{**} accepts all strings that are accepted by M and that start with `a`.
4. If $L(M^{**})$ is infinite, then return *True*, else return *False*.

   d) Given two regular grammars, G_1 and G_2, is $L(G_1) \cap L(G_2)$ empty?

1. Use *grammartofsm*(G_1) to build an FSM M_1 such that $L(M_1) = L(G_1)$.
2. Use *grammartofsm*(G_2) to build an FSM M_2 such that $L(M_2) = L(G_2)$.
3. Build M^* such that $L(M^*) = L(M_1) \cap L(M_2)$.
4. Return *emptyFSM*(M^*).

   e) Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Given two nondeterministic FSMs M_1 and M_2, does $L(M_1)L(M_2)$ contain exactly two strings?"

1. From M_1 and M_2, construct a new FSM M_3 that accepts $L(M_1)L(M_2)$.
2. Let k be the number of states in M_3.
3. Run M_3 on every string in Σ^* of length $< k$.
4. If any number except 2 of them were accepted, return False.
5. If M_3 accepts an infinite number of strings, return False. Else return True.

f) Let Σ = {a, b}. Given a regular grammar G, are all strings in $L(G)$ of even length?

1. Use *grammartofsm*(G) to build an FSM M such that $L(M) = L(G)$.
2. Build M^* such that $L(M^*) = ((a \cup b)(a \cup b))^*(a \cup b)$. In other words, all odd length strings over Σ.
3. Build M^{**} such that $L(M^{**}) = L(M) \cap L(M^*)$. In other words, M^{**} accepts all odd length strings accepted by M.
4. If $L(M^{**})$ is empty, then return *True*, else return *False*.

g) Given an FSM M, does M accept any binary strings that end in 0?

Here's the "analyze M as a directed graph" approach:

1. Since we want only to consider binary strings, remove from M all transitions with labels other than 0 or 1.
2. Make M deterministic.
3. Minimize M. This step removes all unreachable states.
4. If there is a transition $((p, 0), q)$, for some accepting state q, return *True*. Else return *False*.

It can also be done by simulation. To do this, examine M and count its states. Call that count k. Then it is necessary to try all binary strings that end in 0 of length up to k-1.

h) Given an FSM M, does $L(M)$ contain at least one string that starts with ab?

1. Build an FSM M^* that accepts the language $(ab)(a \cup b)^*$.
2. Build an FSM M^{**} that accepts $L(M) \cap L(M^*)$.
3. If $L(M^{**})$ is empty, return *False*, else return *True*.

i) Given an FSM M, does M accept precisely one string?

1. Examine M and count its states. Call that count k.
2. Lexicographically enumerate the strings in Σ_M^* of length less than or equal to $2k$.
3. If exactly one string is accepted, return *True*. Else return *False*.

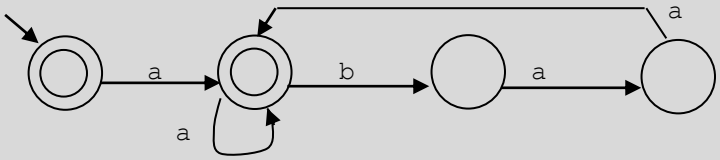
It is sufficient to try only strings up to length $2k$ because: By the argument that we used in the proof of the Pumping Theorem, if M accepts any string of length k or more, it must go through a loop to do so. By trying all strings up to length k-1, we find out whether M accepts anything at all. If it accepts no such "short" strings, then it can't accept any longer ones either since those longer ones would be able to be pumped down to shorter ones that would also have to be accepted. So, if it accepts none, our procedure will correctly return *False*. And, if it accepts more than one such short string, our procedure will also return *False*. We check strings of length between k and $2k$ to see whether M can accept by going through a loop. If we find even one string in this range (and thus at least two total), M accepts an infinite number of strings. Our procedure will return *False*.

10 Summary and References

1) Let $L = \{w \in \{a, b\}^* :$ every b in w has an a immediately to its left and two a's immediately to its right$\}$. For example, the following strings are in L: a, aaa, ε, abaa, aabaa, abaabaa. The following strings are not in L: b, bb, ababab.
   a) Show a regular expression that describes L.

   $a^* \cup a^* (abaa^*)^* a$ or $\varepsilon \cup a^* (abaa^*)^* a$ or $\varepsilon \cup a^* (aba \cup a)^* a$

   b) Show a DFSM that accepts L.



   c) What are the equivalence classes of $\approx_L$?

   [ε]
   [L - ε]
   [all strings of the form $x$ab, where $x \in L$]
   [all strings of the form $x$aba, where $x \in L$]
   [everything else, i.e., strings that can never become in no matter what comes next]

   d) Show a regular grammar that generates L.

   $S \rightarrow \varepsilon \mid aT$
   $T \rightarrow \varepsilon \mid aT \mid bW$
   $W \rightarrow aX$
   $X \rightarrow aT$

2) True or false:
   a) [Luay Nakhleh] If L^* is regular, then L is regular.

   False. Let $L = \{a^p : p$ is prime$\}$. $L^* = \varepsilon \cup aaa^*$. L^* is regular, L is not.

   b) If M_1 is a nondeterministic FSM with k states, then there may exist a deterministic FSM M_2 with fewer than k states such that $L(M_1) = L(M_2)$.

   True. M_1 may have any number of redundant states.

   c) There is a countable number of regular languages over the alphabet $\Sigma = \{a, b\}$.

   True. The number of regular languages over Σ is at least countably infinite because it includes all of {a}, {aa}, {aaa}, {aaaa}, … There can be no more than a countably infinite number of regular languages over Σ since every regular language can be described by some regular expression. The number of regular expressions, given the alphabet Σ, is countably infinite because it is possible to enumerate them lexicographically.

   d) Every regular language can be accepted by some FSM with exactly two states.

   False. There is a finite number of such FSMs and an infinite number of regular languages, so there aren't enough such FSMs.

e) The finite languages are closed under concatenation with the regular languages.

False. $L_1 = \{\varepsilon\}$ is finite. $L_2 = \mathtt{a}^*$ is regular. $L_1 L_2 = \mathtt{a}^*$ is not finite.

f) The finite languages are closed under intersection with the regular languages.

True. For any two languages L_1 and L_2, $|L_1 \cap L_2| \leq min(|L_1|, |L_2|)$.

g) [Luay Nakhleh] Any infinite regular language properly contains another infinite regular language.

True. Let L be any infinite regular language and let w be some string in L. Then $L \cap \neg\{w\}$ is infinite, regular, and a proper subset of L.

h) [Luay Nakhleh] The nonregular languages are closed under complement.

True. Complement is self-dual (i.e., the complement of the complement is itself). So the complement of a nonregular language cannot be regular. If it were, its complement (i.e., the original language) would also be regular since the regular languages are closed under complement.

i) If $L = L_1 L_2$ and L is regular then L_1 and L_2 must be regular.

False. Let $L_1 = \{\mathtt{a}^p : p \text{ is prime}\}$. Let $L_2 = \mathtt{a}^*$. $L = \{\mathtt{a}^p : p \geq 2\}$, which is regular. But L_1 is not regular.

j) $(\neg(\neg L)$ is regular$) \rightarrow (L$ is regular).

True, since $\neg(\neg L) = L$.

k) (L_1 - L_2 is regular) $\rightarrow$ (L_1 is regular).

False. A^nB^n - $A^nB^n = \varnothing$.

l) If $L_1 \cup L_2$ is regular, then both L_1 and L_2 must also be regular.

False. Let $L_1 = \{\mathtt{a}^n\mathtt{b}^m : 0 \leq n \leq m\}$. Let $L_2 = \{\mathtt{a}^n\mathtt{b}^m : 0 \leq m \leq n\}$. $L_1 \cup L_2 = \mathtt{a}^*\mathtt{b}^*$, which is regular. But neither L_1 nor L_2 is regular.

m) (L^R is regular) $\rightarrow$ (L is regular).

True, since the regular languages are closed under reverse and reverse is a self inverse. So, if L^R is regular then so is $(L^R)^R$, which is L.

n) For any language L, $L \cup \{\mathtt{a}^n\mathtt{b}^n : n \geq 0\}$ must not regular.

False. Let $L = \mathtt{a}^*\mathtt{b}^*$. $L \cup \{\mathtt{a}^n\mathtt{b}^n : n \geq 0\} = \mathtt{a}^*\mathtt{b}^*$, which is regular.

o) Given any language L, it cannot be true that L - $\{\mathtt{a}^n\mathtt{b}^n : n \geq 0\}$ is regular.

False. Let $L = \{\mathtt{a}^n\mathtt{b}^n : n \geq 0\}$. L - $\{\mathtt{a}^n\mathtt{b}^n : n \geq 0\} = \varnothing$, which is regular.

p) The finite languages are closed under Kleene star.

False. Counterexample:
Let $L_1 = \{\mathtt{a}\}$. L_1 is finite.
Let $L = L_1^*$. $L = \mathtt{a}^*$, which is infinite.

q) Every subset of a nonregular language is nonregular.

False. Let $L = \{a^n b^n : n \geq 0\}$. L is not regular. But $\{\varepsilon\}$ is a regular subset of L.

r) The finite languages are closed under *maxstring* = $\{w: w \in L$ and $\forall z \in \Sigma^* (z \neq \varepsilon \rightarrow wz \notin L)\}$.

True. Every string in *maxstring*(L) is also in L so there cannot be more strings in *maxstring*(L) than there are in L.

s) The infinite languages are closed under *maxstring*.

False. Let $L = a^*$. L is infinite. But *maxstring*(L) = $\varnothing$.

t) If *chop*(L) is regular then L is regular.

False. Counterexample:
Let $L = \{a^n b a^n, n \geq 0\}$. L is not regular. But *chop*(L) = $\{a^n, n$ is even$\}$, which is regular.

u) Define the class IR to be the class of languages that are both infinite and regular. If L_1 and L_2 are in IR and $L_1 = L_2 \cap L_3$, then L_3 must also be in IR.

False. Let $L_2 = \{a^n b^* : n$ is even$\}$. Let $L_3 = \{a^n b^* : n$ is prime$\}$. Then $L_1 = aab^*$. All three of these languages are infinite. But, while L_1 and L_2 are regular and thus in IR, L_3 is not regular and thus is not in IR.

v) If $L_1 \subseteq L_2$ and L_2 is regular, then L_1 must be regular.

False. Let $L_2 = a^*$. Let $L_1 = \{a^n : n$ is prime$\}$. Then $L_1 \subseteq L_2$ and L_2 is regular, but L_1 is not.

w) [Luay Nakhleh] Let L be such that, for each $w \in L$, there exists some DFSM that accepts w. Then L must be regular.

False. Given any individual string w, there is a simple DFSM that accepts it. But there's only a DFSM that accepts L (thus making L regular) if there's a single DFSM that accepts all the strings in L. We are not told that that is true. If this claim were true, every language would be regular.

x) Let $G = \{S \rightarrow aS, S \rightarrow bS, S \rightarrow a, S \rightarrow b\}$. $L(G)$ is regular.

True. G is a regular grammar.

y) Let $G = \{S \rightarrow aSa, S \rightarrow aSb, S \rightarrow bSa, S \rightarrow bSb, S \rightarrow a, S \rightarrow b\}$. $L(G)$ is regular.

True. G is not a regular grammar. But it defines a regular language, namely the set of odd length strings of a's and b's.

z) Let $G = \{S \rightarrow SS, S \rightarrow aSa, S \rightarrow bSb, S \rightarrow bSa, S \rightarrow aSb, S \rightarrow \varepsilon\}$. $L(G)$ is regular.

True. G is not a regular grammar. But it defines a regular language, namely the set of even length strings of a's and b's.

aa) Let $G = \{S \rightarrow SS, S \rightarrow bSb, S \rightarrow bSa, S \rightarrow aSb, S \rightarrow \varepsilon\}$. $L(G)$ is regular.

False. $L(G) = \{w \in \{a, b\}^* : \#_b(w) = \#_a(w) + 2k$, for some nonnegative integer $k\}$, which can easily be shown, using the Pumping Theorem, not to be regular.

bb) If $L(G)$ is finite, then G is a regular grammar.

False. Let $G = \{S \rightarrow \texttt{aaa}\}$. $L(G) = \{\texttt{aaa}\}$ is finite and thus regular, but G does not satisfy the constraints of a regular grammar.

cc) If $L(G)$ is infinite, then G is not a regular grammar.

False. Let $G = \{S \rightarrow \texttt{a}S, S \rightarrow \texttt{a}\}$. $L(G) = \texttt{a}^*$, which is infinite, but G is a regular grammar.

Part III: Context-Free Languages and Pushdown Automata

11 Context-Free Grammars

1) Show a context-free grammar for each of the following languages L:
   a) $\{a^i b^j : j = 4i + 2\}$.

$S \rightarrow aSbbbb \mid bb$

   b) $\{a^i b^j c^k, i > k, 0 \le j < 3, k \ge 0\}$.

$S \rightarrow aS$
$S \rightarrow aSc$
$S \rightarrow aB$
$B \rightarrow \varepsilon$
$B \rightarrow b$
$B \rightarrow bb$

   c) $\{w \in (L_1L_2)^*\}$, where $L_1 = \{ww^R : w \in \{0, 1\}^*\}$ and $L_2 = \{ww^R : w \in \{a, b\}^*\}$.

$S \rightarrow L_1L_2\ S \mid \varepsilon$
$L_1 \rightarrow 0\ L_1\ 1 \mid \varepsilon$
$L_2 \rightarrow a\ L_2\ b \mid \varepsilon$

   d) $\{w \in \{a, b\}^* : w$ contains twice as many b's as a's$\}$.

$S \rightarrow SaSbSbS$
$S \rightarrow SbSaSbS$
$S \rightarrow SbSbSaS$
$S \rightarrow \varepsilon$

   e) $\{w \in \{a, b\}^* :$ each even length prefix of w contains at least twice as many a's as b's$\}$.

If w is otherwise in L and its length is even, then, once, we can tack a b onto the right of it and it will still be in L since adding the b adds no new even length prefixes. So, as we're generating w, we need to keep track of whether its length is even or odd so far. E will generate even length strings. O will generate odd length ones.

$S \rightarrow Eb \mid E \mid O$
$E \rightarrow EE \mid OO$
$O \rightarrow OE \mid EO$
$E \rightarrow Oa \mid aO$
$O \rightarrow Ea \mid aE$
$E \rightarrow aaOb \mid aaOa$
$O \rightarrow aaEb \mid aaEa$
$E \rightarrow \varepsilon$

2) Let $L = \{a^m b^{2n} c^{3n} d^p : p > m$, and $m, n \ge 1\}$.
   a) What is the shortest string in L?

abbcccdd

b) Write a context-free grammar to generate L.

Two solutions:
$S \rightarrow$ aXdd, $X \rightarrow X$d, $X \rightarrow$ aXd, $X \rightarrow$ bb$Y$ccc, $Y \rightarrow$ bb$Y$ccc, $Y \rightarrow \varepsilon$, or
$S \rightarrow$ aSd, $S \rightarrow S$d, $S \rightarrow$ aMdd, $M \rightarrow$ bbccc, $M \rightarrow$ bb$M$ccc)

3) For each of the following languages L, state whether L is regular or context-free but not regular. If L is regular, show a regular grammar for it. Otherwise, show a context-free grammar.

a) $\{w \in \{\text{a}, \text{b}\}^* : w = x\,(\text{aa})^*\,x^R$, for some string $x \in \{\text{a}, \text{b}\}^*\}$

Context-free but not regular.
$S \rightarrow$ aSa
$S \rightarrow$ bSb
$S \rightarrow \varepsilon$

b) $\{w \in \{\text{a}, \text{b}\}^* : w$ does not contain any substring of three consecutive a's$\}$

Regular.
$S \rightarrow \varepsilon$
$S \rightarrow$ bS
$S \rightarrow$ aA /* After an a, there can only be one more.
$A \rightarrow \varepsilon$
$A \rightarrow$ bS
$A \rightarrow$ aB /* The second a, so need to end or generate a b.
$B \rightarrow \varepsilon$
$B \rightarrow$ bS

4) Let $L = \{w \in \{\text{a}, \text{b}\}^* :$ every suffix of w has at least as many a's as b's$\}$.

a) Show the first five strings in a lexicographic enumeration of L.

ε, a, aa, ba, aaa

b) Show a context-free grammar that generates L.

$S \rightarrow S$a | bSa | SS | ε

c) Is the grammar you wrote for part b ambiguous? Prove your answer.

Our grammar is very ambiguous. An example of a string with several derivations is bbaaaa.

d) Show an invariant that can be used to prove that your grammar generates no strings that are not in L. Show that your invariant is true when the working string is S and that it is maintained by every rule in your grammar.

Let *st* be the working string. Then I = "every suffix of *st* has at least as many a's as b's".

If *st* = S, then there are 0 a's and b's, so I holds.

$S \rightarrow S$a: increases only the number of a's. So if I hold before it is applied, it still does.
$S \rightarrow$ bSa: does not change the relationship between the number of a's and the number of b's.
$S \rightarrow SS$: does not change the relationship between the number of a's and the number of b's.
$S \rightarrow \varepsilon$: does not change the relationship between the number of a's and the number of b's.

5) Let L be the language generated by the following grammar G:

$S \rightarrow 1S2$
$S \rightarrow 1S2S$
$S \rightarrow 12$

a) Show the first five strings in a lexicographic enumeration of L.

12, 1122, 111222, 112212, 11112222

b) Prove that L is not regular.

Notice that, for any string s in L, $\#_1(s) = \#_2(s)$. We use the Pumping Theorem to show that L is not regular. Let $w = 1^k2^k$. Then y must be 1^p for some nonzero p. Let $q = 2$. The resulting string will have more 1's than 2's. It is thus not in L.

6) Let $L = \{a^ib^jc^k : i < j \text{ or } j < k\}$.

a) Show a context-free grammar that generates L.

$S \rightarrow I \mid K$
$I \rightarrow XC$ /* $i < j$
$X \rightarrow aXb \mid Xb \mid b$
$C \rightarrow cC \mid \varepsilon$
$K \rightarrow AY$ /* $j < k$
$Y \rightarrow bYc \mid Yc \mid c$
$A \rightarrow aA \mid \varepsilon$

b) Show that your grammar is ambiguous by showing one example of a string in L that has two parse trees. Show the trees.

Any string of the form $a^ib^jc^k : i < j < k$ will have two parse trees. So, for example, bcc does.

7) Consider the grammar of English that is given in Example 11.6. Add to it the following rules:

$NP \rightarrow \varepsilon$
$VP \rightarrow V\ ADV$
$ADJ \rightarrow$ nesting | green
$ADV \rightarrow$ gallantly
$N \rightarrow$ birds | popsorn | murals
$V \rightarrow$ paint | paints | swim | swims

Using this augmented grammar, show a parse tree for each of the following sentences:

a) Swim gallantly.
b) Birds paint.
c) Birds paint murals.
d) Green popcorn swims.

8) Consider the following context-free grammar *G*:

$S \rightarrow T\#T$
$T \rightarrow ABA$
$T \rightarrow C$
$A \rightarrow \texttt{a}A$
$A \rightarrow \varepsilon$
$B \rightarrow \texttt{b}B$
$B \rightarrow \varepsilon$
$C \rightarrow \texttt{c}C$
$C \rightarrow \texttt{c}$

a) Show the leftmost derivation of the string `ab#cc` produced by *G*.

$S \Rightarrow T\#T \Rightarrow ABA\#T \Rightarrow \texttt{a}ABA\#T \Rightarrow \texttt{a}BA\#T \Rightarrow \texttt{ab}BA\#T \Rightarrow \texttt{ab}A\#T \Rightarrow \texttt{ab}\#T \Rightarrow \texttt{ab}\#C \Rightarrow$
$\texttt{ab}\#\texttt{c}C \Rightarrow \texttt{ab}\#\texttt{cc}$

b) *G* is ambiguous. Prove this claim by showing a string that has at least two parse trees. Show at least two of the trees.

`aa#c`

c) Convert G to Chomsky normal form. You must use the procedure described in class (and show your work). It is not sufficient simply to write down the final result.

The nullable variables are: B, A, T.

$S \to T\#T$
$S \to T\#$
$S \to \#T$
$S \to \#$
$T \to ABA$
$T \to BA$
$T \to AA$
$T \to AB$
$T \to A$
$T \to B$
$T \to C$
$A \to \texttt{a}A$
$A \to \texttt{a}$
$B \to \texttt{b}B$
$B \to \texttt{b}$
$C \to \texttt{c}C$
$C \to \texttt{c}$

Removing unit productions:

$S \to T\#T$
$S \to T\#$
$S \to \#T$
$S \to \#$
$T \to ABA$
$T \to BA$
$T \to AA$
$T \to AB$
~~$T \to A$~~ $T \to \texttt{a}A \mid \texttt{a}$
~~$T \to B$~~ $T \to \texttt{b}B \mid \texttt{b}$
~~$T \to C$~~ $T \to \texttt{c}C \mid \texttt{c}$
$A \to \texttt{a}A$
$A \to \texttt{a}$
$B \to \texttt{b}B$
$B \to \texttt{b}$
$C \to \texttt{c}C$
$C \to \texttt{c}$

$S \to TX_{\#}T$

$S \to TX_{\#}$
$S \to X_{\#}T$
$S \to \#$
$T \to ABA$

$T \to BA$
$T \to AA$
$T \to AB$
$T \to X_aA \mid \texttt{a}$
$T \to X_bB \mid \texttt{b}$
$T \to X_cC \mid \texttt{c}$
$A \to X_aA$
$A \to \texttt{a}$
$B \to X_bB$
$B \to \texttt{b}$
$C \to X_cC$
$C \to \texttt{c}$
$X_a \to \texttt{a}$
$X_b \to \texttt{b}$
$X_c \to \texttt{c}$
$X_{\#} \to \#$

$S \to TX_1$
$X_1 \to X_{\#}T$
$S \to TX_{\#}$
$S \to X_{\#}T$
$S \to \#$
$T \to AX_2$
$X_2 \to BA$
$T \to BA$
$T \to AA$
$T \to AB$
$T \to X_aA \mid \texttt{a}$
$T \to X_bB \mid \texttt{b}$
$T \to X_cC \mid \texttt{c}$
$A \to X_aA$
$A \to \texttt{a}$
$B \to X_bB$
$B \to \texttt{b}$
$C \to X_cC$
$C \to \texttt{c}$
$X_a \to \texttt{a}$
$X_b \to \texttt{b}$
$X_c \to \texttt{c}$
$X_{\#} \to \#$

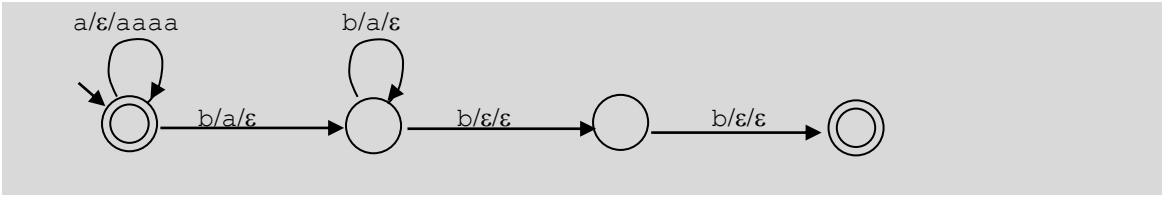
9) Convert each of the following grammars to Chomsky Normal Form:

a) $S \rightarrow A \mid B$
$A \rightarrow \mathtt{a}A \mid \mathtt{a}C$
$B \rightarrow \mathtt{b}B \mid \mathtt{b}C$
$C \rightarrow \mathtt{pqr}W$
$W \rightarrow TV$
$T \rightarrow t \mid \varepsilon$
$V \rightarrow v \mid \varepsilon$

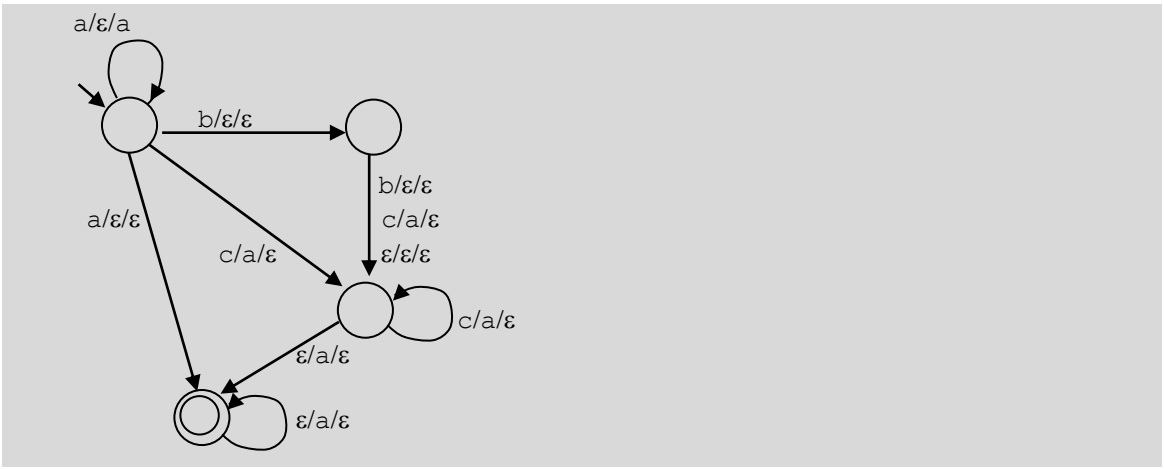
12 Pushdown Automata

1) Build a PDA to accept each of the following languages L:

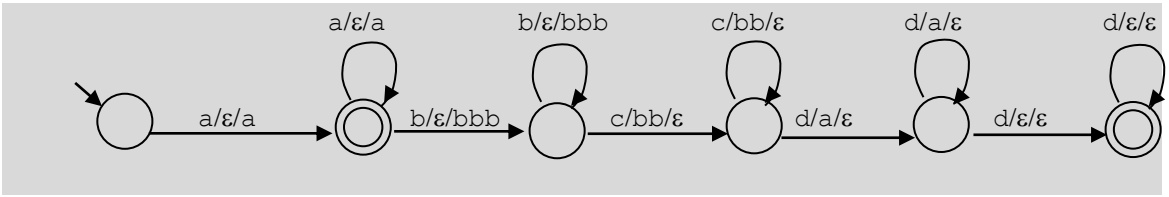
a) $\{a^i b^j : j = 4i + 2\}$.



b) $\{a^i b^j c^k, i > k, 0 \le j < 3, k \ge 0\}$



c) $\{a^m b^{2n} c^{3n} d^p : p > m$, and $m, n \ge 1\}$.



d) $\{w \in \{0, 1\}^* :$ every 0 has a matching 1 somewhere$\}$.

$M = (\{1, 2, 3\}, \{0, 1\}, \{0, 1, \#\}, \Delta, 1, \{3\})$, where $\Delta =$

{ ((1, ε, ε), (2, #)),
((2, 1, 1), (2, 11)),
((2, 1, 0), (2, ε)),
((2, 1, #), (2, 1#)),
((2, 0, 0), (2, 00)),
((2, 0, 1), (2, ε)),
((2, 0, #), (2, 0#)),
((2, ε, #), (3, ε))
((2, ε, 1), (3, ε)),
((3, ε, 1), (3, ε)),
((3, ε, #), (3, ε)), }

2) Consider the following (highly ambiguous) grammar *G* for Boolean expressions:

$E \rightarrow \neg E$
$E \rightarrow E \vee E$
$E \rightarrow E \wedge E$
$E \rightarrow E \rightarrow E$
$E \rightarrow (E)$
$E \rightarrow \forall$ `id` E
$E \rightarrow \exists$ `id` E
$E \rightarrow$ `id`

a) Show the PDA that *cfgtoPDAbottomup* will build on input *G*.

$M = (\{p, q\}, \{\texttt{id}, \neg, \vee, \wedge, \rightarrow, (,), \forall, \exists\}, \{E, \texttt{id}, \neg, \vee, \wedge, \rightarrow, (,), \forall, \exists\}, \Delta, p, \{q\})$, where $\Delta =$
{ $((p, \texttt{id}, \varepsilon), (p, \texttt{id}))$,
$((p, \neg, \varepsilon), (p, \neg))$,
$((p, \vee, \varepsilon), (p, \vee))$,
$((p, \wedge, \varepsilon), (p, \wedge))$,
$((p, \rightarrow, \varepsilon), (p, \rightarrow))$,
$((p, (, \varepsilon), (p, ()))$,
$((p,), \varepsilon), (p,))$,
$((p, \forall, \varepsilon), (p, \forall))$,
$((p, \exists, \varepsilon), (p, \exists))$,
$((p, \varepsilon, E\neg), (p, E))$,
$((p, \varepsilon, E \vee E), (p, E))$,
$((p, \varepsilon, E \wedge E), (p, E))$,
$((p, \varepsilon, E \rightarrow E), (p, E))$,
$((p, \varepsilon, (E)), (p, E))$,
$((p, \varepsilon, E$ `id` $\forall), (p, E))$,
$((p, \varepsilon, E$ `id` $\exists), (p, E))$,
$((p, \varepsilon,$ `id`$), (p, E))$,
$((p, \varepsilon, E), (q, \varepsilon))$ }

b) Trace the execution of one accepting path of your PDA on the input $\forall$ `id` $(\neg$`id` $\rightarrow ($`id` $\vee$ `id`$))$

3) Let *L* be the language of Boolean queries. A primitive query is just `id = id` or it is `id` $\neq$ `id`. Then we can build Boolean queries from primitive queries using the logical connectors $\wedge$, $\vee$, and $\neg$. Parentheses are also allowed. Here are some example grammatical queries:

`(id = id)` $\wedge \neg$ `(id` $\neq$ `id)`
`(id = id)` $\vee$ `(id = id` $\wedge$ `id = id)`

Remember that we're just using `id` as a shorthand here for any identifier, which could be a field name or a constant such as 3. Clearly these queries don't make much sense if you take `id` as a literal.

a) Show a context free grammar that generates *L*.

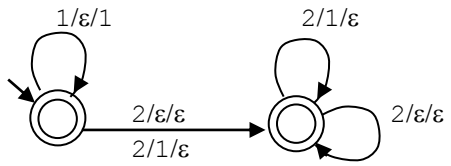
$S \rightarrow P$
$S \rightarrow \neg S$
$S \rightarrow S \wedge S$
$S \rightarrow S \vee S$
$S \rightarrow (S)$
$P \rightarrow$ `id` = `id`
$P \rightarrow$ `id` $\neq$ `id`

b) Show a PDA that accepts L.

It is straightforward to build one from the grammar shown above by using either *cfgtoPDAtopdown* or *cfgtoPDAbottomup*.

c) We could augment L in various ways. For example, we could allow wildcards. We could provide the ability to required that a value be an element of some set. Choose a useful feature and augment your grammar from part (a) to allow it.

4) Consider the following PDA M:



a) Give a concise description of $L(M)$.

$\{1^n2^m : 0 \le n \le m\}$

b) Is M deterministic? Justify your answer.

No. Whenever there is a 1 on the stack and the input symbol is 2, the two transitions from the start state to the other state compete with each other.

c) Is $L(M)$ deterministic context-free? Justify your answer.

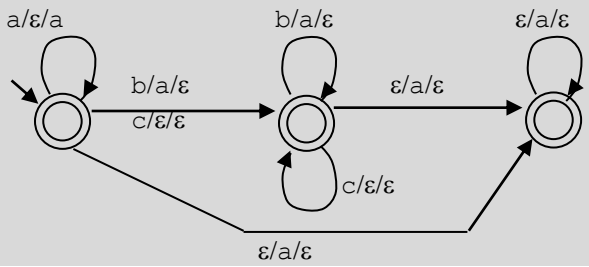
Yes. There exists a deterministic PDA that accepts $L(M)$. It works similarly to the way M works except that, before it begins reading input, it pushes a marker # onto the bottom of the stack. Then it only takes the 2 transitions that don't pop a 1 if the stack contains no 1's.

5) Let $L = \{a^n x : x \in \{b, c\}^* \text{ and } \#_b(x) \le n\}$.

a) Show a context free grammar that generates L.

$S \rightarrow aS$
$S \rightarrow aSb$
$S \rightarrow Sc$
$S \rightarrow \varepsilon$

b) Show a natural PDA that accepts L.

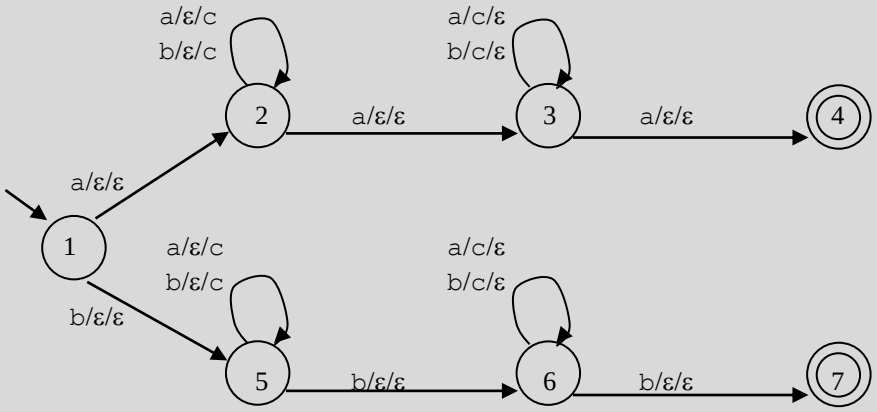


6) Let $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* :$ the first, middle, and last characters of w are identical$\}$.

a) Show a context-free grammar that generates L.

$S \rightarrow A \mid B$
$A \rightarrow \mathtt{a}\ M_A\ \mathtt{a}$
$M_A \rightarrow CM_AC \mid \mathtt{a}$
$B \rightarrow \mathtt{b}\ M_B\ \mathtt{b}$
$M_B \rightarrow CM_BC \mid \mathtt{b}$
$C \rightarrow \mathtt{a} \mid \mathtt{b}$

b) Show a natural PDA that accepts L.

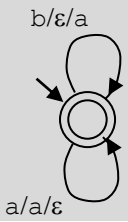


7) Let $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* : \#_{\mathtt{a}}(w) = \#_{\mathtt{b}}(w)$ and, for every prefix p of w, $\#_{\mathtt{a}}(p) \le \#_{\mathtt{b}}(p)\}$.

a) Show a context-free grammar that generates L.

$S \rightarrow \mathtt{b}S\mathtt{a} \mid SS \mid \varepsilon$

b) Show a deterministic PDA that accepts L.

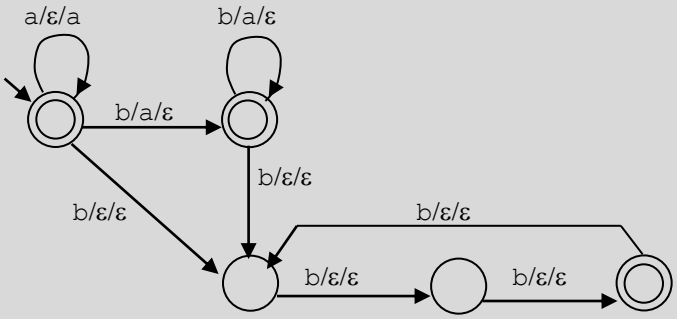


8) Let $L = \{\mathtt{a}^n\mathtt{b}^m : n > 0$ and $m = 3k+n$ for some $k \ge 0\}$.

a) Show a context-free grammar that generates L.

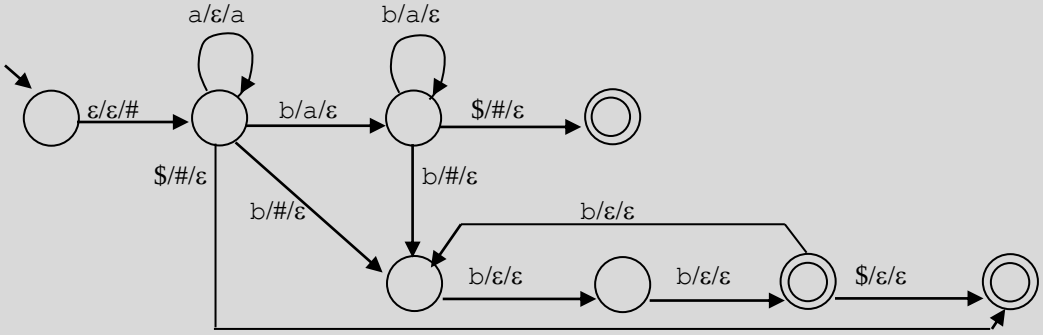
$S \rightarrow NK$
$N \rightarrow \mathtt{a}N\mathtt{b} \mid \varepsilon$ /* Generate $\mathtt{a}^n\mathtt{b}^n$.
$K \rightarrow \varepsilon \mid \mathtt{bbb}K$ /* Generate $\mathtt{b}^{3k}$.

b) Show a PDA P that accepts L.



c) Is P deterministic? If not, can you make an equivalent deterministic PDA for L? If so, do it. If not, can you make a deterministic PDA that accepts $L\$$? If so, do it.

There is not a deterministic PDA that accepts L. But there is one that accepts $L\$$:

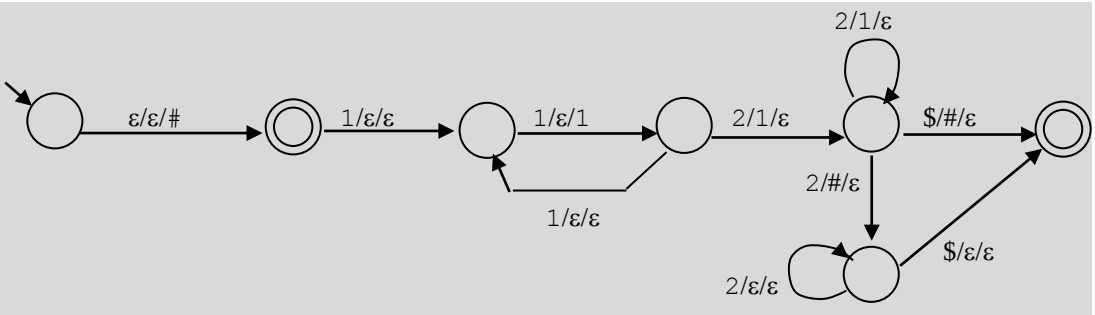


9) Let $L = \{1^n2^m : n > 0$ and n is even and $\exists k \geq 0\ (m = n/2 + k)\}$.

a) Show a context-free grammar that generates L.

$S \rightarrow 112B$
$S \rightarrow 11S2$
$B \rightarrow B2 \mid \varepsilon$

b) Show a deterministic PDA that accepts $L\$$.



10) [Luay Nakhleh] Let A and B be two regular languages. Let $C = \{xy : x \in A, y \in B, \text{and } |x| = |y|\}$. Prove that C is context-free by showing a PDA M such that $L(M) = C$.

a) Describe in English the key ideas in the construction of M.

Since A and B are regular, there exist DFSMs M_A and M_B that accept them. The first piece of M will essentially run M_A except that, every time it reads an input symbol, it will push a # onto its stack. The second piece of M will essentially run M_B except that, every time it reads an input symbol, it will pop a # from its stack. M will have one ε-transition from each accepting state of M_A to the start state of M_B. Taking that transition is equivalent to guessing that the middle of the input string has been reached. M will accept iff it reads all its input, lands in an accepting state of M_B, and clears its stack.

b) Give a formal description of M.

Since A is regular, it is accepted by some DFSM $M_A = (K_A, \Sigma_A, \delta_A, s_A, A_A)$. Since B is regular, it is accepted by some DFSM $M_B = (K_B, \Sigma_B, \delta_B, s_B, A_B)$. To construct M, do the following:

1. Rename the states of M_A and M_B so that they have no state names in common.
2. For every transition $((p, c), q)$ in M_A, add to M the transition $((p, c, \varepsilon), (q, \#))$.
3. For every accepting state q in M_A, add to M the transition $((q, \varepsilon, \varepsilon), (s_B, \varepsilon))$.
4. For every transition $((p, c), q)$ in M_B, add to M the transition $((p, c, \#), (q, \varepsilon))$.
5. Make s_A the start state of M.
6. Make each state in A_B an accepting state of M.

13 Context-Free and Noncontext-Free Languages

1) For each of the following languages L, state whether L is regular, context-free but not regular, or not context-free and prove your answer.

a) $\{xwx^R : x, w \in \{0, 1\}^+\}$.

Regular. The idea is that x can contain a single symbol and the rest of any string in L can be thought of as being in w. So, all it takes to be in L is to have length at least three and to begin and end with the same symbol. The we have that: $L = \{x\,w\,x : x \in \{0, 1\}$ and $w = \{0, 1\}^+\}$
$= (0(1 \cup 0)^+0) \cup (1(0 \cup 1)^+1)$.

b) [Luay Nakhleh] $\{w : \exists y \in \{a\}^* (w = yy^R)\}$.

Regular. L can be described by the regular expression $(aa)^*$.

c) $\{ww : w \in (ab)^n : n \geq 0\}$.

Regular. $L = \{(ab)^{2n} : n \geq 0\}$. So it can be described by the regular expression $(abab)^*$.

d) $\{w \in \{0 - 9\}^* : w$, when viewed as a base 10 integer, with no leading zeros, is divisible by 20$\}$.

Regular. The following regular expression defines L:

$0 \cup (((1\text{-}9)(0\text{-}9)^* \cup \varepsilon)\,(0 \cup 2 \cup 4 \cup 6 \cup 8)\,0)$

e) $\{w \in \{a, b, c\}^*$: all instances of aa precede all instances of $cc\}$.

Regular. A straightforward FSM keeps track of whether it has yet seen an instance of cc. If it has, then, if it sees an instance of aa, it goes to a dead state. All other states are accepting.

f) $\{w \in \{a, b\}^* : w = (a \cup b)^* a^n b^n (a \cup b)^*$ for some $n \geq 0\}$.

Regular. The key is that we can let n be 0. So $L = (a \cup b)^*$.

g) $\{x\,(a \cup b)^*\,x : x \in \{a, b\}^*\}$.

Regular. The key is that x can be equal to ε. So, letting x be ε, this expression can generate exactly the language $(a \cup b)^*$. If x takes on any other value, all we get is an alternative derivation for some string that can be generated just from $(a \cup b)^*$. So $L = (a \cup b)^*$, which is regular.

h) The language of arithmetic expressions without parentheses. Specifically, $L = \{w \in \{0\text{-}9, +, *, -, /\}^* : w$ is an arithmetic expression composed of decimal integers sameand the four binary operators +, *, -, /$\}$

Regular. The following regular expression defines L:

$(0 - 9)^+ ((+ \cup * \cup - \cup /)\,(0 - 9)^+)^*$

i) $L(G)$, where $G = S \rightarrow TS\mathtt{b}$
$S \rightarrow T\mathtt{b}$
$T \rightarrow T\mathtt{a}$
$T \rightarrow \varepsilon$

Regular (even though this grammar isn't regular). T generates $\mathtt{a}^*$. S generates strings that end in at least one $\mathtt{b}$ and that may have an initial $\mathtt{a}$ region. So $L(G) = \mathtt{a}^*\mathtt{b}^+$.

j) [Luay Nakhleh] $\{u\#v : u$ and v are the binary string encodings (with no leading zeros) of integers ≥ 1 and $u+v$ is an odd number$\}$.

Regular. The key is that if one, but not both, of u and v ends in 1, then $u\#v \in L$. Otherwise it isn't. So the following regular expression defines L:

$(\mathtt{1} \cup (\mathtt{1}(\mathtt{0} \cup \mathtt{1})^*\mathtt{1}))\#(\mathtt{1}(\mathtt{0} \cup \mathtt{1})^*\ \mathtt{0}) \ \cup \ (\mathtt{1}(\mathtt{0} \cup \mathtt{1})^*\ \mathtt{0})\#(\mathtt{1} \cup (\mathtt{1}(\mathtt{0} \cup \mathtt{1})^*\mathtt{1}))$

k) [Luay Nakhleh] $\{u\#v^R : u$ and v are the binary string encodings (with no leading zeros) of integers ≥ 1 and $v = 2u\}$.

Context-free not regular. The key is that, if $v = 2u$, then $\langle v \rangle = \langle u \rangle \mathtt{0}$. So every string in L has the form $u\#\mathtt{0}u^R$. The following context-free grammar generates L:

$S \rightarrow \mathtt{1}\ T\ \mathtt{1}$ /* u must start with $\mathtt{1}$ since no leading $\mathtt{0}$'s.
$T \rightarrow \mathtt{0}\ T\ \mathtt{0} \mid \mathtt{1}\ T\ \mathtt{1} \mid \#\ \mathtt{0}$

Proof not regular is by pumping. Let $w = \mathtt{1}^k\#\mathtt{0}\mathtt{1}^k$. Then y is $\mathtt{1}^p$, for some nonzero p, and it must occur in the initial $\mathtt{1}$ region. Pump in once. The resulting string is $\mathtt{1}^{k+p}\#\mathtt{0}\mathtt{1}^k$. But it is not true that $\mathtt{1}^k\mathtt{0} = 2\cdot(\mathtt{1}^{k+p})$. So this string is not in L.

l) $\{w \in \{\mathtt{0}\text{-}\mathtt{9}\}^* :$ the number of odd digits in w is equal to the number of even digits in $w\}$.

Context-free not regular. The following grammar generates L:

$S \rightarrow ESO$
$S \rightarrow OSE$
$S \rightarrow \varepsilon$
$E \rightarrow 0 \mid 2 \mid 4 \mid 6 \mid 8$
$O \rightarrow 1 \mid 3 \mid 5 \mid 7 \mid 9$

Proof not regular by pumping. Let $w = \mathtt{1}^k\mathtt{2}^k$. Since $|xy| \leq k$, y is $\mathtt{1}^p$, for some nonzero p. When we pump in, we increase the number of odd digits ($\mathtt{1}$'s) but not the number of even digits.

m) $\{ww^R : w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^*$ and $|w| \equiv_3 0\}$.

Context-free not regular. The following grammar generates L:

$S \rightarrow \mathtt{a}\ T\ \mathtt{a} \mid \mathtt{b}\ T\ \mathtt{b} \mid \mathtt{c}\ T\ \mathtt{c} \mid \varepsilon$ /* S must generate ww^R : $|w| \equiv_3 0$.
$T \rightarrow \mathtt{a}\ W\ \mathtt{a} \mid \mathtt{b}\ W\ \mathtt{b} \mid \mathtt{c}\ W\ \mathtt{c} \mid \varepsilon$ /* T must generate ww^R : $|w| \equiv_3 2$.
$W \rightarrow \mathtt{a}\ S\ \mathtt{a} \mid \mathtt{b}\ S\ \mathtt{b} \mid \mathtt{c}\ S\ \mathtt{c} \mid \varepsilon$ /* T must generate ww^R : $|w| \equiv_3 1$.

Proof not regular by pumping. Let $w = \mathtt{a}^{2k}\mathtt{b}^k\mathtt{b}^k\mathtt{a}^{2k}$. Then y is $\mathtt{a}^p$, for some nonzero p, and it must occur in the initial $\mathtt{a}$ region. Pump in once. The resulting string is $\mathtt{a}^{2k+p}\mathtt{b}^k\mathtt{b}^k\mathtt{a}^{2k}$. It is not in L because there are more $\mathtt{a}$'s in its first half than in its second half. Thus its first half is not the reverse of its second half.

n) $\{w \neq w^R : w \in \{a, b\}^*\}$.

Context-free not regular. L can be generated by the grammar $G = (\{S, D, X\}, \{a, b\}, R, S\}$, where R =

$S \rightarrow aSa \mid bSb \mid D$
$D \rightarrow aXb \mid bXa$ /* D generates one mismatched pair.
$X \rightarrow aXa \mid aXb \mid bXa \mid bXb \mid \varepsilon \mid a \mid b$

If L were regular, then $\neg L = \{w = w^R : w \in \{a, b\}^*\}$ would also be regular. But we show that it is not by pumping. Let $w = a^k b a^k$. Then y is a^p, for some nonzero p, and it must occur in the initial a region. Pump in once. The resulting string is $a^{k+p}ba^k$. It is not in L because there are $k+p$ a's before the b reading from the left but only k a's before the b, reading from the right.

o) $\{w : \exists x \in \{a, b\}^* (w = [x][x^R])\}$. For example, the string, $[aab][baa] \in L$.

Context-free not regular. L can be generated by the following grammar:

$S \rightarrow [T]$
$T \rightarrow aTa \mid bTb \mid][$

We show that L is not regular by pumping. Let $w = [a^k b][ba^k]$. y must occur in the first a region, so it must be a^p, for some nonzero p.. Set q to 2. The resulting string is $[a^{k+p}b][ba^k]$, which is not in L because the first bracketed region changed but the second one didn't.

p) $\{a^*b^*c^* - \{a^n b^n c^n : n \geq 0\}\}$.

Context-free not regular. $L = \{a^i b^j c^k : i, j, k \geq 0$ and $i \neq j$ or $j \neq k\}$. L can be accepted by the PDA shown in Example 12.8 except that the top branch should not be present.

If L were regular, then $\neg L$ would also be regular.

$\neg L = \{w \in \{a, b, c\}^*$ with letters out of order$\} \cup \{a^n b^n c^n : n \geq 0\}$. If $\neg L$ is regular, then so is:
$L_1 = \neg L \cap a^*b^*c^*$. But:
$L_1 = \{a^n b^n c^n : n \geq 0\}$, which is not even context-free, much less regular.

q) $\{a^n b^m c^k : n = m + k$ or $m = n + k\}$.

Context-free not regular. A grammar for L is:

$S \rightarrow A \mid B$
$A \rightarrow aAc \mid A'$ /* $n = m + k$
$A' \rightarrow aA'b \mid \varepsilon$
$B \rightarrow B_1 B_2$ /* $m = n + k$
$B_1 \rightarrow aB_1b \mid \varepsilon$
$B_2 \rightarrow bB_2c \mid \varepsilon$

Proof not regular by pumping. Let $w = a^{2k} b^k c^k$. Note that $w \in L$ because it satisfies the first condition ($n = m + k$). (Do not get confused by the two different uses of the variable name k.) Then y is a^p, for some nonzero p. Pump in once. The resulting string is $a^{2k+p} b^k c^k$. It is not in L because it satisfies neither of the conditions for membership in L:

- $2k + p \neq k + k$, and
- $k \neq 2k + p + k$

r) $\{a^i b^j c^k\}^* : i, j, k \geq 0 \text{ and } j < i + k\}$.

Context-free not regular. A context-free grammar for L is:

$S \rightarrow S_1$	/* Will guarantee at least one a without a matching b.
$S \rightarrow S_2$	/* Will guarantee at least one c without a matching b.
$S_1 \rightarrow X_1\ Y_1$	/* X_1 is the a/b region. Y_1 is the b/c region.
$X_1 \rightarrow \mathtt{a}\ X_1\ \mathtt{b} \mid \mathtt{a}\ X_1 \mid \mathtt{a}$	/* Makes at least one more a than b.
$Y_1 \rightarrow \mathtt{b}\ Y_1\ \mathtt{c} \mid Y_1\ \mathtt{c} \mid \varepsilon$	
$S_2 \rightarrow X_2\ Y_2$	/* X_2 is the a/b region. Y_2 is the b/c region.
$X_2 \rightarrow \mathtt{a}\ X_2\ \mathtt{b} \mid \mathtt{a}\ X_2 \mid \varepsilon$	
$Y_2 \rightarrow \mathtt{b}\ Y_2\ \mathtt{c} \mid Y_2\ \mathtt{c} \mid \mathtt{c}$	/* Makes at least one more c than b.

Not regular, by pumping. Let $w = \mathtt{a}^{k+1}\mathtt{b}^k$. Set q to 0. The resulting string is $\mathtt{a}^{k+1-p}\mathtt{b}^k$. It is not in L because, since p is at least 1, there are not more a's than b's.

s) $\{a^i b^j c^k : (i \text{ is even and } j = i) \text{ or } (i \text{ is odd and } j = k)\}$.

Context-free, not regular. A context-free grammar for L is:

$S \rightarrow A_E$
$S \rightarrow A_O$
$A_E \rightarrow A_1\ C$
$C \rightarrow \varepsilon \mid \mathtt{c}C$
$A_1 \rightarrow \varepsilon \mid \mathtt{aa}A_1\mathtt{bb}$
$A_O \rightarrow A_2\ K$
$A_2 \rightarrow \mathtt{a} \mid \mathtt{aa}A_2$
$K \rightarrow \varepsilon \mid \mathtt{b}K\mathtt{c}$

We show not regular using the Pumping Theorem. Let $w = \mathtt{a}^{2k}\mathtt{b}^{2k}$. y must occur in the first k characters, so it must be $\mathtt{a}^p$ for some nonzero p. Pump in twice. There will still be an even number of a's. But the number a's won't equal the numer of b's. So the resulting string is not in L. Alternatively, pump in once. If $|y|$ is even, the argument is the same as the one we just gave. If $|y|$ is odd, the resulting string is also not in L because the number of b's doesn't equal the number of c's.

t) $\{w \in \{0, 1\}^* : \#_0(w) = \#_1(w)$ and, at every point in w, reading from the left, there have not more 1's than 0's$\}$. =

Context-free, not regular. A context-free grammar for L is:

$S \rightarrow \varepsilon$
$S \rightarrow S\ S$
$S \rightarrow 0\ S\ 1$

We show not regular using the Pumping Theorem. Let $w = 0^k1^k$. y must occur in the first k characters, so it must be 0^p for some nonzero p. Pump out once. The resulting string is not in L because it does not contain equal numbers of 0's and 1's.

u) [Luay Nakhleh] $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } j > i + k\}$.

Context-free, not regular. A grammar for L is:

$S \rightarrow TBX$
$T \rightarrow aTb \mid \varepsilon$
$X \rightarrow bXc \mid \varepsilon$
$B \rightarrow bB \mid b$

Not regular, by pumping. Let $w = a^k b^{k+1}$. Set q to 2. The resulting string is $a^{k+p} b^{k+1}$. It is not in L because there are not more b's than a's.

v) [Luay Nakhleh] $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } j > max(i, k)\}$.

Not context-free. We prove it using the Pumping Theorem. Let k be the constant from the Pumping Theorem and let $w = a^k b^{k+1} c^k$. Let region 1 contain all the a's, region 2 contain all the b's, and region 3 contain all the c's. If either v or y crosses numbered regions, pump in once. The resulting string will not be in L because it will violate the form constraint. We consider the remaining cases:

(1, 1): Pump in once. This increases the number of a's and thus the *max* of the number of a's and c's. But the number of b's is unchanged so it no longer greater than that maximum.
(2, 2): Pump out once. The *max* of the number of a's and c's is unchanged. But the number of b's is decreased and so it is no longer greater than that maximum.
(3, 3): Same argument as (1, 1) but increases the number of c's.
(1, 2).(2, 3): Pump out once. The *max* of the number of a's and c's is unchanged. But the number of b's is decreased and so it is no longer greater than that maximum.
(1, 3): Not possible since $|vxy|$ must be less than or equal to k.

w) $\{x\#y\#z : x, y, z \in \{a, b\}^+ \text{ and } (|x| - |y| = 2 \text{ or } |y| - |z| = 2\}$. For example, abbbb#aabbb#aba $\in L$.

Context-free not regular. A grammar for L is:

$S \rightarrow A \# X \mid X \# B$
$A \rightarrow C A C$ /* $|x| - |y| = 2$
$A \rightarrow C C C \# C$
$B \rightarrow C B C$ /* $|y| - |z| = 2$
$B \rightarrow C C C \# C$
$C \rightarrow a \mid b$
$X \rightarrow C X \mid X C \mid C$

L is not regular, which we show by pumping. Let $w = a^{k+2} \# a^k \# a^{k+2}$. y must occur in the first a region and must be a^p for some nonzero p. Pump in. The three regions of the resulting string are now of the following lengths:

x: $k + 2 + p$
y: k
z: $k + 2$

So neither of the required conditions is met and the resulting string is not in L.

x) L_1^*, where $L_1 = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$.

Context-free not regular. A grammar for L_1 is in the book. The context-free languages are closed under Kleene star. But this fact isn't really necessary, since $L = L_1$.

L is not regular, which we show by pumping. Note that every string in *L* contains an equal number of a's and b's. Let $w = a^k b^k$. y must be a^p for some nonzero *p*. Pump in once. The resulting string is not in *L* because it has more a's than b's.

y) L_1^*, where $L_1 = \{0^+1^n0^+1^n : n \geq 0\}$.

Context-free but not regular. *L* can be generated by the following context-free grammar:

$$S \rightarrow Z\ T\ S \mid \varepsilon$$
$$T \rightarrow 1\ T\ 1 \mid Z$$
$$Z \rightarrow 0\ Z \mid 0$$

L is not regular. If it were, then $L' = L \cap 01^*01^*$ would also be regular. But we can use the Pumping Theorem to show that it is not.

Let $w = 0\ 1^k\ 0\ 1^k$
1 | 2 |3| 4

If *y* includes region 1, pump in once. The resulting string will no longer be in 01^*01^*. So it is not in *L′*. The only other place *y* can fall is in region 2. Pump in once. The number of 1's in region 2 will no longer equal the number of 1's in region 4. So the resulting string is not in *L′*.

z) $L_1 \cap L_2$, where $L_1 = \{a^n b^m : 0 \leq m \leq n\}$ and $L_2 = \{a^n b^m : 0 \leq n \leq m\}$.

Context-free but not regular. $L = A^nB^n = \{a^n b^m : n \geq 0\}$.

aa) $\{ss^Rtt^R : s, t \in \{a, b\}^*\}$.

Context-free but not regular. *L* can be generated by the following context-free grammar:

$$S \rightarrow S' \mid T'$$
$$S' \rightarrow a\ S' a \mid b\ S' b \mid \varepsilon$$
$$T' \rightarrow a\ T' a \mid b\ T' b \mid \varepsilon$$

L is not regular, which we show using the Pumping Theorem. Let $w = a^k b\ b\ a^k b^k a\ a\ b^k$.
1 |2 |3 | 4 | 5 |6|7| 8
s | s^R | *t* | t^R

Note that, although either *s* or *t* may be empty, given the definition of *L*, neither of them can be empty in *w* since *w* starts with an a and ends with a b. *y* must fall in region 1. It is a^p for some nonzero *p*. Pump in once. The resulting string is not in *L*. The boundary between *s* and *t* cannot move since *t* ends in a b. So it must also start with a b. But *s* starts with an a, so it must end in one also. Thus the boundary between *s* and *t* must remain at the boundary between regions 4 and 5. But, after pumping, the number of a's in region 1 does not equal the number of a's in region 4. So the substring in regions 1 through 4 is no longer of the form ss^R.

bb) $\{ss^Rss^R : s \in \{a, b\}^*\}$.

Not context-free. If L were context-free, then $L_1 = L \cap a^*bba^*a^*bba^*$ would also be context free. But we show that it is not by pumping.

Let $w = a^k\ bb\ a^k a^k\ bb\ a^k$
1 | 2 | 3 | 4 | 5

Observe that any string in L_1 must satisfy both of the following properties that relate the lengths of the five regions of any such string (including w):

[A] $|3| = 2 \cdot |1| = 2 \cdot |5|$
[B] $|1| = |5|$

If either v or y from the Pumping Theorem contains two or more distinct letters, pump in once. The resulting string will not be in L_1 because it will violate the form constraint. If nonempty v or y is in region 2 or region 4, pump in once. The resulting string will violate the constraint that both region 2 and region 4 must be exactly the string bb. We consider the remaining cases for where nonempty v and y can fall:

(1, 1), (5, 5) : violate [B].
(3, 3) : violates [A].
(1, 3), (3, 5) : violate [B]
All others: ruled out by the requirement that $|vxy|$ must be less than k.

cc) $\{w \in \{a, b, c, d\}^* : \#_a(w) = \#_b(w) = 2 \cdot \#_c(w)\}$.

Not context-free. If L were context-free, then $L_1 = L \cap a^*b^*c^*$ would also be context free. But we show that it is not by pumping.

Let $w = a^{2k} b^{2k} c^k$.
1 | 2 | 3

If either v or y from the Pumping Theorem contains two or more distinct letters, pump in once. The resulting string will not be in L_1 because it will violate the form constraint. We consider the remaining cases:
(1, 1) Pump in. Fewer b's than a's.
(2, 2) Pump out. Fewer b's than a's.
(3, 3) Pump in. Not enough a's or b's.
(1, 2) Pump out. Fewer b's than twice the number of c's.
(2, 3) Pump in. Unequal a's and b's.
(1, 3) $|vxy|$ must be less than k.

dd) $\{w \in \{a, b, c\}^* : \#_a(w) = max(\#_b(w), \#_c(w))\}$.

Not context-free, which we show by pumping.

Let $w = a^k b^k c^k$.
1 | 2 | 3

If either *v* or *y* from the pumping theorem contains two or more distinct letters, pump in once. The resulting string will not be in *L* because it will violate the form constraint. We consider the remaining cases:
(1, 1) Pump in once. Too many a's, since the number of a's increased but $max(\#_b(w), \#_c(w))$ didn't change.
(2, 2), (3, 3), (2, 3) Pump in once. $max(\#_b(w), \#_c(w))$ increased, but $\#_a(w)$ did not.
(1, 2) Pump out. $max(\#_b(w), \#_c(w))$ doesn't change, but $\#_a(w)$ does.
(1, 3) $|vxy|$ must be less than *k*.

ee) $\{w \in \{0\text{-}9\}^*$: the number of digits that are equal to 0 mod 3 equals the number of digits equal to 1 mod 3 equals the number of digits equal to 2 mod 3$\}$. For example, 012, 312, 645, 645672 are in *L*, but 111, 12123 are not.

Not context-free. If *L* were context-free, then $L_1 = L \cap 1^*2^*3^*$ would also be context-free. But we show that it is not by pumping.

Let $w = 1^k 2^k 3^k$.
1 | 2 | 3

If either *v* or *y* from the pumping theorem crosses regions, pump in once. The resulting string will violate the form constraint and so not be in L_1. We now consider the other ways in which *v* and *y* could occur:
(1, 1): We change the number of digits in 1 but not in 2 or 3, so we've changed the number of digits equal to 1 mod 3 but not the number equal to either 2 or 0.
(2, 2): Same argument except we change the number of digits equal to 2 mod 3 and not the others.
(3, 3): Same argument except we change the number of digits equal to 0 mod 3 and not the others.
(1, 2): We change the number of digits equal to 1 mod 3 and the number equal to 2, but not the number equal to 0.
(2, 3): Same argument except that we don't change the number of digits equal to 1 mod 3.
(1, 3): Not possible since $|vxy| \leq k$.

ff) $\{w : w = yxx^Ry$ and $x, y \in \{a, b\}^*\}$.

Not context-free. To avoid clashing with the names from the pumping theorem, we will rename the regions of each string in *L* to *m* and *n*. So each string in *L* is of the form mnn^Rm. To make it easy to see why *L* is not context-free, we focus on the case where *n* is empty. If *L* were context-free, then $L_1 = L \cap a^*b^*a^*b^*$ would also be context-free. But we show that it is not by pumping.

Let $w = a^k b^k a^k b^k$.
1 | 2 | 3 | 4

If either *v* or *y* from the pumping theorem crosses regions, pump in once. The resulting string will violate the form constraint and so not be in L_1. If $|vy|$ is odd, pump in once, resulting in an odd length string. All strings in *L* (and thus also L_1) have even length. We now consider the other ways in which *v* and *y* could occur:
(1, 1): Pump in once. *m* must start with an a (since the first character of *w* is an a) and end with a b (since the last character of *w* is a b). The first copy of *m* starts with $k + |vy|$ a's, so the second one must too, but there aren't enough.

(4, 4): A similar argument. Pump in once. Now the second copy of m ends with $k + |vy|$ b's, so the second one must too, but there aren't enough.
(2, 2): Pump in once. m must start with an a and end with a b. The first copy of m ends with $k + |vy|$ b's, so the second one must too, but there aren't enough.
(3, 3): A similar argument. Pump in once. m must start with an a and end with a b. The second copy of m starts with $k + |vy|$ a's, so the second one must too, but there aren't enough.
(1, 2): Pump in once. As for (1, 1), the a region of the first m grows, but the a region of the second m doesn't.
(3, 4): A similar argument. Pump in once. As for (3, 3), a region of second m grows, but a region of first m doesn't.
(2, 3): Pump in once. If we think of the resulting string as still being just mm, we've changed the first half and the second half in different ways. But maybe we've just introduced a nonempty n. But that cannot be so since that n would have to start with a b. So n^R would have to end with a b also, but the pumped in string ends with an a. So there is no way to parse the resulting string so it is in L_1.
(1, 3), (1, 4), (2, 4): Not possible since $|vxy| \le k$.

gg) $\{w = xyz : x \in 0^*, y \in 1^*, z \in 0^*, |x| = |z| \text{ and } |y| = 2 \cdot |z|\}$.

Not context-free. Let $w = 0^k 1^{2k} 0^k$.
1 | 2 | 3

If either v or y from the pumping theorem crosses regions, pump in once. The resulting string will violate the form constraint and so not be in L. We now consider the other ways in which v and y could occur:

(1, 1), (1, 2), (2, 3), (3, 3): Pump in once. The lengths of the x and z regions will no longer be equal because one changed and the other didn't.
(2, 2): Pump out. Since the length of the y region changed and the length of the z region didn't, it will no longer be true that $|y| = 2 \cdot |z|$.
(1, 3): Not possible since $|vxy| \le k$.

hh) $\{x\, x_{neg} : x \in \{0, 1\}^*\}$, where $x_{neg} = x$ with all 0's replaced by 1's and 1's replaced by 0's.

Not context-free. If L were context-free, then $L_1 = L \cap 0^*1^*0^*1^*0^*1^*$ would also be context-free. But we show that it is not by pumping. Let $w = 0^k 1^k 0^k 1^k 0^k 1^k$.
1 | 2 | 3 | 4 | 5 | 6

If either v or y crosses regions, pump in once. The resulting string will violate the form constraint and so not be in L_1. If $|vy|$ is odd, pump in once. The resulting string cannot be in L_1 since all strings in L, and thus in L_1, have even length. Otherwise:

- If v and y are in some combination of regions 1, 2, 3: Pump in once. At least one 0 from region 3 flows into the beginning of the second half of the string. But then both the first half and the second half start with a 0. Thus the second half is not x_{neg}.
- If v and y are in some combination of regions 4, 5, 6: Similar argument except now both halves of the string end in 1.
- (3, 4): Pump out. The 1 region that begins the second half of the string won't have as many 1's as there are 0's at the beginning of the first half of the string.
- All other combinations: Not possible since $|vxy| \le k$.

ii) $\{w \in \{a, b\}^* : w = xyz, |x| = |y|, z = (x \text{ with every } a \text{ replaced by } b \text{ and every } b \text{ replaced by } a)\}$.

Not context-free. If L were context-free, then $L_1 = L \cap a^*ba^*b^*a$ would also be context-free. But we show that it is not by pumping. To avoid clashing with the names from the pumping theorem, we will rename the regions of w *d*, *e*, and *f*.

$$\begin{array}{l} \quad\ \ | \ d \ \ | \ \ e \ | \ f \\ \text{Let } w = a^k\, b\, a^{k+1}\, b^k\, a. \\ \quad\ \ | \ 1|2| \ \ 3 \ \ | 4 \, | \, 5 \end{array}$$

We break w into regions as shown above. If either v or y from the pumping theorem crosses numbered regions, pump in once. The resulting string will not be in L_1 because it will violate the form constraint. If either v or y contains region 2 or 5, pump in once and the form constraint will be violated. If $|vy|$ is not divisible by 3, the resulting string will not be in L_1, since $|d| = |e| = |f|$. We now consider the other ways in which v and y could occur:

(1, 1): Pump in once. Since $|vy|$ is divisible by 3, each of the three segments, *d*, *e*, and *f*, grows by at least one character. In other words, the boundaries between regions shift to the left. So *d* still starts with a. But *f* also starts with a, when it needs to start with b. So the resulting string is not in L_1.

(3, 3): Same argument except that the *d*/*e* boundary shifts to the right.

(4, 4): Pump out once. The boundaries now move to the left. *d* will end in a, but so will *f*.

(1, 3): Pump in once. *f* now starts with a, as does *d*.

(3, 4): Pump in once. *d* ends in a, as does *f*.

(1, 4): Not possible since $|vxy| \le k$.

jj) $\{x\ a\ y : x, y \in \{0, 1\}^* \text{ and } x \text{ occurs somewhere as a substring of } y\}$.

Not context-free. If L were context-free, then $L_1 = L \cap 0^*1^*a0^*1^*$ would also be context-free. But we show that it is not by pumping. Let $w = 0^k 1^k a\, 0^k\, 1^k$.

$$| 1| 2 |3 | 4 |5$$

We break w into regions as shown above. If either v or y from the pumping theorem crosses numbered regions, pump in once. The resulting string will not be in L_1 because it will violate the form constraint. If either v or y contains region 3, pump in once and the form constraint will be violated. We now consider the other ways in which v and y could occur:

(1, 1), (2, 2), (1, 2): Pump in. The region before the a is too long to be a substring of the region after the a.

(4, 4), (5,5), (4, 5): Pump out. The region before the a is too long to be a prefix of the region after the a.

(2, 4): Pump out. There are more 0's before the a than after the a. So the region before the a does not occur as a substring of the region after the a.

All other possibilities are ruled out by the requirement that $|vxy|$ be $\le k$.

kk) $\{w \in \{a, b, c\}^* :$ every a has a matching b and a matching c somewhere in w (and no b or c is considered to match more than one a)$\}$.

Not context-free. If L were context-free, then $L_1 = L \cap a^*b^*c^*$ would also be context-free. But we show that it is not by pumping. Let $w = a^k\, b^k\, c^k$.

$$1 \mid 2 \mid 3$$

We break w into regions as shown above. If either v or y crosses numbered regions, pump in once. The resulting string will not be in L_1 because it will violate the form constraint. We now consider the other ways in which v and y could occur:

(1, 1): Pump in once. The number of a's went up, but the number of b's didn't so there is no longer a matching b for every a.
(2, 2): Pump out once. The number of b's went down, but the number of a's didn't so there is no longer a matching b for every a.
(3, 3): Pump out once. The number of c's went down, but the number of a's didn't so there is no longer a matching c for every a.
(1, 2): Pump in once. The number of a's went up, but the number of c's didn't so there is no longer a matching c for every a.
(2, 3): Pump out once. The number of c's went down, but the number of a's didn't so there is no longer a matching c for every a.
(1, 3): Not possible since $|vxy| \le k$.

ll) (Luay Nakhleh) $\{a^i b^j c^k : i > 1$ and $j, k \ge 0$ and $i \cdot j$ is prime$\}$.

Not context-free. If L were context-free, then $L_1 = L \cap a^*b$ would also be context-free. But we show that it is not by pumping. Let $w = a^n b$., where n is the smallest prime that is greater than $k + 1$. Copy proof from Example 8.13 for the rest.

mm) $\{1^{n!} : n \ge 0\}$.

Not context-free. Let $w = 1^{(k+1)!}$. vy must be 1^p for some $0 < p \le k$. Pump in once. The maximum length of the resulting string is $(k + 1)! + k$. Since $k \ge 1$, $k < (k + 1)!$. So the new string is less than twice the length of the original w. But the next string in L, after w, has length $(k + 2)! = (k + 2)(k + 1)!$. This string has more than twice the length of w. So the string that results from pumping is too short to be in L.

nn) $\{0^i 1^j : i, j \ge 0$ and $j = i^2\}$.

Not context-free. Let $w = 0^k 1^{k^2}$. Let region 1 be all the 0's and region 2 be all the 1's. For any string x, initially w but then any string created by pumping, let $p(x)$ be $\#_0(x)$ and let $q(x)$ be $\#_1(x)$. In order for x to be in L, it must be the case that $q(x) = p(x)^2$. (Alternatively, $p(x) = \sqrt{q(x)}$).

First, we notice that neither v nor y can cross regions or, when we pump in, we will get interleaved 0's and 1's and thus strings that are not in L. So we must consider three possibilities for (x, y):
(1, 1): Pump out, thus reducing the number of 0's and creating a string x where $p(x) < \sqrt{q(x)}$.
(2, 2): Pump in, thus increasing the number of 1's and creating a string x where $q(x) > p(x)^2$.
(1, 2): Pump in. We require that $q(x) = p(x)^2$. We've started with $p(x) = k$ and $q(x) = k^2$. Suppose we add just a single 0 when we pump. Then $p(x) = k + 1$. So $q(x)$ must be $k^2 + 2k + 1$. So we must add $2k + 1$ copies of 1. But $|vxy| \le k$. So we can't add that many 1's. And if y contains more than one 0, we will need even more 1's. So there is no way to pump in once and get a string in L.

oo) $\{w \in \{0, 1\}^* : \#_1(w) = (\#_0(w))^2\}$.

Not Context Free. Let $L_1 = L \cap 0^*1^*$.
$= 0^i 1^j$, $i. j \ge 0$ and $j = i^2$.

If L is CF, then so is L_1, since the CF languages are closed under intersection with the regular languages. But we have already proved (in the previous problem) that L_1 is not CF.

pp) $\{x\#y : x, y \in \{0, 1\}^*$ and when x and y are viewed as binary numbers, $y = 2x\}$. For example, the string $101\#1010 \in L$.

Not Context Free. Let $L_1 = L \cap 1^*0^*\#1^*0^*$.

If L is CF, then so is L_1, since the CF languages are closed under intersection with the regular languages. But L_1 is not CF, which we show by pumping.

Let $w = 1^k 0^k \# 1^k 0^{k+1}$.
1 | 2 |3| 4 | 5

We break w into regions as shown above. If either v or y from the pumping theorem crosses numbered regions, pump in once. The resulting string will not be in L_1 because it will violate the form constraint. If either v or y contains region 3, pump in once and the form constraint will be violated. We now consider the other ways in which v and y could occur:
(2, 4): Set q to 2. The arithmetic constraint imposed by L will be violated since it requires that the number of 0's in region 5 be exactly one greater than the number of 0's in region 2.
All other possibilities are ruled out by the requirement that $|vxy|$ be $\leq k$.

2) Let $L_1 = L_2 \cup L_3$.
a) Show values for L_1, L_2, and L_3, such that L_1 is context-free but neither L_2 nor L_3 is.

Let: $L_1 = \{a^i b^i c^* : i \geq 1\}$.
$L_2 = \{a^i b^i c^j : j \leq i\}$.
$L_3 = \{a^i b^i c^j : j \geq i\}$.

b) Show values for L_1, L_2, and L_3, such that L_1 is regular but neither L_2 nor L_3 is.

Let: $L_1 = \{a^* b^*\}$.
$L_2 = \{a^i b^j : j \leq i\}$.
$L_3 = \{a^i b^j : j \geq i\}$.

3) Give an example of a regular language L_1 that has a superset L_2 that is context-free but not regular.

$L_1 = \{\varepsilon\}$. $L_2 = A^n B^n$.

4) Let $L = \{w \in \{a, \#\}^* : w$ is in the sequence: a, a#aa, a#aa#aaa, a#aa#aaa#aaaa, …$\}$.
a) Is L regular, context-free but not regular, or not context-free? Prove your answer.

Not context-free, which we prove using the Pumping Theorem. Let $w = a\#aa\#aaa\#aaaa \ldots \#a^k$. The next longer string in L contains k+2 additional characters (k+1 a's plus one new #). Pump in once. The maximum number of pumped-in characters is k. So the resulting string is too short to be in L.

b) Now consider $\neg L$. Is it regular, context-free but not regular, or not context-free? Prove your answer.

Context-free. The idea is that, to recognize that a string w is not in L, it suffices to find a single adjacent pair of a regions that fail to satisfy the constraint that the second one contain exactly one more a than the first one does. So a PDA for nondeterministically chooses a region to begin examining. It pushes the a's in that region onto the stack. It then pops a's for each a in the next region and accepts if there's not exactly one extra a in the second sequence.

5) [Luay Nakhleh] Let $L' = \texttt{1*0*}$. Define $x_{neg} = x$ with all `0`'s replaced by `1`'s and `1`'s replaced by `0`'s. Consider the language:

$L = \{w_1w_2 : w_1 \in L', w_2 \in L', \text{ and } \exists x \in L' (w_2 = x_{neg})\}$

Is L regular, context-free but not regular, or not context-free? Prove your answer.

Regular. Consider any string x in L'. Its `1`'s (if there are any) come first, then its `0`'s (if there are any). So, in x_{neg}, the `0`'s come first, then the `1`'s. No string with `0`'s followed by `1`'s can be in L'. So the only strings that can be both in L' and equal to x_{neg} for some x in L' are strings that have at most one distinct symbol. Thus the only candidates for w_2 are strings in the language ($\texttt{0*} \cup \texttt{1*}$). But w can be any string in L'. So:

$L = \texttt{1*0*}(\texttt{0*} \cup \texttt{1*})$
$\quad \texttt{1*0*1*}$

6) Remember the childhood song, "Old McDonald". In case you don't, here are the words to one version:

Old McDonald had a farm, E-I-E-I-O
And on his farm he had a cow, E-I-E-I-O
With a "moo" "moo" here and a "moo" "moo" there
Here a "moo" there a "moo"
Everywhere a "moo" "moo"
Old McDonald had a farm, E-I-E-I-O

Old McDonald had a farm, E-I-E-I-O
And on his farm he had a pig, E-I-E-I-O
With a "snort" "snort" here and a "snort" "snort" there
Here a "snort" there a "snort"
Everywhere a "snort" "snort"
With a "moo" "moo" here and a "moo" "moo" there
Here a "moo" there a "moo"
Everywhere a "moo" "moo"
Old McDonald had a farm, E-I-E-I-O

Old McDonald had a farm, E-I-E-I-O
And on his farm he had a horse, E-I-E-I-O
With a "neigh" "neigh" here and a "neigh" "neigh" there
Here a "neigh" there a "neigh"
Everywhere a "neigh" "neigh"
With a "snort" "snort" here and a "snort" "snort" there
Here a "snort" there a "snort"
Everywhere a "snort" "snort"
With a "moo" "moo" here and a "moo" "moo" there
Here a "moo" there a "moo"
Everywhere a "moo" "moo"
Old McDonald had a farm, E-I-E-I-O

...mouse.. "squeek"

...porcupine... "oo-ah"

Suppose that we want to consider the language of Old McDonald songs, which we'll define to consist of songs with the form of the original but that may contain any finite number of verses and any set of animals (with their associated sounds), in any order. So the classic song, as given above, can be defined by specifying: cow/moo, pig/snort, horse/neigh, mouse/squeek, porcupine/oo-ah. We want to define grammars for verses and for whole songs of this form. We want a grammar that is general and allows for an arbitrarily large collection of animals and sounds. (Clearly if the set of possible songs is finite, the song language is regular and not interesting from our current point of view.) We'll start by defining the following nonterminal symbols:

HADAFARM → Old McDonald had a farm,
ONFARM → And on his farm he had a
EIEIO → E-I-E-I-O
ANIMAL → cow | pig | horse | mouse | porcupine | chick | cat
SOUND → "moo" | "snort" | "neigh" | "squeek" | "oo-ah" | "meow"

a) As a first step, we will ignore the way in which one verse of this song must be related to the verses that come before and after it. Instead, we will focus just on what a single legal verse can look like. Do the best you can to write a context-free grammar for such a single verse.

V → *HADAFARM EIEIO ONFARM ANIMAL EIEIO WITHA HADAFARM EIEIO*
WITHA → With a *SOUND SOUND* here and a *SOUND SOUND* there Here a *SOUND* there a *SOUND* Everywhere a *SOUND SOUND*

b) Are there any characteristics of a legal single verse that are not captured by your grammar? If so, show an example of a verse that you can generate but that is not allowed.

There are two problems:

- There is no requirement that the verse use the same *SOUND* throughout.
- There is no requirement that the *SOUND* match the *ANIMAL*.

So this grammar can generate:

Old McDonald had a farm, E-I-E-I-O
And on his farm he had a cow, E-I-E-I-O
With a "snort" "snort" here and a "snort" "snort" there
Here a "snort" there a "neigh"
Everywhere a "moo" "click"
Old McDonald had a farm, E-I-E-I-O

We could fix this problem by having a separate verse grammar for each animal/sound pair.

c) Ignoring any of the problems you mentioned above, how would your grammar have to change to add dogs, who say, "bow" "wow" (instead of, for example, "bow" "bow").

We would need to distinguish between $SOUND_1$'s, which can be the first syllable and $SOUND_2$'s, which can be the second syllable.

d) Now consider the problem of writing a grammar for an entire song. Again, do the best you can to capture legal song structure as a context-free grammar. In addition to any problems that you mentioned in part b, are there characteristics of legal songs that you cannot describe?

The structure that we want is:

Verse 1: $animal_1$ $sound_1$
Verse 2: $animal_2$ $sound_2$ $sound_1$
Verse 3: $animal_3$ $sound_3$ $sound_2$ $sound_1$

We can write:

SONG → *V* | *V SONG*
V → *HADAFARM EIEIO ONFARM ANIMAL EIEIO WITHA HADAFARM EIEIO*
WITHA → With a *SOUND SOUND* here and a *SOUND SOUND* there Here a *SOUND* there a *SOUND*
Everywhere a *SOUND SOUND*
WITHAS → *WITHA* | *WITHAS*

This lets us produce a song that is a sequence of verses. And each verse can be an arbitrarily long sequence of sound fragments. But we have two new problems:

- Nothing guarantees that each verse has one more song fragment than the one before it.
- Nothing guarantees that the order of the song fragments matches their order in earlier verses.

Neither of these problems can be fixed in a context-free grammar that's general enough to allow us to add arbitrary animal/sound pairs to the repertoire.

7) Define *bothways*(L) = $L \cap L^R$. Are the context-free languages closed under *bothways*?

No. Let $L = \{a^n b^n a^m : n, m \geq 0\}$. L is context-free since it can be generated by the following grammar:

$S \rightarrow S_1 A$
$S_1 \rightarrow aS_1b \mid \varepsilon$
$A \rightarrow aA \mid \varepsilon$

Then $L^R = \{a^m b^n a^n\}$. So *bothways*(L) = $\{a^n b^n a^m\} \cap \{a^m b^n a^n\} = \{a^n b^n a^n\}$, which is not context-free. We show this by using the Pumping Theorem. Let $w = a^k b^k a^k$. If either v or y contains two distinct letters, pump in once. The letters will be out of order. If neither of them does, then between them they can be in at most two of the three regions. Pump in once. The resulting string will no longer have the three regions of equal length.

8) Define *reverseanddouble*(L) = $\{w : \exists x \in L \ (w = x^R x^R)\}$. Are the context-free languages closed under *reverseanddouble*?

No. Let: $L = (a \cup b)^*$, which is regular and thus context-free.
Then: *reverseanddouble*(L) = $\{ww : w \in \{a, b\}^*\}$, which is not context-free.

9) For each of the following claims, state whether it is *True* or *False*. Prove your answer.

a) If $L = L_1^+$ and L is context-free, then L_1 must be context-free.

False. Let: $L_1 = \{a^p$: where p is prime$\}$, which is not context-free.
Then: $L = aaa^*$, which is regular and thus context-free.

b) If $L = L_1 L_2$ and L is context-free, then L_1 must be context-free.

False. Let: $L_1 = \{a^p$: where p is prime$\}$, which is not context-free.
Let: $L_2 = a^*$.
Then: $L = aaa^*$, which is regular and thus context-free.

c) If $L = L_1 \cap L_2$ and L_1 and L_2 are context-free, then L must be context-free.

False. Let: $L_1 = \{a^n b^n c^* : n \geq 0\}$, which is context-free.
Let: $L_2 = \{a^* b^n c^n : n \geq 0\}$, which is context-free.
Then: $L = \{a^n b^n c^n : n \geq 0\}$, which is notcontext-free.

d) If $L = \{w = yxx^R y : x, y \in \{a, b\}^*\}$, then $abababab \in L$.

True. Let $y = abab$ and let $x = \varepsilon$.

e) If $L = \{w = yxx^R y : x, y \in \{a, b\}^+\}$, then $ababbaab \in L$.

True. Let $y = ab$ and let $x = ab$.

10) Let $\Psi(L)$ be as defined in Section 13.7, in our discussion of Parikh's Theorem. For each of the following languages L, first state what $\Psi(L)$ is. Then give a regular language that is letter-equivalent to L.

a) $\{w \in \{a, b\}^*$: the first, middle, and last characters of w are identical$\}$.

$\Psi(L) = \{(3+i, j) : i, j \geq 0$ and $i+j$ is even$\} \cup \{(i, 3+j) : i, j \geq 0$ and $i+j$ is even$\}$.

$aaa((a \cup b)(a \cup b))^* \cup bbb((a \cup b)(a \cup b))^*$ is letter-equivalent to L.

b) $\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w \neq w^R\}$.

$\Psi(L) = \{(i, j) : i, j \geq 1\}$. Note that no string in either $\mathtt{a}^*$ or $\mathtt{b}^*$ is in L.

$(\mathtt{ab})(\mathtt{a} \cup \mathtt{b})^*$ is letter-equivalent to L.

c) $\{w : \exists x \in \{\mathtt{a}, \mathtt{b}\}^* (w = [x][x^R])\}$.

Define $\Sigma = \{[,], \mathtt{a}, \mathtt{b}\}$. $\Psi(L) = \{(2, 2, 2i, 2j) : i, j \geq 0\}$.

$[[]](\mathtt{aa} \cup \mathtt{bb})^*$ is letter-equivalent to L.

14 Decision Procedures for Context-Free Languages

1) Give a decision procedure to answer each of the following questions:
   a) Given an FSM M and a PDA P, is $L(M) \cap L(P)$ empty?

1. Use *intersectPDAandFSM* to build a PDA P^* where $L(P^*) = L(M) \cap L(P)$.
2. Invoke *PDAtoCFG*(P^*) to build a CFG G where $L(G) = L(P^*)$.
3. Return *decideCFLempty*(G).

15 Parsing

1) Let *L* be the language defined by the following grammar *G* (with *BLOCK* as the start symbol):

BLOCK → *ST*
BLOCK → *ST BLOCK*
ST → `while` *BOOL BLOCK*
ST → `do id = id to id` *BLOCK*
ST → `do` *BLOCK* `until` *BOOL*
ST → `id := id`
BOOL → `id` *OP* `id`
OP → = | ≠ | > | <

a) Apply the left factoring heuristic to *G*, creating *G'*.

We need to apply the heuristic to the rules for *BLOCK* and to two of the *ST* rules:

BLOCK → *ST BLOCK′*
BLOCK′ → ε
BLOCK′ → ε
ST → `while` *BOOL BLOCK*
ST → `do` *ST′*
ST → `id := id`
ST′ → `id = id to id` *BLOCK*
ST′ → *BLOCK* `until` *BOOL*
BOOL → `id` *OP* `id`
OP → = | ≠ | > | <

b) Use *cfgtoPDAtopdown* to construct a PDA *M* from *G'*.

[1] (*p*, ε, ε), (*q*, *BLOCK*)
[2] (*q*, ε, *BLOCK*), (*q*, *ST BLOCK′*)
[3] (*q*, ε, *BLOCK′*), (*q*, ε)
[4] (*q*, ε, *BLOCK′*), (*q*, *BLOCK*)
[5] (*q*, ε, *ST*), (*q*, `while` *BOOL BLOCK*)
[6] (*q*, ε, *ST*), (*q*, `do` *ST′*)
[7] (*q*, ε, *ST*), (*q*, `id` := `id`)
[8] (*q*, ε, *ST′*), (*q*, `id = id to id` *BLOCK*)
[9] (*q*, ε, *ST′*), (*q*, *BLOCK* `until` *BOOL*)
[10] (*q*, ε, *BOOL*), (*q*, `id` *OP* `id`)
[11] (*q*, ε, *OP*), (*q*, =)
[12] (*q*, ε, *OP*), (*q*, ≠)
[13] (*q*, ε, *OP*), (*q*, >)
[14] (*q*, ε, *OP*), (*q*, <)
[15] (*q*, `id`, `id`), (*q*, ε)
[16] (*q*, =, =), (*q*, ε)
[17] (*q*, ≠, ≠), (*q*, ε)
[18] (*q*, <, <), (*q*, ε)
[19] (*q*, >, >), (*q*, ε)
[20] (*q*, `to`, `to`), (*q*, ε)
[21] (*q*, `while`, `while`), (*q*, ε)
[22] (*q*, `do`, `do`), (*q*, ε)
[23] (*q*, `until`, `until`), (*q*, ε)
[24] (*q*, :=:, :=), (*q*, ε)

c) *M* is clearly nondeterministic. Show the places where nondeterminisim occurs. For each, indicate whether or not a one character lookahead would be able to resolve the nondeterminism.

- Rules [3] and [4] conflict. One character lookahead won't help.
- Rules [5], [6] and [7] conflict. One character lookahead solves the problem.
- Rules [8] and [9] conflict. One character lookahead does not solve the problem because *BLOCK* could begin with `id`.
- Rules [11], [12], [13] and [14] conflict. One character lookahead solves the problem.

d) Is *G* ambiguous? If not , argue why not. If so, show some string $w \in L$ that has more than one parse in *G* (and show at least two parses for it).

Yes. Let *w* be: `while id: = id id:=id id:= id`.

2) Consider the following context-free grammar *G*:

[1] $S \rightarrow D$
[2] $S \rightarrow E$
[3] $D \rightarrow \mathtt{d}D$
[4] $D \rightarrow \mathtt{d}D\mathtt{e}$
[5] $D \rightarrow \mathtt{d}F$
[6] $E \rightarrow E\mathtt{e}$
[7] $E \rightarrow \mathtt{d}E\mathtt{e}$
[8] $E \rightarrow F\mathtt{e}$
[9] $F \rightarrow \varepsilon$

a) Give a concise description of *L*(*G*).

$\{\mathtt{d}^n\mathtt{e}^m : n \neq m\}$

b) Use *removeleftrecursion* to remove the left recursion in *G* (so that it could be used for top-down parsing).

The only left recursive rule is [6]. We replace rules [6], [7], and [8] by:

$E \rightarrow \mathtt{d}E\mathtt{e}E'$
$E \rightarrow F\mathtt{e}E'$
$E' \rightarrow \mathtt{e}E''$
$E' \rightarrow \varepsilon$

3) Consider the precedence table, for an arithmetic expression grammar, shown in Section 15.3.3. Call the precedence relation that it encodes *R*.

a) What happens if we add (*T*, *) to *R*?

+ and * will have equal precedence.

b) What happens if we delete (*F*, +) from (the original) *R*?

We'll fail to parse any expression containing +.

c) What happens if we add (*F*, `id`) to (the original) *R*?

Nothing. *F* can't be followed by an `id` in the expression language, so this entry doesn't make any difference.

16 Summary and References

1) For each of the following languages L:
   (i) State whether L is regular or context-free but not regular.
   (ii) If L is not regular, prove that it is not.
   (iii) Write a grammar for L. If L is regular, write a regular grammar. If L is context-free, write a context-free grammar.

   a) $\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w = x\ (\mathtt{aa})^*\ x^R$, for some string $x \in \{\mathtt{a}, \mathtt{b}\}^*\}$.

(i) Context-free, not regular.
(ii) We show that L is not regular by using the Pumping Theorem. Let $w = \mathtt{b}^k\mathtt{aab}^k$. Then y must occur in the first b region and be $\mathtt{b}^p$ for some nonzero p. Pump in once. There are only two a's in the resulting string, so neither of them can be part of x or x^R. All the b's, on the other, hand, can only be in x and x^R. But there are more b's in the initial region than in the final one. So the final one is not the reverse of the initial one and the resulting string is not in L.
(iii) $S \rightarrow \mathtt{a}\ S\ \mathtt{a}$
$S \rightarrow \mathtt{b}\ S\ \mathtt{b}$
$S \rightarrow A$
$AA \rightarrow \mathtt{aa}\ A$
$AA \rightarrow \varepsilon$

   b) $\{w \in \{\mathtt{a}, \mathtt{b}\}^* : w = x\ (\mathtt{a} \cup \mathtt{b})^*\ x^R$, for some string $x \in \{\mathtt{a}, \mathtt{b}\}^*\}$.

(i) Regular. We can let x be ε. So L is $(\mathtt{a} \cup \mathtt{b})^*$
(ii) N/A
(iii) $S \rightarrow \mathtt{a}\ S$
$S \rightarrow \mathtt{b}\ S$
$S \rightarrow \varepsilon$

2) [Luay Nakhleh] For each of the following pairs of languages L_1 and L_2, state (with a justification) whether:
   - $L_1 \subseteq L_2$,
   - $L_2 \subseteq L_1$,
   - $L_1 = L_2$, or
   - None of the above.

   a) L_1: The language generated by the CFG whose rules are: $S \rightarrow \mathtt{0}S\mathtt{1} \mid \mathtt{1}S\mathtt{0} \mid \varepsilon$.
   L_2: The language generated by the regular expression $(\mathtt{0} \cup \mathtt{1})^*$.

$L_1 \subseteq L_2$. L_2 contains all strings over the alphabet {0, 1}. But L_1 requires equal numbers of 0's and 1's. For example, $\mathtt{0} \in L_2$ but $\mathtt{0} \notin L_1$.

   b) L_1: The language accepted by the PDA $M = (\{p, q\}, \{\mathtt{0}, \mathtt{1}\}, \{\mathtt{X}\}, q, \{p\}, \delta)$, where $\delta =$
   $\{((q, \mathtt{0}, \varepsilon), (q, \mathtt{X}))$,
   $((q, \mathtt{1}, \mathtt{X}), (p, \varepsilon))$,
   $((p, \mathtt{1}, \mathtt{X}), (p, \varepsilon))\}$

   L_2: The language generated by the CFG whose rules are: $S \rightarrow \mathtt{0}S\mathtt{1} \mid \mathtt{0}S \mid \varepsilon$.

$L_1 \subseteq L_2$. $L_1 = \{\mathtt{0}^n\mathtt{1}^n : 0 < n\}$.
$L_2 = \{\mathtt{0}^n\mathtt{1}^m : 0 \le m \le n\}$.

L_2 is not a subset of L_1 because $\varepsilon \in L_2$ but $\varepsilon \notin L_1$.

c) L_1: The language defined by the regular expression $(0 \cup 1)^* \; 11 \; (0 \cup 1)^*$.
L_2: The language defined by the regular expression $(0^* \; 1^* 11)^* \; 0^* 110^* \; 1^*$.

$L_2 \subseteq L_1$. L_1 contains all strings over the alphabet $\{0, 1\}$ that contain the substring 11 somewhere. Every string in L_2 also contains the substring 11. But there are additional constraints. So, for example, $11010 \in L_1$ but $11010 \notin L_2$.

d) L_1: The language accepted by the NDFSM $M = (\{p, q\}, \{0, 1\}, q, \{q\}, \Delta)$, where $\Delta =$
$\{((q, 0), \{p\}),$
$((q, 1), \varnothing),$
$((p, 0), \{p, q\}),$
$((p, 1), p)\}$
L_2: The language generated by the CFG whose rules are: $S \rightarrow AS \mid \varepsilon, A \rightarrow 0B, B \rightarrow 0B \mid 1B \mid 0$.

$L_1 = L_2 = \varepsilon \cup (0 \; (0 \cup 1)^* \; 0)$.

3) True or false:
a) If $L_1 \cap L_2$ is regular and L_2 is regular, then L_1 must be context free.

False. Let $L_2 = \{aa\}$. It is regular. Let $L_1 = \{a^p : p \text{ is prime}\}$. It is not context-free. But $L_1 \cap L_2 = \{aa\}$, which is regular.

b) If $L_1 \cap L_2$ is regular and L_2 is context-free, then L_1 must be context free.

False. Let $L_2 = \{aa\}$. It is regular and thus also context-free. Let $L_1 = \{a^p : p \text{ is prime}\}$. It is not context-free. But $L_1 \cap L_2 = \{aa\}$, which is regular.

c) If $L_1 \cup L_2$ is a context free language, then L_1 and L_2 must be context free.

False. Let $L_1 = \{a^p : p \text{ is prime}\}$. Let $L_2 = \{a^p : p \text{ is not prime}\}$. $L_1 \cup L_2 = a^*$, which is context-free. But neither L_1 nor L_2 is.

d) If $L_1 \subseteq L_2$ and L_2 is context-free, then L_1 must also be context-free.

False. Let $L_1 = \{a^n b^n c^n : n \geq 0\}$. Let $L_2 = \{a^n b^n c^m : n, m \geq 0\}$. $L_1 \subseteq L_2$ and L_2 is context free, but L_1 is not context-free.

e) If L is context free, then $L \cap \neg L$ must also be context free.

True. $L \cap \neg L = \varnothing$, which is regular and thus context-free.

f) If $\neg L$ is context free, then L must also be context free.

False. Let $L = A^n B^n C^n$. Then $\neg L$ is context free, but L is not.

g) If L is context free, then $L \cup \neg L$ must be regular.

True. For any L, $L \cup \neg L = \Sigma^*$, which is regular.

h) If L_1 is context-free but not regular and $L_2 \subseteq L_1$, then L_2 cannot be regular.

False. Let L_1 be $\{a^n b^n : n \geq 0\}$, which is context-free but not regular. Let L_2 be $\varnothing$, which is a subset of L_1 and is regular.

i) If L_1 is context-free and $L_2 \subset L_1$, then L_2 must be regular.

False. Let L_1 be $\{a^n b^n : n \geq 0\}$, which is context-free. Let L_2 be $\varnothing$, which is a subset of L_1 and is regular.

j) The regular languages are closed under intersection with the context-free languages.

False. Let L_1 be a^*b^*, which is regular. Let L_2 be $\{a^n b^n : n \geq 0\}$, which is context-free. $L_1 \cap L_2 = \{a^n b^n : n \geq 0\}$, which is context-free but not regular.

k) The context-free languages are closed under concatenation with the regular languages.

True. Every regular language is also context-free and the context-free languages are closed under concatenation.

l) The regular languages are closed under concatenation with the context-free languages.

False. Let L_1 be $\{\varepsilon\}$, which is regular. Let L_2 be $\{a^n b^n : n \geq 0\}$, which is context-free. $L_1 L_2 = \{a^n b^n : n \geq 0\}$, which is context-free but not regular.

m) The context-free languages are closed under union with the regular languages.

True. Every regular language is also context-free and the context-free languages are closed under union.

n) The regular languages are closed under union with the context-free languages.

False. Let L_1 be $\varnothing$, which is regular. Let L_2 be $\{a^n b^n : n \geq 0\}$, which is context-free. $L_1 \cup L_2 = \{a^n b^n : n \geq 0\}$, which is context-free but not regular.

o) If M_1 is any nondeterministic PDA, then there exists no finite state machine M_2 such that $L(M_1) = L(M_2)$.

False. Let M_1 be the NDPDA that, from its start state, has two transitions labeled a/ε/ε, one to state q_1 and one to state q_2. Make both q_1 and q_2 accepting. $L(M_1) = \{a\}$, which is regular. So there exists an FSM M_2 such that $L(M_2) = \{a\}$.

p) If M_1 is is an NDPDA with k states, then there cannot exist a DFSM M_2 with fewer than k states such that $L(M_1) = L(M_2)$.

False. M_1 could have many redundant states. Or M_1could be a PDA that doesn't use its stack. Then it can easily be transformed into an NDFSM. But then that can be transformed into an equivalent DFSM.

q) If there exists a top down, context-free parser for a language L, then there must also exist a bottom up context free parser for L.

True. If there exists a top down, context-free parser for L, then L is context-free. Any context-free language can also be parsed bottom up.

r) $\{ww^R : w \in \{a, b\}^*\}$ is accepted by some NDPDA with exactly two states.

True. This is true for any context-free language, as we saw in the proof of Theorem 12.1.

s) $(a \cup b \cup c)^* - \{a^m b^n c^p : m > n \text{ and } n > p\}$ is accepted by some NDPDA with exactly two states.

True. This is true for any context-free language, as we saw in the proof of Theorem 12.1.

t) Let M_1 be an arbitrary NDFSM. Then there exists some NDPDA M_2 such that $L(M_1) = L(M_2)$, but it is possible that there does not exist a DPDA M_3 such that $L(M_1) = L(M_3)$.

False. There exists some DFSM M_4 such that $L(M_1) = L(M_4)$. From M_4, we can build M_3. It makes transitions just as M_4 does. It ignores its stack.

u) If L_1 and L_2 are both deterministic context-free then $L_1 \cup L_2$ must also be deterministic context-free.

False. Let $L_1 = \{a^n b^n c^* : n \geq 0\}$. Let $L_2 = \{a^* b^n c^n : n \geq 0\}$. $L_1 \cup L_2$ is not deterministic context-free. Any PDA to accept it would not be able to decide whether or not to push a's.

v) $\{a^n b^n : n \geq 1\} \cup \{a^n b^{2n} : n \geq 1\}$ is deterministic context-free.

False.

dd) Let $G = \{S \rightarrow [S], S \rightarrow SS, S \rightarrow \varepsilon\}$. There exists a Chomsky normal form grammar for $L(G)$.

False. $\varepsilon \in L(G)$, yet no Chomsky normal form grammar can generate ε.

ee) The set of context-free languages that can be generated by some context-free grammar with an odd number of rules is closed under concatenation.

True. Suppose that L_1 can be generated by G_1, which contains an odd number of rules. Suppose that L_2 can be generated by G_2, which contains an odd number of rules. Then we can build G_3 to generate $L_1 L_2$. First rename the nonterminals of G_1 and G_2 so that there is no overlap between the two grammars and the symbol S isn't used. Let S_1 be the start symbol of the new G_1 and let S_2 be the start symbol of the new G_2. G_3 then contains all the rules of G_1, all the rules of G_2, and one new rule $S \rightarrow S_1 S_2$. G_3 has an odd number of rules (odd + odd + 1).

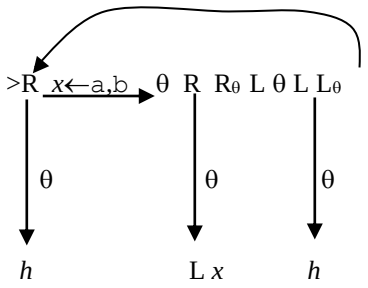
ff) The context-free languages over a single character alphabet are closed under intersection.

True. By Theorem 13.8, any context-free language over a single-character alphabet is regular. The regular languages are closed under intersection.

Part IV: Turing Machines and Undecidability

17 Turing Machines

1) Consider the following Turing Machine *M*, described in our macro language:



a) What is the result of running *M* on the input θabab?

θ

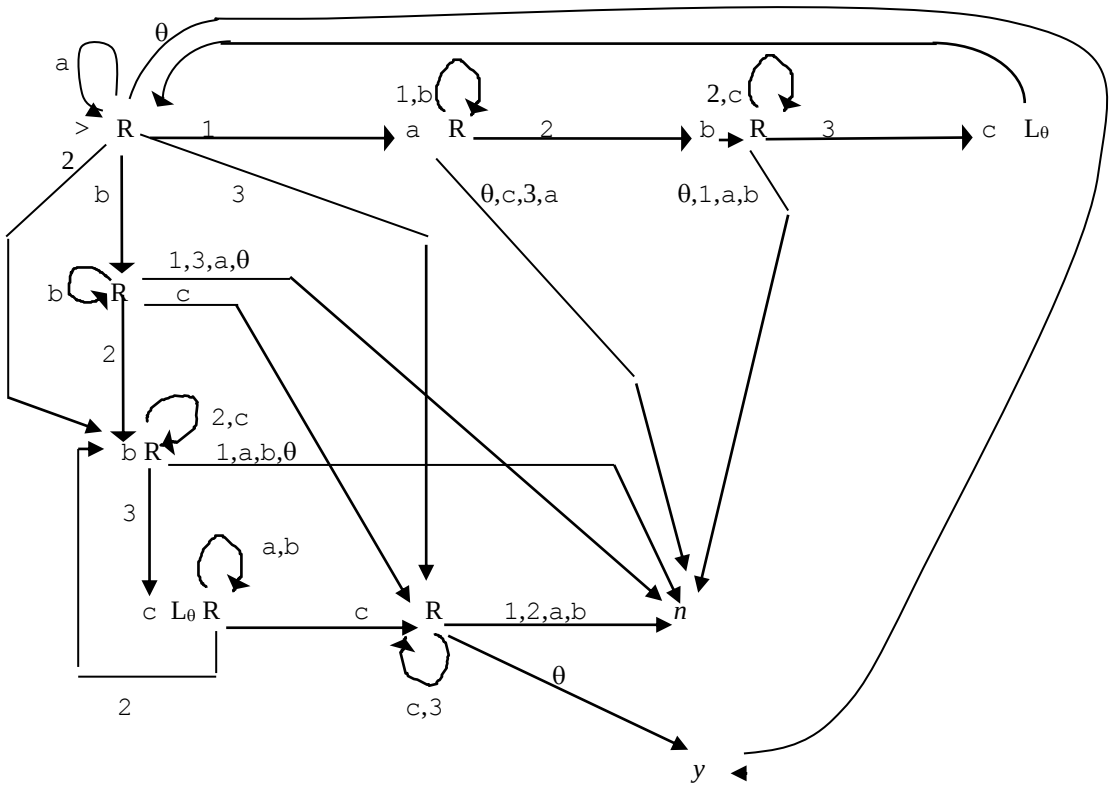
b) What is the result of running *M* on the input θabaab?

a

c) Assume that Σ_M = {a, b}. Give a short English description of what *M* does. Don't describe how it operates. Describe its result.

Blanks out the input string except that, if there is a middle character, it remains.

2) Consider the following Turing Machine *M*:

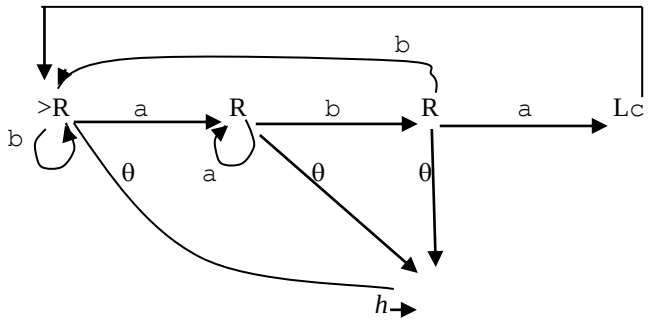


a) 122333 ∈ *L*(*M*) (a) True (b) False
b) 332211∈ *L*(*M*) (a) True (b) False
c) 112233333 ∈ *L*(*M*) (a) True (b) False
d) 11333 ∈ *L*(*M*) (a) True (b) False
e) 222111333 ∈ *L*(*M*) (a) True (b) False
f) 22233333 ∈ *L*(*M*) (a) True (b) False
g) 1223333 ∈ *L*(*M*) (a) True (b) False
h) 1212333 ∈ *L*(*M*) (a) True (b) False
i) 1122333 ∈ *L*(*M*) (a) True (b) False
j) 123123123 ∈ *L*(*M*) (a) True (b) False
k) Describe *L*(*M*).

L(*M*) = {$a^i b^j c^k$: $0 \le i \le j \le k$}.

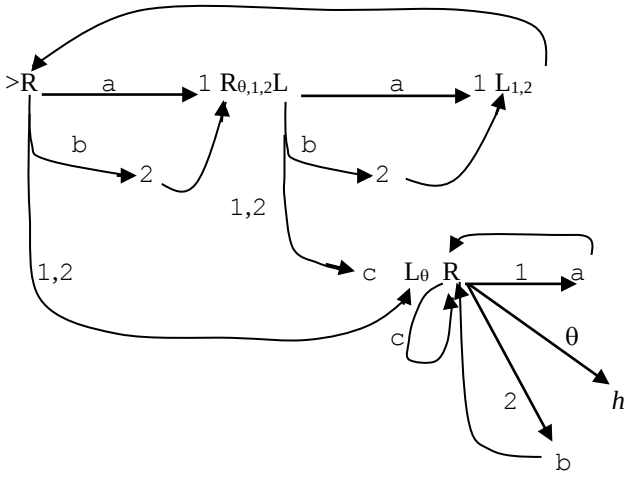
3) Give a short English description of what each of these Turing machines does:

a) Σ_M = {a, b}. *M* =



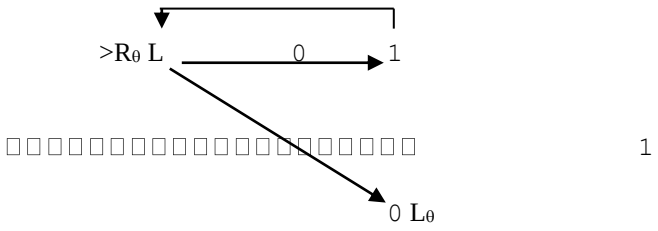
Replaces every substring of the form aba with aca.

b) Σ_M = {a, b}. *M* =



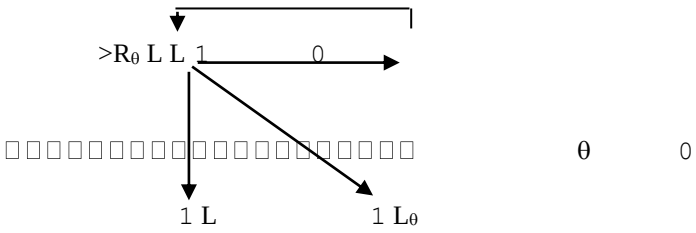
If the input string is of even length, it is unchanged. If it is of odd length, the middle character is replaced by c.

c) Assume that the input to *M* is in 1 (0 ∪ 1)*. *M* =



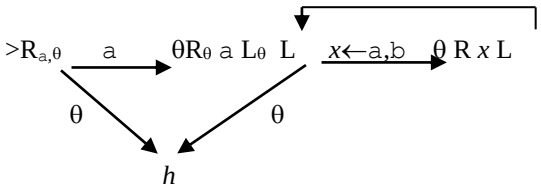
Viewing its input as a binary number, *M* subtracts 1.

d) Assume that the input to *M* is in 1 (0 ∪ 1)*. *M* =



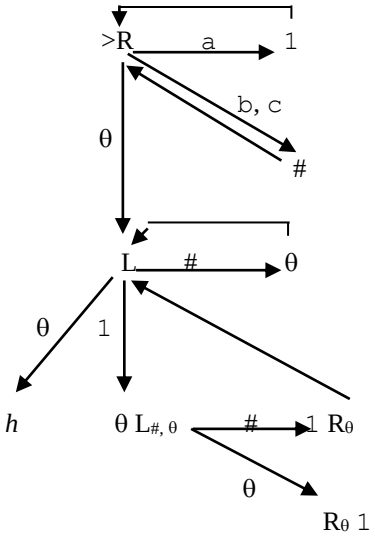
Viewing its input as a binary number, *M* adds 2.

e) Σ_M = {a, b}. *M* =



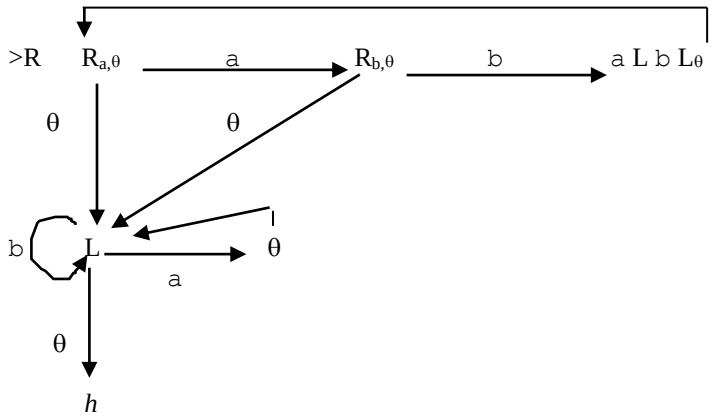
The leftmost a will be moved to the right end of the string (and the resulting hole closed up).

f) Σ_M = {a, b, c}. *M* =



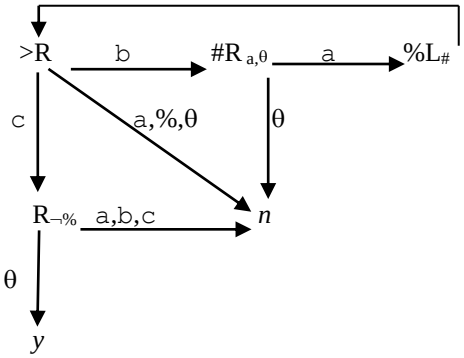
Outputs, in unary, the number of a's in its input string.

g) $\Sigma_M = \{a, b\}$. M =



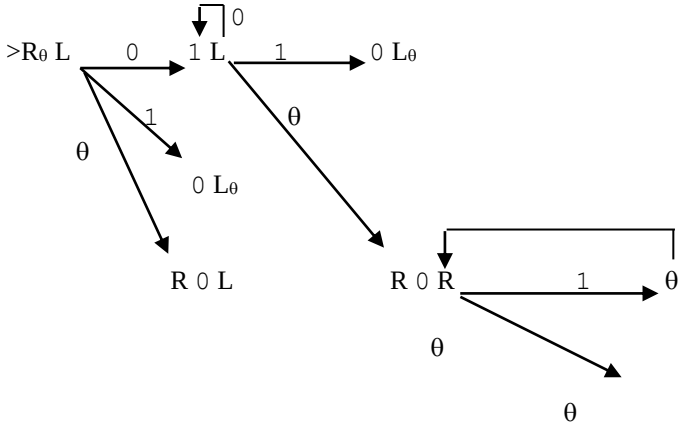
Squeezes out all the a's in its input string and moves the b's so they are all together, without holes.

4) Give a clear formal description of language accepted by each of these Turing machines:
a) $\Sigma_M = \{a, b, c\}$. M =



$\{b^k c a^k : k \geq 0\}$

5) What function does the following Turing machine *M* compute?

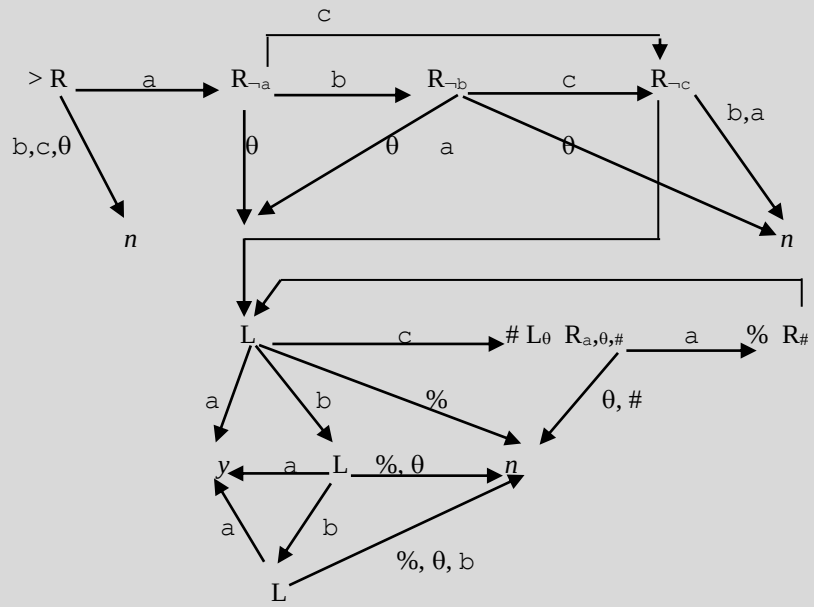


Call *M*'s input *w*. Interpret *w* the as the binary encoding of some natural number *n*. If *w* is ε, treat it as an encoding of 0. Then *M* computes:

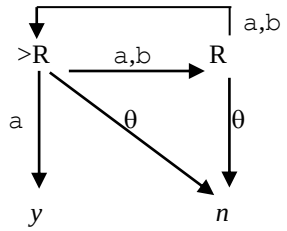
$f(n) = 0$ if n is 0
$n - 1$ otherwise

6) Using our macro language, describe a Turing machine that decides $L = \{a^i b^j c^k, i > k, 0 \le j < 3, k \ge 0\}$.

First check for form (a's then b's then c's). Then, since what we need to do is to make sure that there is an a for every c, we'll work from the right. To make the machine easy to read, there will be three states labeled *n*.



7) Consider the following TM M, where $\Sigma_M = \{a, b\}$:



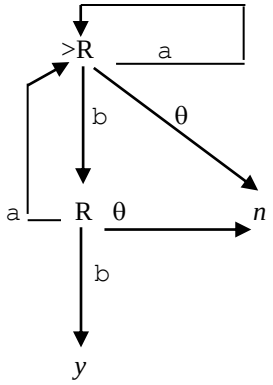
a) Give a clear, formal description of $L(M)$.

((a ∪ b) (a ∪ b))* a (a ∪ b)*

b) Suppose that we want to invoke the Universal TM U on *M* and the input string ab. Show the input to U.

First we must write compile the macro language statement of *M* into a set of transitions. Then we can encode them.

8) Consider the following TM M, where $\Sigma_M = \{a, b\}$:



a) Give a clear, formal description of $L(M)$.

(a ∪ b)* bb (a ∪ b)*

b) Suppose that we want to invoke the Universal TM U on *M* and the input string ab. Show the input to U.

First we must write compile the macro language statement of *M* into a set of transitions. Then we can encode them.

9) Suppose that *M* is a Turing machine with 6 states and a tape alphabet of 5 symbols (including θ). We're going to consider encoding *M* for input to the Universal Turing machine U. Let c be the third symbol in *M*'s tape alphabet. How would the following transition be encoded:

((3, c), (4, c, →))

(q011,a011,q100,a011,→)

10) Consider a two-tape Turing machine *M*, where $\Sigma_M = \{\theta, \texttt{a}, \texttt{b}, \texttt{c}\}$. Suppose that we want to simulate *M* with a one-tape Turing machine *T* using the technique described in Section 17.3.1. How large must Σ_T be?

$$\begin{aligned}\Sigma_T &= \Sigma_M \cup (\Sigma_M \times \{\texttt{0}, \texttt{1}\})^2 \\ |\Sigma_T| &= |\Sigma_M| + (2 \cdot |\Sigma_M|)^2 \\ &= 4 + 8^2 \\ &= 68\end{aligned}$$

11) Consider a three-tape Turing machine *M*, where $\Sigma_M = \{\theta, \texttt{a}, \texttt{b}, \texttt{c}\}$. Suppose that we want to simulate *M* with a one-tape Turing machine *T* using the technique described in Section 17.3.1. How large must Σ_T be?

$$\begin{aligned}\Sigma_T &= \Sigma_M \cup (\Sigma_M \times \{\texttt{0}, \texttt{1}\})^3 \\ |\Sigma_T| &= |\Sigma_M| + (2 \cdot |\Sigma_M|)^3 \\ &= 4 + 8^3 \\ &= 516\end{aligned}$$

18 The Church-Turing Thesis

19 The Unsolvability of the Halting Problem

20 Decidable and Semidecidable Languages

1) [Luay Nakhleh] Consider the language $L = \{<A, B> : A \text{ and } B \text{ are FSMs and } L(A) \subseteq L(B)\}$. Is L decidable?

L is in D. Note that saying that $L(A) \subseteq L(B)$ is equivalent to saying that $L(A) \cap \neg L(B)$ is empty. The following procedure exploits algorithms from Chapters 8 and 9 and decides L:

1. From B, construct (using the technique described in the proof of Theorem 8.4) a new FSM B' that accepts $\neg L(B)$.
2. From A and B', construct (using the technique described in the proof of Theorem 8.4) a new FSM F that accepts $L(A) \cap L(B')$.
3. Return *emptyFSM*(F).

21 Decidability and Undecidability Proofs

1) For each of the following languages L, state whether it is in D, in SD/D, or not in SD. Prove your answer. Assume that any input of the form <M> is a description of a Turing machine.

   a) {<M, q> : there is some configuration (p, $u\underline{a}v$) of TM M, with $p \neq q$, that yields a configuration whose state is q}.

   D. We don't need to consider the infinite number of possible configurations of M. All we need to do is to examine M's (finite) transition table δ to see whether there is any transition from some state other than q (call it p) to q. If there is such a transition (i..e., if $\exists$ p, σ, τ, D such that $\delta(q, \sigma) = (q, \tau, D)$), then the answer is yes. Otherwise, the answer is no.

   Notice that this solution worked because we did not ask whether M, when operating on some input string, ever reaches state q. That question is undecidable. We asked only whether there is some configuration (be it reachable or not) that could lead M to state q. And that question can be answered by a simple examination of δ.

   b) {<M, w> : M, on input w, begins by moving right one square onto w. Then it never moves off w}

   D. If $M = (K, \Sigma, \Gamma, \delta, s, H)$ cannot move off w, then it is operating with only a fixed length ($|w| + 1$) tape. There are $|K|$ states. There are $|\Gamma|^{|w|+1}$ distinct tape values and $|w| + 1$ squares on which the read/write had can be. So the maximum number of configurations M can enter is $max = |K| \cdot |\Gamma|^{|w|+1} \cdot |w|+1$. If M has not halted after max steps, then it is in a loop and it will just keep doing the same thing over and over. So the following algorithm decides L:

   1. Run M on w for 1 step. If it did not move right, halt and reject.
   2. If it did move right, then run M on w for max steps or until M halts or moves off the w region of the tape.
      - If M halted without moving off w, halt and accept.
      - If M moved off the w region, halt and reject.
      - If M did not halt, halt and accept (since it is just going to loop and continue not to move off w).

   c) {<M, w> : M, on input w, never writes on its tape}.

   D. If $M = (K, \Sigma, \Gamma, \delta, s, H)$ cannot write on its tape, it is effectively an FSM. More importantly, observe that there are $|K|$ states. There is only one value for the tape. If M moves off its input onto the blank region, it can count up to $|K|$ blanks past the end of w by keeping the count in its state. If it moves farther than that off the end of w (without writing anything), it is in a loop and it will continue to move without writing, since it must, a second time in state q, reading a blank, do the same thing it did the first time it was in state q reading a blank. So we need consider only $|w| + 2 \cdot |K|$ positions for M's read/write head. Let $max = |K| \cdot (|w| + 2 \cdot |K|)$. If M has not written on its tape after max steps, then it is in a loop and it will just keep doing the same thing over and over. So the following algorithm decides L:

   1. Run M on w for max steps or until M halts or writes on its tape.
      - If M halted without writing, halt and accept.
      - If M wrote on its tape, halt and reject.
      - Otherwise, halt and accept (since it is just going to loop and continue not to write on its tape).

   d) {<M, w> : M, on input w, moves left no more than once}.

   D. Without being able to move left, M cannot read what it writes on its tape. Further, if it can move only to the right (with one exception allowed), then it must eventually either halt or move off its input and onto the blank region that comes after the input on the tape. Once it does that, the only thing that can affect its future behavior is its state. It has a finite number of states. If it ever enters one a second time, it is in a loop and it will continue forever doing exactly what it has been doing. In particular, if it didn't move left during

its first time through the loop, it won't on later passes through the loop. So the following algorithm decides L:

1. Begin running M on w.
2. If, at any point, it moves left a second time, halt and reject.
3. If, at any point, it halts without moving left a second time, halt and accept.
4. If, before one of those things happens, it moves off its input then continue the simulation, but begin keeping track of the states that M enters.
5. If, without moving left, it halts, halt and accept.
6. If, without moving left, it enters a state a second time, halt and accept. It is in a loop that does not force it to move left, so it never will.
7. If it had already moved left once before it ran off its tape then it may not do so again. If it moves left even once more, halt and reject.
8. If it had not already moved left once before it ran off its tape, it may do so exactly once. If it does, start over keeping track of states. We need to find out whether the leftward move is part of a loop (and so will happen again). Continue the simulation. If M moves left again, halt and reject. If it halts without moving left, halt and accept. If it reenters a state, halt and accept. It's in a loop that doesn't cause it to move left.

e) $\{\langle M\rangle$: there exists some input on which M eventually moves right at least once$\}$. (Assume our usual convention that says that M's read/write head is initially positioned immediately to the left of its first input character.)

D. Note that the input to M is irrelevant since our decision procedure will make its decision before M has a chance to read any of its input. The following algorithm decides L:

1. Simulate M on a blank tape (or on any other fixed input string), keeping track of each (state, input character) pair that is encountered.
2. If M ever moves right, accept.
3. If M ever enters a (state, input character) configuration a second time, reject. It is in a loop in which it doesn't move right.

Notice that, eventually, this procedure must halt either in step 2 or step 3 since there is a finite number of distinct (state, input character) pairs.

f) $\{\langle M, w\rangle$: if $|w|$ is even then M halts on input w in exactly $|w|$ steps$\}$.

D. The following algorithm decides L:

1. If $|w|$ is odd, accept.
2. Else run M on w for $|w|$ steps or until it halts, whichever comes first. If it halted in exactly $|w|$ steps, accept. Else reject.

g) $\{\langle M\rangle : |\langle M\rangle| < 2\}$.

D. L must be finite and thus regular (and thus in D), since the number of strings of length < 2 over the fixed alphabet that is used to encode Turing machines is finite and every element of L must be of length < 2. In fact, using the TM encoding scheme described in the book, there are no legal TM descriptions of length less than 4. So $L = \varnothing$.

h) $\{\langle M\rangle$: M has an even number of halting states$\}$.

D. A halting state is a state with no transitions out. So a decision procedure for L simply walks through the description of M and checks whether there are any such states.

i) $\{<M_a, M_b> :$ the number of states in M_a is twice the number of states in $M_b\}$.

D. A decision procedure walks through the descriptions of the two machines and counts the number of states in each. Then it checks that the first contains twice as many states as the second.

j) $\{<M> :$ TM M halts on ε and $|<M>| < 1000\}$.

D. L is finite and thus regular (and thus in D), since the number of strings of length < 1000 over the fixed alphabet that is used to encode Turing machines is finite and every element of L must be of length < 1000.

k) [Luay Nakhleh] $\{<M> :$ M is a Turing Machine that has fewer than 100 states and that halts on input $0\}$.

D. L is finite and thus regular as well as decidable.

l) $\{<M> :$ if $|w| < 50$ then M halts on w in fewer than $2 \cdot |w|$ steps$\}$.

D. The following algorithm decides L: On input $<M>$ do:

1. Lexicographically enumerate the strings w of length < 50 in Σ_M^*. For each do:
    1.1. Run M on w for $2 \cdot |w| - 1$ steps or until it naturally halts.
    1.2. If it did not naturally halt, exit the loop and reject.
2. If all simulations halted, accept.

m) $\{<M> :$ there are more than three strings w such that M halts on w in some number of steps t and t is less than the number of income tax returns filed in the US in 2000$\}$.

D. Let r be the number of income tax returns filed in the US in 2000. The key to the fact that L is decidable is that, if M can execute only r steps, then it can look at only the first r characters of any input string. So, to check whether there are at least three strings that meet the requirement, it suffices to consider only strings of length up to r. And we need only run M on each of them for at most r steps. So a finite number of simulation steps gives us the answer. Thus the following algorithm decides L: On input $<M>$ do:

1. Lexicographically enumerate the strings w of length $\leq r$ in Σ_M^*. For each do:
    1.1. Run M on w for r steps or until it naturally halts.
2. If more than three simulations accepted, accept. Otherwise, reject.

n) $\{<M> : \forall M' ((L(M) = L(M')) \rightarrow (|<M>| > |<M'>|))\}$.

D. $L = \varnothing$. There is always an equivalent machine that has extra states and transitions and thus has a longer description.

o) $\{<M> : \exists M' (<M> \neq <M'>$ and $L(M) = L(M')\}$.

D. $L = \{<M>\}$. For every TM, there's another one that accepts the same language.

p) [Luay Nakhleh] $\{<M> : <M> \in L(M_0)$, where M_0 is a fixed Turing machine that halts on all inputs$\}$.

D. To decide whether $<M>$ is in L, all we have to do is to run M_0 on it. M_0 is guaranteed to halt. We accept if M_0 accepts and reject if it rejects.

q) [Luay Nakhleh] $\{<M, w> : \exists M_0 (w \notin L(M) \cap L(M_0))\}$.

D. L includes all strings of the form $<M, w>$. This is so because there exists an M_0 such that $L(M_0) = \varnothing$. Then, for any M, $L(M) \cap L(M_0) = \varnothing$. So no string w is in that language.

r) $\{<M> : L(M)$ contains at least one odd length string$\}$.

SD/D: The following algorithm semidecides L:

Run M on the odd length strings in Σ^* in lexicographic order, interleaving the computations. As soon as one such computation has accepted, accept.

Proof not in D: R is a reduction from H = $\{<M, w> :$ TM M halts on $w\}$ to L, defined as follows:

$R(<M, w>)$ =
1. Construct the description of $M\#(x)$ that operates as follows:
   1.1. Erase the tape.
   1.2. Write w on the tape.
   1.3. Run M.
   1.4. Accept.
2. Return *Oracle*$(<M\#>)$

If *Oracle* exists and decides L, then C = *Oracle*$(R(<M, w>))$ decides H:

- $<M, w> \in$ H: M halts on w so $M\#$ accepts everything and thus accepts at least one odd length string, so *Oracle* accepts.
- $<M, w> \notin$ H: M doesn't halt on w so $M\#$ doesn't halt and thus accepts nothing and so does not accept at least one odd length string so *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

s) $\{<M> : |L(M)| > 3\}$.

SD/D: The following algorithm semidecides L:

Run M on the strings in Σ^* in lexicographic order, interleaving the computations. As soon as four such computations have accepted, halt and accept.

Proof not in D: R is a reduction from H = $\{<M, w> :$ TM M halts on $w\}$ to L, defined as follows:

$R(<M, w>)$ =
1. Construct the description of $M\#(x)$ that operates as follows:
   1.1. Erase the tape.
   1.2. Write w on the tape.
   1.3. Run M.
   1.4. Accept.
2. Return $<M\#>$.

If *Oracle* exists and decides L, then C = *Oracle*$(R(<M, w>))$ decides H:

- $<M, w> \in$ H: M halts on w so $M\#$ accepts everything and thus accepts at least four strings, so *Oracle* accepts.
- $<M, w> \notin$ H: M doesn't halt on w so $M\#$ doesn't halt and thus accepts nothing and so does not accept at least four strings so *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

t) {<*M*, *q*> : *M* ever enters state *q* when started on an empty tape}.

SD/D: The following algorithm semidecides *L*:

Run *M* on ε, keeping track of each state this is entered. If *M* ever reaches state *q*, halt and accept.

Proof not in D: *R* is a reduction from H = {<*M*, *w*> : TM *M* halts on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M*.
   1.4. Enter the halting state *h*.
2. Return <*M*#, *h*>.

If *Oracle* exists and decides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:

- <*M*, *w*> ∈ H: *M* halts on *w* so *M*# enters state *h* on all inputs, including ε, so *Oracle* accepts.
- <*M*, *w*> ∉ H: *M* doesn't halt on *w* so *M*# doesn't halt on anything. In particular, it doesn't halt on ε and so it doesn't enter state *h*. *Oracle* rejects.

But no machine to decide *H* can exist, so neither does *Oracle*.

u) {<*M*> : |*L*(*M*)| < 3 or |*L*(*M*)| > 5}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
   1.1. Save *x*.
   1.2. Erase the tape.
   1.3. Write *w* on the tape.
   1.4. Run *M* on *w*.
   1.5. If *x* is ε, or `a` or `b`, accept; else reject.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:

- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *M*# always gets stuck in step 1.4. *M*# doesn't accept anything. |*L*(*M*#)| = 0, which is less than 3. So *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*, so *M*# makes it to step 1.5 and accepts exactly 3 strings. But that's neither less than 3 nor greater than 5. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

v) {<*M*> : |*L*(*M*)| ≤ 3}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M* on *w*.
   1.4 Accept.
Return <*M*#>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:

- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *M*# always gets stuck in step 1.3. *M*# accepts nothing. So |*L*(*M*#)| = 0, which is less than 3, so *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*, so *M*# always accepts. |*L*(*M*#)| >3. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

w) {<*M*> : *M* accepts the binary encoding of the 6th Fermat number}.

SD/D:

The following algorithm semidecides *L*: Run *M* on the binary encoding of the 6th Fermat number. If it accepts, accept.

Proof not in D: *R* is a reduction from H = {<*M*, *w*> : TM *M* halts on *w*} to *L*, defined as follows:

R(<*M*, w>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Erase the tape.
    1.2. Write *w* on the tape.
    1.3. Run *M* on *w*.
    1.4. Accept.
2. Return <*M*#>.

If *Oracle* exists and decides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:

- <*M*, *w*> ∈ H: *M* halts on *w* so *M*# accepts everything, including the string that is the binary encoding of the 6th Fermat number. *Oracle* accepts.
- <*M*, *w*> ∉ H: *M* doesn't halt on *w* so *M*# accepts nothing. *M*# does not the binary encoding of the 6th Fermat number, so *Oracle* rejects.

But no machine to decide *H* can exist, so neither does *Oracle*.

x) {<*M*> : TM *M* accepts at least one string that is the binary encoding of a prime Fermat number}. Assume the existence of a procedure *primeF*(*x*: binary number), which returns *True* if *x* is the a prime Fermat number and *False* otherwise.

SD/D:

The following algorithm semidecides *L*:

1. Create a generator by lexicographically enumerating the strings in {0, 1}+. As each string is generated, test to see whether it is the binary encoding of a prime Fermat number. If it is, then output it.
2. In interleaved mode, run *M* on the set that is generated in step 1. If *M* ever accepts one of those strings, halt and accept.

Proof not in D: *R* is a reduction from H = {<*M*, *w*> : TM *M* halts on *w*} to *L*, defined as follows:

R(<*M*, w>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Erase the tape.
    1.2. Write *w* on the tape.
    1.3. Run *M* on *w*.
    1.4. Accept.
2. Return <*M*#>.

If *Oracle* exists and decides *L*, then $C = Oracle(R(\langle M, w\rangle))$ decides H:

- $\langle M, w\rangle \in$ H: *M* halts on *w* so *M*# accepts everything. There exists at least one string (e.g., `11`) that is the binary encoding of a prime Fermat number. Since *M*# accepts everything, it accepts that string. *Oracle* accepts.
- $\langle M, w\rangle \notin$ H: *M* doesn't halt on *w* so *M*# accepts nothing. *M*# does not accept at least one string that is the binary encoding of a prime Fermat number, so *Oracle* rejects.

But no machine to decide *H* can exist, so neither does *Oracle*.

y) $\{\langle M\rangle : \texttt{ab} \in L(M)\}$.

SD/D: The following algorithm semidecides *L*:

Run *M* on `ab`. If it accepts, halt and accept.

Proof not in D: *R* is a reduction from *H* to *L* defined as follows:

$R(\langle M, w\rangle) =$

1. Construct the description $\langle M\#\rangle$ of a new Turing machine $M\#(x)$, which operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M* on *w*.
   1.4. Accept.
2. Return $\langle M\#\rangle$.

If *Oracle* exists and decides *L*, then $C = Oracle(R(\langle M, w\rangle))$ decides H:

- *R* can be implemented as a Turing machine.
- *C* is correct: *M*# halts on everything or nothing, depending on whether *M* halts on *w*. So:
  - $\langle M, w\rangle \in H$: *M* halts on *w*, so *M*# accepts all inputs, including `ab`. *Oracle* accepts.
  - $\langle M, w\rangle \notin H$: *M* does not halt on *w*, so *M*# halts on nothing. *Oracle* rejects.

But no machine to decide *H* can exist, so neither does *Oracle*.

z) $\{\langle M\rangle : \exists x\ (|x| \equiv_3 0 \text{ and } x \in L(M))\}$.

SD/D. *L* can be semidecided by the following algorithm:

Run *M* on the strings in Σ^* whose length is divisible by 3 in lexicographic order, interleaving the computations. As soon as one such computation has accepted, halt and accept.

Proof not in D: *R* is a reduction from $H = \{\langle M, w\rangle : \text{TM } M \text{ halts on } w\}$ to *L*, defined as follows:

$R(\langle M, w\rangle) =$

1. Construct the description of $M\#(x)$ that operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M*.
   1.4. Accept.
2. Return $\langle M\#\rangle$.

If *Oracle* exists and decides *L*, then $C = Oracle(R(\langle M, w\rangle))$ decides H: Note that the result of running *M*# is independent of its input. It accepts everything or nothing, depending on whether *M* halts on *w*.

- $\langle M, w\rangle \in$ H: *M* halts on *w* so *M*# accepts everything, including at least one string whose length is divisible by 3. So *Oracle* accepts.
- $\langle M, w\rangle \notin$ H: *M* doesn't halt on *w* so *M*# doesn't halt on any inputs. So it does not accept any strings whose length is divisible by 3. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

aa) $\{<M> : 1^{|<M>|} \in L(M)\}$.

SD/D. L can be semidecided by the following algorithm:

1. Run M on $1^{|<M>|}$.
2. If it halts and accepts, accept. Else loop.

Proof not in D: R is a reduction from H = $\{<M, w> :$ TM M halts on $w\}$ to L, defined as follows:

$R(<M, w>)$ =
  1. Construct the description of $M\#(x)$ that operates as follows:
    1.1. Erase the tape.
    1.2. Write w on the tape.
    1.3. Run M.
    1.4. Accept.
  2. Return $<M\#>$.

If *Oracle* exists and decides L, then C = *Oracle*($R(<M, w>)$) decides H: Note that the result of running $M\#$ is independent of its input. It accepts everything or nothing, depending on whether M halts on w.

- $<M, w> \in$ H: M halts on w so $M\#$ accepts everything, including $1^{|<M>|}$. So *Oracle* accepts.
- $<M, w> \notin$ H: M doesn't halt on w so $M\#$ doesn't halt on any inputs. So it does not accept $1^{|<M>|}$. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

bb) $\{<M> :$ M accepts both `a` and `b` and takes the same number of steps to do so in each case$\}$.

SD/D. L can be semidecided by the following algorithm:

1. Run M on `a` and count steps.
2. If step 1 halts, run M on `b` and count steps.
3. If step 2 halts, compare the two step counts. If they are the same, accept. Else reject.

Proof not in D: R is a reduction from H = $\{<M, w> :$ TM M halts on $w\}$ to L, defined as follows:

$R(<M, w>)$ =
  1. Construct the description of $M\#(x)$ that operates as follows:
    1.1. Erase the tape.
    1.2. Write w on the tape.
    1.3. Run M.
    1.4. Accept.
  2. Return $<M\#>$.

If *Oracle* exists and decides L, then C = *Oracle*($R(<M, w>)$) decides H. Note that the result of running $M\#$ is independent of its input. The number of steps it executes is a function only of the length of its input (since it must erase its tape). So it will behave identically on `a` and `b`.

- $<M, w> \in$ H: M halts on w so $M\#$ accepts everything, including `a` and `b`. It takes the same number of steps to do so in both cases. So *Oracle* accepts.
- $<M, w> \notin$ H: M doesn't halt on w so $M\#$ doesn't halt on any inputs. So it accepts neither `a` nor `b`. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

cc) $\{<M, w> :$ M halts on w and does so in some number of steps k, where $k > 2 \cdot |w|\}$.

SD/D: The following algorithm semidecides L:

Run M on w, counting the steps that are executed. If M halts and the number of steps is greater than $2 \cdot |w|$, halt and accept.

Proof not in D: R is a reduction from H = {<M, w> : TM M halts on w} to L, defined as follows:
R(<M, w>) =
1. Construct the description of M#(x) that operates as follows:
   1.1 Compute $k = 2 \cdot |x| + 1$.
   1.2 Erase the tape.
   1.2 Write w on the tape.
   1.3 Run M on w.
   1.4 Loop for k steps, then halt
2. Return <M#, w>.

If *Oracle* exists and decides L, then C = *Oracle*(R(<M, w>)) decides H. R can be implemented as a Turing machine. And C is correct. If M# halts on input x, then it takes more than $2 \cdot |x|$ steps to do so. So:

- <M, w> $\in$ H: M halts on w so M# halts on everything, including w. It takes more than $2 \cdot |w|$ steps to do so. So *Oracle* accepts.
- <M, w> $\notin$ H: M doesn't halt on w so M# halts on nothing. In particular, it does not halt on w. So *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

dd) {<M_a, M_b> : if M_a does not accept ε then M_b rejects ε}.

SD/D: Note that L is equivalent to {<M_a, M_b> : M_a accepts ε or M_b rejects ε}.

The following algorithm semidecides L:

Run both machines on ε in parallel. If either M_a accepts or M_b rejects, halt and accept.

Proof not in D: R is a reduction from H = {<M, w> : TM M halts on w} to L, defined as follows:
R(<M, w>) =
1. Construct the description of M#(x) that, on any input, immediately accepts.
2. Construct the description of M##(x) that operates as follows:
   2.1 Erase the tape.
   2.2 Write w on the tape.
   2.3 Run M on w.
   2.4 Accept.
3. Return <M##, M#>.

If *Oracle* exists and decides L, then C = *Oracle*(R(<M, w>)) decides H. R can be implemented as a Turing machine. And C is correct. Note that M# rejects nothing. M## accepts everything or nothing, depending on whether M halts on w. So:

- <M, w> $\in$ H: M halts on w so M## accepts everything, including ε. So *Oracle* accepts.
- <M, w> $\notin$ H: M doesn't halt on w so M## accepts nothing, including ε. M# rejects nothing. So *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

ee) {<M, n> : M accepts at least one string of length > n}.

SD/D: The following algorithm semidecides L:

Run M on the strings in Σ^* of length > n in lexicographic order, interleaving the computations. As soon as one such computation has accepted, halt and accept.

Proof not in D: R is a reduction from H to L defined as follows:

$R(<M, w>)$ =

1. Construct the description $<M\#>$ of a new Turing machine $M\#(x)$, which that operates as follows:
   1.1. Erase the tape.
   1.2. Write w on the tape.
   1.3. Run M on w.
   1.4. Accept.
2. Return $<M\#, 1>$.

If *Oracle* exists and decides L, then $C = Oracle(R(<M, w>))$ decides H:

- R can be implemented as a Turing machine.
- C is correct: $M\#$ halts on everything or nothing, depending on whether M halts on w. So:
  - $<M, w> \in H$: M halts on w, so $M\#$ accepts all inputs, including at least one string of length 1. *Oracle* accepts.
  - $<M, w> \notin H$: M does not halt on w, so $M\#$ halts on nothing. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

ff) $\{<M_a, M_b> : \varepsilon \in L(M_a) \cup L(M_b)\}$

SD/D. The following algorithm semidecides L:

Run M_a and M_b in parallel on on ε. As soon as one of the computations accepts, halt and accept.

Proof not in D: R is a reduction from H to L defined as follows:

$R(<M, w>)$ =

1. Construct the description $<M\#>$ of a new Turing machine $M\#(x)$, which operates as follows:
   1.1. Erase the tape.
   1.2. Write w on the tape.
   1.3. Run M on w.
   1.4. Accept.
2. Construct the description $<M\#\#>$ of a new Turing machine $M\#\#(x)$ that operates as follows:
   2.1. Reject.
3. Return $<M\#, M\#\#>$.

If *Oracle* exists and decides L, then $C = Oracle(R(<M, w>))$ decides H:

- R can be implemented as a Turing machine.
- C is correct: $M\#$ accepts everything or nothing, depending on whether M halts on w. So:
  - $<M, w> \in H$: M halts on w, so $M\#$ accepts all inputs, including ε. Since at least one of $M\#$ and $M\#\#$ accepts ε, *Oracle* accepts.
  - $<M, w> \notin H$: M does not halt on w, so $M\#$ accepts nothing. $M\#\#$ also accepts nothing. So ε is accepted by neither of them. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

gg) $\{<M_a, M_b> : \varepsilon \in L(M_a) \cap L(M_b)\}$

SD/D. The following algorithm semidecides L:

Run M_a on ε. If it accepts, run M_b on ε. If it accepts, halt and accept.

Proof not in D: R is a reduction from H to L defined as follows:

$R(<M, w>)$ =

1. Construct the description $<M\#>$ of a new Turing machine $M\#(x)$, which operates as follows:
   1.1. Erase the tape.
   1.2. Write w on the tape.
   1.3. Run M on w.
   1.4. Accept.
2. Construct the description $<M\#\#>$ of a new Turing machine $M\#\#(x)$, which operates as follows:
   2.1. Accept.
3. Return $<M\#, M\#\#>$.

If *Oracle* exists and decides L, then $C = Oracle(R(<M, w>))$ decides H:

- R can be implemented as a Turing machine.
- C is correct: $M\#$ accepts everything or nothing, depending on whether M halts on w. $M\#\#$ accepts everything. So:
  - $<M, w> \in H$: M halts on w, so $M\#$ accepts everything. So does $M\#\#$. So they both accept ε and *Oracle* accepts.
  - $<M, w> \notin H$: M does not halt on w, so $M\#$ accepts nothing. So $L(M\#) \cap L(M\#\#) = \varnothing$,which does not contain ε. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

hh) [Don Baker] $\{<M> : M$ rejects at least one of `aabb` or `bbaa`$\}$.

SD/D: The following algorithm semidecides L:

Run M on `aabb` and `bbaa`, interleaving the computations. As soon as one such computation has rejected, halt and accept.

Proof not in D: R is a reduction from H to L defined as follows:

$R(<M, w>)$ =

1. Construct the description $<M\#>$ of a new Turing machine $M\#(x)$ that operates as follows:
   1.1. Erase the tape.
   1.2. Write w on the tape.
   1.3. Run M on w.
   1.4. Reject.
2. Return $<M\#>$.

If *Oracle* exists and decides L, then $C = Oracle(R(<M, w>))$ decides H:

- R can be implemented as a Turing machine.
- C is correct: $M\#$ rejects everything or nothing, depending on whether M halts on w. So:
  - $<M, w> \in H$: M halts on w, so $M\#$ rejects all inputs, including `aabb` and `bbaa`. *Oracle* accepts.
  - $<M, w> \notin H$: M does not halt on w, so $M\#$ rejects nothing. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

ii) $\{<M> : M$ accepts the string that corresponds to the binary encoding of $|<M>|\}$.

SD/D: The following algorithm semidecides L:

Run M on the string that corresponds to the binary encoding of $|<M>|$. If it accepts, halt and accept.

Proof not in D: *R* is a reduction from *H* to *L* defined as follows:

R(<*M*, *w*>) =

1. Construct the description <*M*#> of a new Turing machine *M*#(*x*), which that operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M* on *w*.
   1.4. Accept.
2. Return <*M*#>.

If *Oracle* exists and decides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:

- *R* can be implemented as a Turing machine.
- *C* is correct: *M*# halts on everything or nothing, depending on whether *M* halts on *w*. So:
  - <*M*, *w*> ∈ *H*: *M* halts on *w*, so *M*# accepts all inputs, including the string that corresponds to the binary encoding of |<*M*>|. *Oracle* accepts.
  - <*M*, *w*> ∉ *H*: *M* does not halt on *w*, so *M*# halts on nothing. *Oracle* rejects.

But no machine to decide *H* can exist, so neither does *Oracle*.

jj) {<*M*> : *M* does not accept the string that corresponds to the binary encoding of |<*M*>|}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =

1. Construct the description of *M*#(*x*) that operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M* on *w*.
   1.4. Accept.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:

- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *M*# always gets stuck in step 1.3. *M*# doesn't accept anything. So, in particular, it fails to accept the string that corresponds to the binary encoding of |<*M*>|. So *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*, so *M*# always accepts. So it accepts the string that corresponds to the binary encoding of |<*M*>|. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

kk) {<*M*> : *M* fails to accept at least one of `aabb` or `bbaa`}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =

1. Construct the description of *M*#(*x*) that operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M* on *w*.
   1.4. Accept.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:

- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *M*# always gets stuck in step 1.3. *M*# doesn't accept anything. So, in particular, it fails to accept both `aabb` and `bbaa`. So *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*, so *M*# always accepts. So it accepts both `aabb` and `bbaa`. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

ll) {<*M*> : *M* rejects at most two even length strings}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Erase the tape.
    1.2. Write *w* on the tape.
    1.3. Run *M* on *w*.
    1.4. Reject.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:
- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *M*# always gets stuck in step 1.3. So it rejects nothing. 0 is not more than 2, so *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*, so *M*# always rejects. So it rejects more than two even length strings. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

mm) {<*M*> : when the strings in *L*(*M*) are listed in lexicographic order, the third such string is `bbbb`}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. If *x* is ε or `a` or `bbbb`, accept.
    1.2. Erase the tape.
    1.3. Write *w* on the tape.
    1.4. Run *M* on *w*.
    1.5 Accept.
Return <*M*#>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:
- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *M*# accepts just the three strings ε, `a` and `bbbb`. Thus `bbbb` is the third string in the lexicographic enumeration of *L*(*M*#). So *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*, so *M*# always accepts. Since its input alphabet includes `a` and `b`, it accepts, among other things, `aa`. So `bbbb` is not the third string in the lexicographic enumeration of *L*(*M*#). So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

nn) {<*M*> : *M* does not accept any string *w* such that `000` is a prefix of *w*}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Erase the tape.
    1.2. Write *w* on the tape.
    1.3. Run *M* on *w*.
    1.4 Accept.
Return <*M*#>.

If *Oracle* exists and semidecides L, then C = *Oracle*(R(<M, w>)) semidecides ¬H:

- <M, w> ∈ ¬H: M does not halt on w, so M# always gets stuck in step 1.3. M# accepts nothing, and, in particular, no string that has `000` as a prefix. So *Oracle* accepts.
- <M, w> ∉ ¬H: M halts on w, so M# always accepts. In particular, it accepts strings with `000` as a prefix. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

oo) {<M> : M accepts every string in $\mathtt{a}^+$ and rejects every string in $\mathtt{b}^+$}

¬SD: R is a reduction from ¬H = {<M, w> : TM M does not halt on w} to L, defined as follows:
R(<M, w>) =
1. Construct the description of M#(x) that operates as follows:
   1.1. If $x \in \mathtt{a}^+$ then accept.
   1.2. Save its input x on a second tape.
   1.3. Erase the tape.
   1.4. Write w on the tape.
   1.5. Run M on w for $|x|$ steps or until it halts.
   1.6. If M would have halted during the simulation, then loop,
   1.7. Reject.
2. Return <M#>.

If *Oracle* exists and semidecides L, then C = *Oracle*(R(<M, w>)) semidecides ¬H:

- <M, w> ∈ ¬H: M# accepts all strings in $\mathtt{a}^+$. For all others: M does not halt on w so M# always makes it to step 1.7 and rejects everything, including $\mathtt{b}^+$. So *Oracle* accepts.
- <M, w> ∉ ¬H: M# accepts all strings in $\mathtt{a}^+$. For all others: M halts on w in some number of steps we can call k. So M# rejects all strings of length less than k. But, on all strings of length k or greater M# loops. Since there are arbitrarily long strings in $\mathtt{b}^+$, M# fails to reject some strings in $\mathtt{b}^+$. *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

pp) {<M> : both L(M) and ¬L(M) are infinite}.

¬SD. R is a reduction from ¬H = {<M, w> : TM M does not halt on w} to L, defined as follows:
R(<M, w>) =
1. Construct the description of M#(s) that operates as follows:
   1.1. If $s \in \{x \in \mathtt{a}^*: |x| \text{ is even}\}$, accept.
   1.2. Erase the tape.
   1.3. Write w.
   1.4. Run M on w.
   1.5. Accept.
2. Return <M#>.

If *Oracle* exists and semidecides L, then C = *Oracle*(R(<M, w>)) semidecides ¬H:

- <M, w> ∈ ¬H: does not halt on w. M# accepts $\{x \in \mathtt{a}^*: |x| \text{ is even}\}$ but gets stuck on every other input. So L(M#) = $\{x \in \mathtt{a}^*: |x| \text{ is even}\}$, which is infinite. ¬$L$($M$) = {$w$: w contains some character other than `a` or $|w|$ is odd}, which is also infinite. So *Oracle* accepts.
- <M, w> ∉ ¬H: M halts on w. So M# accepts everything. L(M#) = Σ*, which is infinite. But ¬L(M) = ∅, which is finite. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

qq) {<*M*> : *L*(*M*) is not regular}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Save its input *x* on a second tape.
    1.2. Erase the tape.
    1.3. Write *w*.
    1.4. Run *M* on *w* for |*x*| steps or until it halts.
    1.5. If *M* would have halted, then loop.
    1.6. Else: if $x \in a^n b^n$, accept. Otherwise reject.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:
- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *M*# always gets to step 1.6. So $L(M\#) = a^n b^n$, which isn't regular. *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*. Suppose it does so in *k* steps. Then, for all strings of length *k* or more, *M*# loops at step 1.5. So *L*(*M*#) is finite and thus regular. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

rr) {<*M*> : *L*(*M*) is decidable}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Save its input *x* on a second tape.
    1.2. Erase the tape.
    1.3. Write *w*.
    1.4. Run *M* on *w*.
    1.5. See whether *x* ∈ H by doing:
        1.5.1. Check the syntax of *x* to see if it's of the form <*M′*, *w′*>. If not, reject.
        1.5.2. Run *M′* on *w′*.
        1.5.3. Accept.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H. Note that, if *M*# makes it through the gate at step 1.4, then the language that it accepts is H (which is not decidable). If it fails to make it through the gate, then the language that it accepts is ∅, which is decidable. So:
- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *L*(*M*#) = ∅, which is decidable. *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*. So *L*(*M*#) = H, which is not decidable. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

ss) {<*M*> : TM *M* accepts the binary representations of all even numbers}.

¬SD: Assume that Σ ≠ ∅. *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Save its input *x* on a second tape.
    1.2. Erase the tape.
    1.3. Write *w*.
    1.4. Run *M* on *w* for |*x*| steps or until it halts.
    1.5. If *M* would have halted, then loop.
    1.6. Else accept.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then $C = Oracle(R(\langle M, w\rangle))$ semidecides ¬H:

- $\langle M, w\rangle \in$ ¬H: *M* does not halt on *w*, so *M*# always gets to step 1.6. So it accepts everything, including the binary representations of all even numbers, so *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: *M* halts on *w*. Suppose it does so in *k* steps. Then, for all strings of length *k* or more, *M*# loops at step 1.5. For any *k*, there is a binary string of length greater than *k* that represents an even number. So it is not true that *M*# accepts all binary representations of even numbers. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

tt) $\{\langle M\rangle : \forall w\ (|w| \text{ is prime} \rightarrow w \in L(M)\}$.

¬SD: Assume that $\Sigma \neq \varnothing$. *R* is a reduction from ¬H = $\{\langle M, w\rangle$: TM *M* does not halt on *w*$\}$ to *L*, defined as follows:

R($\langle M, w\rangle$) =
1. Construct the description of *M*#(*x*) that operates as follows:
   1.1. Save its input *x* on a second tape.
   1.2. Erase the tape.
   1.3. Write *w*.
   1.4. Run *M* on *w* for |*x*| steps or until it halts.
   1.5. If *M* would have halted, then loop.
   1.6. Else accept.
2. Return $\langle M\#\rangle$.

If *Oracle* exists and semidecides *L*, then $C = Oracle(R(\langle M, w\rangle))$ semidecides ¬H:

- $\langle M, w\rangle \in$ ¬H: *M* does not halt on *w*, so *M*# always gets to step 1.6. So it accepts everything, including all strings with length that is prime. So *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: *M* halts on *w*. Suppose it does so in *k* steps. Then, for all strings of length *k* or more, *M*# loops at step 1.5. For any *k*, there is a prime number greater than *k*. And there are strings whose length is that number. So it is not true that *M*# accepts all strings whose length is prime. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

uu) [Don Baker] $\{\langle M\rangle$: TM *M* does not halt on any input that is the binary encoding of a prime number$\}$.

¬SD: *R* is a reduction from ¬H = $\{\langle M, w\rangle$: TM *M* does not halt on *w*$\}$ to *L*, defined as follows:

R($\langle M, w\rangle$) =
1. Construct the description of *M*# that operates as follows:
   1.1. Erase the tape.
   1.2. Write *w*.
   1.3. Run *M* on *w*.
2. Return $\langle M\#\rangle$.

If *Oracle* exists and semidecides *L*, then $C = Oracle(R(\langle M, w\rangle))$ semidecides ¬H:

- $\langle M, w\rangle \in$ ¬H: *M* does not halt on *w*, so *M*# halts on nothing. So, in particular, it does not halt on any input that is the binary encoding of a prime number. So *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: *M* halts on *w*, so *M*# halts on everything, including an infinite number of binary encodings of prime numbers. So *Oracle* does not accept.

But *R* cannot exist, so neither does *Oracle*.

vv) {<*M*> : There exists at least one prime number whose encoding as a binary string is not accepted by *M*}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Erase the tape.
    1.2. Write *w*.
    1.3. Run *M* on *w*.
    1.4. Accept
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:
- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *M*# gets stuck in step 1.3. So it accepts nothing. Thus it fails to accept the binary encodings of every prime number. *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*. *M*# accepts everything, including the binary string encodings of all prime numbers. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

ww){<*M*> : TM *M* halts on all strings of the form 1^p, where *p* is prime}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Save its input *x* on a second tape.
    1.2. Erase the tape.
    1.3. Write *w*.
    1.4. Run *M* on *w* for |*x*| steps or until it naturally halts.
    1.5. If *M* would have halted naturally, then loop.
    1.6. Else accept.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:
- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *M*# always gets to step 1.6. So it accepts everything, including all strings of the form 1^p, where *p* is prime. *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*. Suppose it does so in *k* steps. Then, for all strings of length *k* or more, *M*# loops at step 1.5. For any *k*, there is a prime number *p* greater than *k* and thus a string of the form 1^p, where *p* is prime, that *M*# fails to accept. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

xx) {<*M*> : TM *M* halts on all strings *s*, where *s* is the binary encoding of a prime number greater than 5}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Save its input *x* on a second tape.
    1.2. Erase the tape.
    1.3. Write *w*.
    1.4. Run *M* on *w* for |*x*| steps or until it halts.
    1.5. If *M* would have halted, then loop.
    1.6. Else accept.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then $C = Oracle(R(\langle M, w\rangle))$ semidecides ¬H:

- $\langle M, w\rangle \in$ ¬H: *M* does not halt on *w*, so *M*# always gets to step 1.6. So it accepts everything, including all strings that represent the binary encoding of a prime number greater than 5, so *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: *M* halts on *w*. Suppose it does so in *k* steps. Then, for all strings of length *k* or more, *M*# loops at step 1.5. For any *k*, there is a prime number greater than both 5 and *k* and thus a binary string that encodes it. So *M*# fails to accept all such strings. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

yy) {<*M*> : ∀*w* (if *w* is the binary encoding of some integer *n*, then: if *M* halts on *w* then *M* also halts on the binary encoding of 2*n*)}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Save its input *x* on a second tape.
    1.2. Erase the tape.
    1.3. Write *w*.
    1.4. Run *M* on *w*.
    1.5. If *x* = `1`, then halt, else loop.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then $C = Oracle(R(\langle M, w\rangle))$ semidecides ¬H. Note that we can rewrite *L* as: {<*M*> : ∀*w* (if *w* is the binary encoding of some integer *n*, then: *M* does not halt on *w* or *M* does halt on the binary encoding of 2*n*)}.

- $\langle M, w\rangle \in$ ¬H: *M* does not halt on *w*, so *M*# always gets stuck in step 1.4. It halts on nothing. So, for all strings that are binary encodings of integers, the first condition is satisfied. The second doesn't matter. *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: *M* halts on *w*. *M*# always makes it to step 1.5. Consider what happens when *x* is the string `1`. It is the binary encoding of the integer 1. *M*# halts on it. So, if *M*# is to be in *L*, it must also halt on the string `10`. But it doesn't. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

zz) {<*M*> : *M* halts on all strings of the form 1^k, where $k = 2^i$ for some integer $i \geq 0$}. So, for example, if *M* is in *L*, then *M* halts on `1111` and `11111111`.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Save its input *x* on a second tape.
    1.2. Erase the tape.
    1.3. Write *w*.
    1.4. Run *M* on *w* for |*x*| steps.
    1.5. If *M* would have halted, then loop.
    1.6. Else halt.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then $C = Oracle(R(\langle M, w\rangle))$ semidecides ¬H:

- $\langle M, w\rangle \in$ ¬H: *M* does not halt on *w*, so *M*# always gets to step 1.6. So it halts on everything, including all strings of the form 1^k, where $k = 2^i$ for some integer $i \geq 0$. So *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: *M* halts on *w*. Suppose it does so in *n* steps. Then, for all strings of length *n* or more, *M*# loops at step 1.5. For any *n*, there is a string of the form 1^k, where $k = 2^i$ and $k > n$. So *M*# fails to accept all such strings. So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

aaa) $\{<M_a, M_b> : L(M_a) \cap L(M_b) = \varnothing\}$

¬SD: R is a reduction from ¬H = $\{<M, w> :$ TM M does not halt on $w\}$ to L, defined as follows:
$R(<M, w>)$ =
1. Construct the description of $M\#$ that operates as follows:
    1.1. Erase the tape.
    1.2. Write w.
    1.3. Run M on w.
    1.4. Accept.
2. Construct the description of $M\#\#$ that operates as follows:
    2.1. Accept.
3. Return $<M\#, M\#\#>$.

If *Oracle* exists and semidecides L, then C = *Oracle*($R(<M, w>)$) semidecides ¬H:
- $<M, w> \in$ ¬H: M does not halt on w, so $L(M\#) = \varnothing$. $L(M\#) \cap L(M\#\#) = \varnothing$, so *Oracle* accepts.
- $<M, w> \notin$ ¬H: M halts on w, so $L(M\#) = \Sigma^* = L(M\#\#)$. $L(M\# \cap L(M\#\#) = \Sigma^*$ and thus not $\varnothing$. So *Oracle* does not accept.

But R cannot exist, so neither does *Oracle*.

bbb) [Luay Nakhleh] $\{<M_a, M_b> : |L(M_a)| > |L(M_b)|\}$.

¬SD: R is a reduction from ¬H = $\{<M, w> :$ TM M does not halt on $w\}$ to L, defined as follows:
$R(<M, w>)$ =
1. Construct the description of $M\#$ that operates as follows:
    1.1. Erase the tape.
    1.2. Write w.
    1.3. Run M on w.
    1.4. Accept.
2. Construct the description of $M\#\#$ that operates as follows:
    2.1. Accept.

Return $<M\#\#, M\#>$.

If *Oracle* exists and semidecides L, then C = *Oracle*($R(<M, w>)$) semidecides ¬H:
- $<M, w> \in$ ¬H: M does not halt on w, so $L(M\#) = \varnothing$. $L(M\#\#) = \Sigma^*$. $M\#\#$ accepts more strings than $M\#$ does, so *Oracle* accepts.
- $<M, w> \notin$ ¬H: M halts on w, so $L(M\#) = \Sigma^* = L(M\#\#)$. $M\#\#$ does not accept more strings than $M\#$ does. So *Oracle* does not accept.

But R cannot exist, so neither does *Oracle*.

ccc) $\{<M_a, M_b> : |L(M_a)| = 2 \cdot |L(M_b)|\}$.

¬SD: R is a reduction from ¬H = $\{<M, w> :$ TM M does not halt on $w\}$ to L, defined as follows:
$R(<M, w>)$ =
1. Construct the description of $M\#$ that operates as follows:
    1.1. Erase the tape.
    1.2. Write w.
    1.3. Run M on w.
    1.4. Accept.
2. Construct the description of $M\#\#$ that operates as follows:
    2.1. Reject.

Return $<M\#, M\#\#>$.

If *Oracle* exists and semidecides L, then $C = Oracle(R(\langle M, w\rangle))$ semidecides ¬H:

- $\langle M, w\rangle \in$ ¬H: M does not halt on w, so $L(M\#) = \varnothing$. $L(M\#\#) = \varnothing$. Each of them has cardinality 0. $2 \cdot 0 = 0$, so *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: M halts on w, so $L(M\#) = \Sigma^*$, so it is uncountably infinite. $\aleph_0 \neq 2 \cdot 0$. So *Oracle* does not accept.

But R cannot exist, so neither does *Oracle*.

ddd) $\{\langle M_a, M_b\rangle : M_a$ accepts exactly one string that M_b does not accept$\}$.

¬SD: R is a reduction from ¬H = $\{\langle M, w\rangle$: TM M does not halt on $w\}$ to L, defined as follows:

$R(\langle M, w\rangle)$ =
1. Construct the description of $M\#(x)$ that operates as follows on input x:
    1.1. If $x = \varepsilon$ then accept. Else:
    1.2. Erase the tape.
    1.3. Write w.
    1.4. Run M on w.
    1.5. Accept.
2. Construct the description of $M\#\#$ that operates as follows:
    2.1. Reject.

Return $\langle M\#, M\#\#\rangle$.

If *Oracle* exists and semidecides L, then $C = Oracle(R(\langle M, w\rangle))$ semidecides ¬H:

- $\langle M, w\rangle \in$ ¬H: M does not halt on w, so $L(M\#) = \{\varepsilon\}$. $L(M\#\#) = \varnothing$. $M\#$ accepts exactly one string (ε) that $M\#\#$ does not accept. So *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: M halts on w, so $L(M\#) = \Sigma^*$. So $M\#$ accepts an infinite number of strings that are not accepted by $M\#\#$. So *Oracle* does not accept.

But R cannot exist, so neither does *Oracle*.

eee) $\{\langle M_a, M_b, M_c\rangle : L(M_a) = L(M_b) \cup L(M_c)\}$.

¬SD: R is a reduction from ¬H = $\{\langle M, w\rangle$: TM M does not halt on $w\}$ to L, defined as follows:

$R(\langle M, w\rangle)$ =
1. Construct the description of $M\#$ that operates as follows:
    1.1. Erase the tape.
    1.2. Write w.
    1.3. Run M on w.
    1.4. Accept.
2. Construct the description of $M\#\#$ that operates as follows:
    2.1. Loop.
3. Construct the description of $M\#\#\#$ that operates as follows:
    2.1. Loop.
4. Return $\langle M\#, M\#\#, M\#\#\#\rangle$.

If *Oracle* exists and semidecides L, then $C = Oracle(R(\langle M, w\rangle))$ semidecides ¬H. Note that $L(M\#\#) \cup L(M\#\#\#) = \varnothing$.

- $\langle M, w\rangle \in$ ¬H: M does not halt on w, so $L(M\#) = \varnothing$. So $L(M\#) = L(M\#\#) \cup L(M\#\#\#)$, so *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: M halts on w, so $L(M\#) = \Sigma^*$. So $L(M\#) \neq L(M\#\#) \cup L(M\#\#\#)$. So *Oracle* does not accept.

But R cannot exist, so neither does *Oracle*.

fff) $\{<M_a, M_b, M_c> : L(M_c) = L(M_a) \cap L(M_b)\}$.

¬SD: R is a reduction from ¬H = $\{<M, w> :$ TM M does not halt on $w\}$ to L, defined as follows:
$R(<M, w>)$ =
1. Construct the description of $M\#(x)$ that operates as follows:
    1.1. Erase the tape.
    1.2. Write w.
    1.3. Run M on w.
    1.4. Accept.
2. Construct the description of $M\#\#(x)$ that operates as follows:
    2.1. Loop.
3. Construct the description of $M\#\#\#(x)$ that operates as follows:
    2.1. Loop.
4. Return $<M\#\#, M\#\#\#, M\#>$.

W3e3If *Oracle* exists and semidecides L, then C = *Oracle*($R(<M, w>)$) semidecides ¬H. Note that $L(M\#\#) \cap L(M\#\#\#) = \varnothing$.
- $<M, w> \in$ ¬H: M does not halt on w, so $L(M\#) = \varnothing$. So $L(M\#) = L(M\#\#) \cap L(M\#\#\#)$. *Oracle* accepts.
- $<M, w> \notin$ ¬H: M halts on w, so $L(M\#) = \Sigma^*$. So $L(M\#) \neq L(M\#\#) \cap L(M\#\#\#)$. *Oracle* does not accept.

But R cannot exist, so neither does *Oracle*.

ggg) $\{<M> :$ M views its input as a binary encoding of an integer and the only prime number it accepts is 2$\}$.

¬SD: R is a reduction from ¬H = $\{<M, w> :$ TM M does not halt on $w\}$ to L, defined as follows:
$R(<M, w>)$ =
1. Construct the description of $M\#(x)$ that operates as follows:
    1.1. If x is the binary encoding of 2, accept.
    1.2. Erase the tape.
    1.3. Write w.
    1.4. Run M on w.
    1.5. Accept.
2. Return $<M\#>$.

If *Oracle* exists and semidecides L, then C = *Oracle*($R(<M, w>)$) semidecides ¬H. Note that $M\#$ accepts 2, no matter what. At issue, then, is whether or not it accepts anything else.
- $<M, w> \in$ ¬H: M does not halt on w, so, on any input other than 2, $M\#$ loops. $L(M\#) = \{2\}$. *Oracle* accepts.
- $<M, w> \notin$ ¬H: M halts on w, so $L(M\#) = \Sigma^*$. Since there exist prime numbers other than 2 and they will be accepted, *Oracle* does not accept.

But R cannot exist, so neither does *Oracle*.

hhh) $\{<M> :$ M accepts all strings that start with `a` and accepts no strings that start with `b`$\}$.

¬SD. R is a reduction from ¬H = $\{<M, w> :$ TM M does not halt on $w\}$ to L, defined as follows:
$R(<M, w>)$ =
1. Construct the description of $M\#(x)$ that operates as follows:
    1.1. If the first character of x is `a`, accept. Else:
    1.2. Erase the tape.
    1.3. Write w.
    1.4. Run M on w.
    1.5. Accept.
2. Return $<M\#>$.

If *Oracle* exists and semidecides *L*, then $C = Oracle(R(\langle M, w\rangle))$ semidecides ¬H.

- $\langle M, w\rangle \in$ ¬H: *M* does not halt on *w*. *M*# gets stuck in step 1.3. So *M*# accepts all strings that start with `a`. It doesn't accept anything else, including any strings that start with `b`. So *Oracle* halts and accepts.
- $\langle M, w\rangle \notin$ ¬H: *M* halts on *w*. *M*# accepts everything, including strings that start with `b`. So *Oracle* does not accept.

But *R* cannot exist, so neither does *Oracle*.

iii) {<*M*> : TM *M* accepts all strings that start with `a` and rejects all strings that start with `b`}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
   1.1. If the first character of *x* is `a`, accept. Otherwise:
   1.2. Save its input *x* on a second tape.
   1.3. Erase the tape.
   1.4. Write *w*.
   1.5. Run *M* on *w* for |*x*| steps.
   1.6. If *M* would have halted, then loop.
   1.7. Else reject.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then $C = Oracle(R(\langle M, w\rangle))$ semidecides ¬H. *M*# accepts all strings whose first character is `a`.

- $\langle M, w\rangle \in$ ¬H: *M* does not halt on *w*, so *M*# always gets to step 1.7. So it rejects everything except strings that start with `a` (those it already accepted in step 1.1.). So *M*# rejects all strings that start with `b`. So *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: *M* halts on *w*. Suppose it does so in *n* steps. Then, for all strings of length *n* or more, *M*# loops at step 1.6. But there are strings longer than that that start with `b`. *M*# won't reject them. So *Oracle* does not accept.

But *R* cannot exist, so neither does *Oracle*.

jjj) $\{\langle M_a, M_b\rangle : L(M_a) \subset L(M_b)\}$.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
   1.1. Save *x* for later.
   1.2. Erase the tape.
   1.3. Write *w*.
   1.4. Run *M* on *w*.
   1.5. Accept
2. Construct the description of *M*##(*x*) that operates as follows:
   2.1. If *x* = `b`, accept, else reject.
3. Return <*M*#, *M*##>.

If *Oracle* exists and semidecides *L*, then $C = Oracle(R(\langle M, w\rangle))$ semidecides ¬H. Note that $L(M\#\#) = \{b\}$.

- $\langle M, w\rangle \in$ ¬H: *M* does not halt on *w*, so $L(M\#) = \varnothing$. $\varnothing \subset \{b\}$. *Oracle* accepts.
- $\langle M, w\rangle \notin$ ¬H: *M* halts on *w*, so $L(M\#) = \Sigma^*$. $\Sigma^* \not\subset \{b\}$. *Oracle* does not accept.

But *R* cannot exist, so neither does *Oracle*.

kkk) $\{<M_a, M_b> : L(M_a) \subseteq L(M_b) \vee L(M_b) \subseteq L(M_a)\}$.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Save *x* for later.
    1.2. Erase the tape.
    1.3. Write *w*.
    1.4. Run *M* on *w*.
    1.5. If *x* = a, accept, else reject.
2. Construct the description of *M*##(*x*) that operates as follows:
    2.1. If *x* = b, accept, else reject.
3. Return <*M*#, *M*##>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H. Note that *L*(*M*##) = {b}.
- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *L*(*M*#) = ∅. ∅ ⊆ {b}. *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*, so *L*(*M*#) = {a}. Both {a} ⊆ {b} and {b} ⊆ {a} are false. *Oracle* does not accept.

But *R* cannot exist, so neither does *Oracle*.

lll) [Luay Nakhleh] $\{<M> : \forall s \in \Sigma^* (s \in L(M) \rightarrow s^R \notin L(M))\}$.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Erase the tape.
    1.2. Write *w*.
    1.3. Run *M* on *w*.
    1.4. Accept.
2. Return <*M*#>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H. Note that an alternative way to write the description of *L* is: $\{<M> : \forall s \in \Sigma^* (s \notin L(M) \vee s^R \notin L(M))\}$.
- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *L*(*M*#) = ∅. For any *s*, *s* ∉ *L*(*M*). *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*, so *L*(*M*#) = Σ*. So, for any *s*, neither *s* nor s^R fails to be in *L*. *Oracle* does not accept.

But *R* cannot exist, so neither does *Oracle*.

mmm) [Luay Nakhleh] {<*M*, *x*, *y*> : *M* accepts exactly one of the two strings *x* and *y*}.

¬SD: *R* is a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:
R(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. If *x* = ε, accept.
    1.2. Erase the tape.
    1.3. Write *w*.
    1.4. Run *M* on *w*.
    1.5. Accept.
2. Return <*M*#, ε, a>.

If *Oracle* exists and semidecides L, then C = *Oracle*(R(<M, w>)) semidecides ¬H.
- <M, w> ∈ ¬H: M does not halt on w, so M# accepts ε and nothing else, including `a`. *Oracle* accepts.
- <M, w> ∉ ¬H: M halts on w, so L(M#) = Σ*. So, M# accepts both ε and `a`. *Oracle* does not accept.

But R cannot exist, so neither does *Oracle*.

nnn) {<M, x, y> : x ∈ L(M) iff y ∈ L(M)}.

¬SD: R is a reduction from ¬H = {<M, w> : TM M does not halt on w} to L, defined as follows:

R(<M, w>) =
1. Construct the description of M#(x) that operates as follows:
   1.1. Save x.
   1.2. Erase the tape.
   1.3. Write w.
   1.4. Run M on w.
   1.5. If x = ε, else reject.
2. Return <M#, ε, `a`>.

If *Oracle* exists and semidecides L, then C = *Oracle*(R(<M, w>)) semidecides ¬H. Note that <M, x, y> is in L iff either both x and y are accepted by M or neither of them is.
- <M, w> ∈ ¬H: M does not halt on w, so M# accepts nothing. Both ε and `a` are outside L(M#). *Oracle* accepts.
- <M, w> ∉ ¬H: M halts on w, so M# accepts ε and nothing else. In particular, it does not also accept `a`. *Oracle* does not accept.

But R cannot exist, so neither does *Oracle*.

2) Prove each of the following claims:

a) If H ≤$_M$ A^nB^n then A^nB^n ≤$_M$ H.

Independently of whether H ≤$_M$ A^nB^n, A^nB^n ≤$_M$ H. We show that by defining:

R(x) =
1. If x ∈ A^nB^n, then let M* be the simple TM that immediately accepts. Return <M*, ε>.
2. If x ∉ A^nB^n, then let M* be the simple TM that immediately loops. Return <M*, ε>.

R is a mapping reduction from A^nB^n to H since x ∈ A^nB^n iff R(x) ∈ H.

b) If H ≤$_M$ A^nB^n then ¬H is in D.

If H ≤$_M$ A^nB^n then H is decidable if A^nB^n is. But A^nB^n is decidable. So H would also be. Since the decidable languages are closed under complement, ¬H would then also be decidable.

3) [Luay Nakhleh] Suppose that there are four languages *A*, *B*, *C*, and *D*. Each of the languages may or may not be in SD. However, we know the following about them:

- There is a mapping reduction from *A* to *B*.
- There is a mapping reduction from *B* to *C*.
- There is a mapping reduction from *D* to *C*.

For each of the following statements, indicate whether it is:

- CERTAIN to be true, regardless of what the languages *A* through *D* are,
- MAYBE true, depending on what *A* through *D* are, or
- NEVER true, regardless of what *A* through *D* are.

a) *A* is in SD/D and *C* is decidable.

NEVER. Mapping reducibility is transitive, so *A* is mapping reducible to *C*. If *C* is decidable and there's a mapping reduction from *A* to it, then *A* must also be decidable.

b) *A* is not decidable and *D* is not semidecidable.

MAYBE. If *C* is not in SD, then it's possible that none of the others are either, so the statement could be true. On the other hand, if *C* is decidable, then all the others must be too. In that case, the statement would be false.

c) If *C* is decidable, then the complement of *D* is also decidable.

CERTAIN. If *C* is decidable, then all the others also are (since all are reducible to it). Since the set of decidable languages is closed under complement, the complement of *D* must also be decidable.

d) The complement of *A* is not in SD but the complement of *B* is.

NEVER. For any two languages L_1 and L_2, if *R* is a mapping reduction from L_1 to L_2, then it is also a mapping reduction from $\neg L_1$ to $\neg L_2$. So we have that $\neg A$ is reducible to $\neg B$. Thus, if $\neg B$ is in SD, so is $\neg A$.

e) The complement of *B* is not decidable but the complement of *C* is decidable.

NEVER. For any two languages L_1 and L_2, if *R* is a mapping reduction from L_1 to L_2, then it is also a mapping reduction from $\neg L_1$ to $\neg L_2$. So we have that $\neg B$ is reducible to $\neg C$. Thus, if $\neg C$ is decidable, so is $\neg B$.

f) If *A* is decidable, then the complement of *B* is too.

MAYBE. Suppose that $A = B = \varnothing$. Then both *A* and $\neg B$ are decidable and the statement is true. But suppose that *A* is $\varnothing$ and *B* is H (the halting problem language). Then *A* is decidable but $\neg B$ isn't and the statement is false.

g) *A* is decidable but *D* is not.

MAYBE. Suppose that $A = \varnothing$ and $C = D = H$ (the halting problem language). Then *A* is reducible to *C* and is decidable. *D* is also reducible to *C*, but it is not decidable. This makes the statement true. But suppose that $A = C = D = \varnothing$. Then both *A* and *D* are decidable and the statement is false.

22 Undecidable Languages That Do Not Ask Questions about Turing Machines

1) Consider the following instance of the Post Correspondence problem. Does it have a solution? If so, show one.

i	X	Y
1	ab	a
2	ab	ba
3	aa	baa

Both 1, 3 and 1, 2, 3 are solutions.

2) Consider the following instance of the Post Correspondence problem. Does it have a solution? If so, show one.

i	X	Y
1	ab	aabb
2	ab	baba
3	bba	aba
4	bb	bbbb

Has no solutions.

3) Construct a nontrivial instance of PCP, different from the ones in the book, for which a solution exists and show the solution. By nontrivial we mean one in which the *X* and *Y* lists are not identical.

X = 101, 11, 001
Y = 10, 11110, 01
One solution: 1, 2, 1, 3

4) Is $L = \{<M> : M \text{ is a DFSM and } L(M) = \{a, aa\}\}$ decidable? Prove your answer.

Yes. It can be decided by doing the following on input <*M*>:
1. If a is not in Σ_M, halt and reject.
2. Run *M* on a. If it doesn't accept, halt and reject.
3. Run *M* on aa. If it doesn't accept, halt and reject.
4. Check that no other strings are accepted by doing the following: Let *k* be the number of states of *M*. Lexicographically enumerate all strings in Σ_M of length up to 2*k*. Run *M* on each such string. If it accepts any of them, halt and reject. If it accepts none of them, halt and accept. It is sufficient to try strings up to length 2*k*. By the pumping theorem, if *M* accepts any longer string, then it also accepts a shorter string that has had at most *k* characters pumped out. So it accepts some string *s* where $k < |s| < 2k$. If *M* accepts a and aa, but not ε, then *k* is at least 2. So *M* would have to accept some string of length at least 3. So it would have to accept some string other than a or aa with length < 2*k*. So we would find it.

5) [Luay Nakhleh] Is $L = \{<M> : M \text{ is a PDA and } L(M) = \{x : x \in \{a, b\}^* \text{ and } \exists m\ (|x| = 2^m)\}\}$ decidable? Prove your answer.

Yes. $L = \varnothing$, which is regular and thus decidable. The reason is that $L' = \{x : x \in \{a, b\}^* \text{ and } \exists m\ (|x| = 2^m)\}$ isn't context-free. So there is no PDA that accepts it. We prove that L' is not context-free by observing

that, if it were, then $L'' = L' \cap \mathtt{a}^*$ would also be context-free. But we show that it is not by pumping. Let w = a string of 2^k a's. We note that the next longest string in L'' is a string of 2^{k+1} a's.

$vy = \mathtt{a}^p$ for some nonzero p. Since $|vxy| \le k$, p must also be $\le k$. So, if we pump in once, the longest string that can result is one with $2^k + k$ a's, which isn't long enough, since, for all k, $2^k + k < 2^{k+1}$. So the string that results from pumping is not in L''. So L'', and thus L', is not context-free.

6) [Luay Nakhleh] Is $L = \{<G> : G$ is a context-free grammar and, for every string $w \in L(G)$, either w is of even length or w is of odd length and the last character of w is #$\}$ decidable? Prove your answer.

Yes. There are two ways to prove this.

Approach 1: A grammar G is in L unless it can generate an odd length string that doesn't end in #. Let b be the branching factor of G and let n be the number of nonterminals it uses. Then the following procedure decides L:

$P(G)$ =
1. For each odd-length string w in Σ_G^* of length up to $2b^{n+1}$ do:
   If the last character of w is not #, check whether $w \in L(G)$. If it is, exit the loop and reject.
2. Accept.

If there are any odd length strings in $L(G)$ that don't end in #, this procedure will find at least one of them. Suppose that there were one of length greater than $2b^{n+1}$. Call it s. The Pumping Theorem tells us that we can pump v and y out of s and produce another string that is also in $L(G)$. And we can keep doing that as long as the length of the pumped string is greater than k, which is b^{n+1}. Suppose that $|vy|$ is even. Then every time we pump out, we get another odd length string. So we will eventually get an odd length string of length less than k and thus certainly less than $2b^{n+1}$. Suppose, on the other hand, that $|vy|$ is odd. Then, if we pump out an odd number of times, we'll produce an even length string. Even length strings tell us nothing about whether G is in L. But, if we pump out an even number of times, we'll produce another odd length string. We know that $|vy| \le k$ $(= b^{n+1})$. So any string of length greater than $2b^{n+1}$, can be pumped out twice. So do double pumps on s. We will eventually get another odd length string of length less than $2b^{n+1}$. P will find that string.

Approach 2: Decide L using the following algorithm:

$P(G)$ =
1. Use either *cfgtoPDAtopdown* or *cfgtoPDAbottom* up to construct, from G, a PDA P such that $L(P) = L(G)$.
2. Build a DFSM F to accept the regular language $\{w \in \{\mathtt{a}, \mathtt{b}, \#\}^* : |w|$ is odd and the last character of w is not #$\}$.
3. Use *intersectPDAandFSM* to build, from P and F, a new PDA M that accepts $L(P) \cap L(F)$. Note that $L(M) = \{w : w \in L(G)$ and $|w|$ is odd and the last character of w is not #$\}$.
4. G is in L iff $L(M)$ is empty. So use *PDAtoCFG* to build, from M, a grammar G' such that $L(G') = L(M)$.
5. Run *decideCFLempty*(G'). If it returns *True*, return *True*; else return *False*.

7) Show that it is decidable, given a pushdown automaton M with one state, whether $L(M) = \Sigma^*$. (Hint: Show that such an automaton accepts all strings if and only if it accepts all strings of length one.)

M accepts all strings of length one iff:

1) Its single state, which we'll call q, is an accepting state, and
2) For each character c in Σ, $(q, c, \varepsilon) \vdash_M (q, \varepsilon, \varepsilon)$. This will be true iff one of the following is true of M:
   a) M contains the transition $((q, c, \varepsilon), (q, \varepsilon))$. In other words, M has a move it can make without popping anything from the stack (since the stack will be empty when M reads its first input symbol.) And that move pushes nothing, so M can immediately accept c.
   b) M contains the transition $((q, c, \varepsilon), (q, w))$, for some string w, and M contains a set of one or more ε-transitions that enable it to clear w from the stack.

Properties (1) and (2 a) can be determined by a straightforward examination of M. Property (2 b) is equivalent to asking whether M, with w on the stack, accepts ε. So one way to answer the question is to build a new PDA M^* (with more than one state) that operates as follows: From the start state, on ε, push the first character of w and go to state 2. From there, on ε, push the second character of w and go to state 3. And so forth. Once all of w is pushed, stay in the last state, pushing and popping just as M does. Now use *PDAtoCFG* to build a grammar G such that $L(G) = L(M^*)$. Finally, see whether S_G is nullable. If it is, G generates ε and M has property (2 b). Otherwise, M does not have property (2 b).

Next we show that M accepts all strings if and only if it accepts all strings of length one:

If direction: Suppose that M accepts all strings of length one. We must show that it accepts all strings of any length. We prove this by induction on k (the length of the input string).

- ***Base case:*** $k = 1$. True by assumption.
- ***Induction step:*** Show that, if the claim is true for $k = n$ then it must be true for $k = n+1$. Assume it is true for $k = n$. This means that q is an accepting state and, on any input of length $n+1$, there exists at least one path by which M will first read the initial n characters and land back in q with its stack empty. Since, starting in q with an empty stack, it can accept any single character, it can next read charater $n+1$, at which point it will also be able to accept.

Only if direction: If M does not accept all strings of length one then $L(M) \neq \Sigma^*$ since there are strings in Σ^* (e.g., some of length one) that M doesn't accept.

23 Unrestricted Grammars

1) Write an unrestricted grammar for each of the following languages L:

a) $\{\mathtt{a}^n\mathtt{b}^m\mathtt{c}^n\mathtt{d}^{n+m}, n, m \geq 0\}$.

$G = (\{S, T, W, A, B, C, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}, R, S\}$, where R =

$S \rightarrow \varepsilon$	/* In case n and $m = 0$.
$S \rightarrow T$	/* Otherwise.
$T \rightarrow \mathtt{a}CT\mathtt{d}$	/* Generate matching a, c, d.
$T \rightarrow W$	/* Move on to do matching b, d.
$W \rightarrow BW\mathtt{d}$	/* Generate matching b, d.
$W \rightarrow \varepsilon$	/* Finish the generation phase. Have a string like $\mathtt{a}C\mathtt{a}C\mathtt{a}CBB\mathtt{ddddd}$.
$C\mathtt{a} \rightarrow \mathtt{a}C$	/* Move C's to the right.
$CB \rightarrow BC$	/* Move C's to the right.
$C\mathtt{d} \rightarrow \mathtt{cd}$	/* Convert the rightmost C, next to the d region.
$C\mathtt{c} \rightarrow \mathtt{cc}$	/* Propagate conversion leftward in contiguous C's.
$B\mathtt{c} \rightarrow \mathtt{bc}$	/* Convert the rightmost B, next to the b region.
$B\mathtt{b} \rightarrow \mathtt{bb}$	/* Propagate conversion leftward in contiguous B's.

b) $\{\mathtt{a}^n\mathtt{b}^m : n > 0 \text{ and } m = 2^n+1\}$.

The idea is that we'll start by handling the case where $n = 1$. To do that, we'll generate the string $\mathtt{a}M\mathtt{bb}R$. We can then expand M as often as we like, each time introducing an a on one side and the special multiplier symbol X on the other side. Then, we'll push each X all the way to the right. As it goes over b's, it will double them. So the total number of b's will be doubled exactly the same number of times as an a was introduced.

$S \rightarrow \mathtt{a}M\mathtt{bb}R$	/* Generate the base case.
$M \rightarrow \mathtt{a}MX$	/* Do this once for each additional a.
$M \rightarrow \varepsilon$	
$X\mathtt{b} \rightarrow \mathtt{bb}X$	
$XR \rightarrow R$	/* This X has made it all the way through and has finished its job.
$R \rightarrow \mathtt{b}$	/* Generate the one extra b.

c) $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\#\mathtt{1}^n : n > 0\}$.

$S \rightarrow ABCS\mathtt{1} \mid ABC\#\mathtt{1}$	/* Generate a string of the form: $ABCABCABC$…# … 111
$CA \rightarrow AC$	/* Push the C's to the right and the A's to the left.
$CB \rightarrow BC$	"
$BA \rightarrow AB$	"
$C\# \rightarrow \mathtt{c}\#$	/* Convert once letters are in the right place.
$C\mathtt{c} \rightarrow \mathtt{cc}$	
$B\mathtt{c} \rightarrow \mathtt{bc}$	
$B\mathtt{b} \rightarrow \mathtt{bb}$	
$A\mathtt{b} \rightarrow \mathtt{ab}$	
$A\mathtt{a} \rightarrow \mathtt{aa}$	

d) [Don Baker] $\{\mathtt{a}^n\mathtt{b}^{2n}\mathtt{c}^{n+2} : n \geq 0\}$.

$S \rightarrow S_1\mathtt{cc}$	/* Generate the two extra c's once.
$S_1 \rightarrow \mathtt{a}BBS\mathtt{c}$	
$S_1 \rightarrow \varepsilon$	/* At this point, we have a string like a*BB*a*BB*a*BB*ccccc
$B\mathtt{a} \rightarrow \mathtt{a}B$	
$B\mathtt{c} \rightarrow \mathtt{bc}$	
$B\mathtt{b} \rightarrow \mathtt{bb}$	

e) [Don Baker] $\{w \in \{\mathtt{0}, \mathtt{1}, \#\}^*$, where w is of the form $b_i\#b_j$ and b_i and b_j are the binary encodings, without leading zeros of the nonnegative integers i and j respectively and $i < j\}$. For example, $\mathtt{1101\#1111} \in L$.

The idea is that we'll first generate a string of the form $b_i\ M\ b_i^R\ R + T$. Then we'll reverse b_i^R so we'll have $b_i\ M\ b_i\ R + T$. Now, to get rid of the + will require adding 1 to the second b_i. We can also insert additional copies of + so that we can add one an arbitrary number of additional times.

$S \rightarrow S_1\ R + T$	/* Generate a string of the form: $b_i\ M\ b_i^R\ R + T$.
$S_1 \rightarrow \mathtt{0}S_1\mathtt{0}$	″
$S_1 \rightarrow \mathtt{1}S_1\mathtt{1}$	″
$S_1 \rightarrow M$	″
$M \rightarrow MP$	/* Reverse the second copy of b_i. Use P as a pusher.
$P\mathtt{0}\mathtt{0} \rightarrow \mathtt{0}P\mathtt{0}$	″
$P\mathtt{0}\mathtt{1} \rightarrow \mathtt{1}P\mathtt{0}$	″
$P\mathtt{1}\mathtt{0} \rightarrow \mathtt{0}P\mathtt{1}$	″
$P\mathtt{1}\mathtt{1} \rightarrow \mathtt{1}P\mathtt{1}$	″
$P\mathtt{0}R \rightarrow R\mathtt{0}$	″ (hop the wall at R)
$P\mathtt{1}R \rightarrow R\mathtt{1}$	″ (hop the wall at R)
$MR \rightarrow \#$	/* All characters have hopped the wall. Get rid of it. Replace with #.
$T \rightarrow T\mathtt{+}$	/* Generate an arbitrary number of copies of +. Each will add 1.
$T \rightarrow \varepsilon$	″
$\mathtt{0+} \rightarrow \mathtt{1}$	/* Do the addition. Push + to the left until it reaches a 0 or the end.
$\mathtt{1+} \rightarrow \mathtt{+0}$	″
$\#\mathtt{+} \rightarrow \#\mathtt{1}$	″

f) $\{xyx : x, y \in \{\texttt{a}, \texttt{b}\}^* \text{ and } |x| = |y|\}$.

The idea is that we'll first generate a string of the form $X\%x^R\$$. X is x except that after each symbol there is a C. Next, we'll reverse x^R. Then the C's will get moved to the middle of the string and used to make y.

$S \rightarrow S_1\ \$$	
$S_1 \rightarrow \texttt{a}\ C\ S_1\ \texttt{a}$	/* Generate $X\%x^R\$$.
$S_1 \rightarrow \texttt{b}\ C\ S_1\ \texttt{b}$	″
$S_1 \rightarrow \%$	/* An example, at this point, is $\texttt{a}C\texttt{a}C\texttt{b}C\%\texttt{baa}\$$
$\% \rightarrow \%\ P$	/* Reverse x^R. Use P as a pusher.
$P\texttt{aa} \rightarrow \texttt{a}P\texttt{a}$	/* This group of rules pushes a character to the right wall.
$P\texttt{ab} \rightarrow \texttt{b}P\texttt{a}$	
$P\texttt{ba} \rightarrow \texttt{a}P\texttt{b}$	
$P\texttt{bb} \rightarrow \texttt{b}P\texttt{b}$	
$P\texttt{a}\$ \rightarrow \$\texttt{a}$	/* These next two rules jump the wall.
$P\texttt{b}\$ \rightarrow \$\texttt{b}$	
$\%\$ \rightarrow \#$	/* All characters have jumped. Make a new divider before second x.
$C\ \texttt{a} \rightarrow \texttt{a}\ C$	/* Push C's to the right, up against the #.
$C\ \texttt{b} \rightarrow \texttt{b}\ C$	″
$C\ \# \rightarrow \#\ \texttt{a}$	/* Each C that is in the right place gets converted to an a or b.
$C\ \# \rightarrow \#\ \texttt{b}$	″
$\# \rightarrow \varepsilon$	/* If this fires prematurely, any remaining C's will be stuck.

g) $\{w = yxx^Ry : x, y \in \{\texttt{a}, \texttt{b}\}^+\}$.

The idea is that we'll first generate $yx\#x^Ry^R\%$. At this point, the boundary between y and x (and thus between x^R and y^R) has not been fixed. Then we'll randomly choose a boundary between x^R and y^R. Then we'll reverse the characters after the boundary by pushing them to the right and jumping them over the %. We must be careful to choose a boundary that leaves at least one character in x^R and one in y^R.

$S \rightarrow S_1\ \%$	
$S_1 \rightarrow \texttt{a}\ S_1\ \texttt{a}$	/* Generate $yx\#x^Ry^R\%$.
$S_1 \rightarrow \texttt{b}\ S_1\ \texttt{b}$	″
$S_1 \rightarrow \#$	″
$\#\ \texttt{a} \rightarrow \texttt{a}\ \$$	/* Leave at least one character in x^R.
$\#\ \texttt{b} \rightarrow \texttt{b}\ \$$	″
$\$\ \texttt{a}\ \texttt{a} \rightarrow \texttt{a}\ \$\ \texttt{a}$	/* Push the boundary marker $ as far as we like but not past the
$\$\ \texttt{a}\ \texttt{b} \rightarrow \texttt{a}\ \$\ \texttt{b}$	last character.
$\$\ \texttt{b}\ \texttt{a} \rightarrow \texttt{b}\ \$\ \texttt{a}$	″
$\$\ \texttt{b}\ \texttt{b} \rightarrow \texttt{b}\ \$\ \texttt{b}$	″
$\$ \rightarrow @$	/* Commit to location of boundary marker.
$@ \rightarrow @\ P$	/* Spawn a pusher to be used to reverse the chars after @.
$P\texttt{aa} \rightarrow \texttt{a}P\texttt{a}$	/* This group of rules pushes a character to the right wall.
$P\texttt{ab} \rightarrow \texttt{b}P\texttt{a}$	
$P\texttt{ba} \rightarrow \texttt{a}P\texttt{b}$	
$P\texttt{bb} \rightarrow \texttt{b}P\texttt{b}$	
$P\texttt{a}\% \rightarrow \%\texttt{a}$	/* These next two rules jump the wall.
$P\texttt{b}\% \rightarrow \%\texttt{b}$	
$@\% \rightarrow \varepsilon$	

h) $\{(ab)^{2^n} : n \geq 0\}$.

$S \rightarrow$ # ab %
$\# \rightarrow \# D$ /* Each D is a doubler. Spawn (n-1) of them. Each of them will get pushed to the right and will turn each ab into abab as it passes over.
Dab $\rightarrow$ abaD /* Move right and double.
$D\% \rightarrow \%$ /* D's work is done. Wipe it out.
$\# \rightarrow \varepsilon$ /* Get rid of the walls.
$\% \rightarrow \varepsilon$ ″

2) [Luay Nakhleh] Consider the language $L = \{wtw : w, t \in \{a, b\}^* \text{ and } |w| = |t|\}$. If L is context-free, show a context-free grammar for it. Otherwise, show an unrestricted grammar for it.

L is not context-free, so we show an unrestricted grammar for it. The basic idea is that we'll start by generating a string of the form $X\%w^R\#$, where X is w except that there's a T after every character of X. So, for example, we might have aTaTbT%baa#. We'll reverse the final third of the string by hopping the characters, one at a time over the #. Then, we'll move all the T's to the right until the entire T region is immediately to the left of the %. Once a T has joined that region, it will become either an a or a b.

$G = (\{S, T, W, P, \#, \%, \$, a, b\}, \{a, b\}, R, S\}$, where R =
$S \rightarrow W\#$
$W \rightarrow$ a T W a | b T W b | % /* When done, string looks like aTaTbT%baa#.
$\% \rightarrow \%$ P /* Generate a pusher P.
Paa $\rightarrow$ aPa /* This group of rules pushes a character to the right wall.
Pab $\rightarrow$ bPa
Pba $\rightarrow$ aPb
Pbb $\rightarrow$ bPb
Pa# $\rightarrow$ #a /* These next two rules jump the wall.
Pb# $\rightarrow$ #b
%# $\rightarrow$ $ /* All characters have jumped. Make a new divider between w (with T's mixed in) and w^R. So we might have: aTaTb$T$$aab.
Ta $\rightarrow$ aT /* Move T's to the right, then turn them into a's or b's.
Tb $\rightarrow$ bT
$T$$ $\rightarrow$ $a | $b /* Everthing to the right of $ has been converted.
$T$$ $\rightarrow$ a | b /* When the last T is converted, get rid of $.

3) Construct an unrestricted grammar to compute each of the following functions:
a) $pred(n) = m$, where $value_1(n)$ is a natural number greater than 1 and $value_1(m) = value_1(n)$-1.

$G = (\{S, 1\}, \{1\}, R, S\}$, where R =
$1S \rightarrow \varepsilon$
$S1 \rightarrow 1$

b) $f(n) = m$, where $value_1(n)$ is a natural number and $value_1(m) = 3 \cdot value_1(n)+5$.

$G = (\{S, 1\}, \{1\}, R, S\}$, where R =
$S1 \rightarrow 111S$
$SS \rightarrow 11111$

c) *even?*(*n*) = *m*, where *value*$_1$(*n*) is a natural number and *m* = `1` if·*value*$_1$(*n*) is even and *m* = `11` if·*value*$_1$(*n*) is odd.

G = ({*S*, `1`}, {`1`}, *R*, *S*}, where *R* =
- *S*`11` → *S*
- *SS* → `1`
- *S*`1`*S* → `11`

d) *f*: {`a`, `b`}* → {`a`, `b`}*. *f*(*w*) = *w* with every even numbered character removed. (Start numbering the characters at 1.) For example, *f*(`abba`) = `ab`.

S `a` → `a` *T*	/* *S* means an even number of characters to the left. *T* means an odd number.
S `b` → `b` *T*	/* "
T `a` → *S*	/* When going from *T* (odd) to *S* (even), wipe out the current character.
T `b` → *S*	/* "
SS → ε	
TS → ε	

e) *f*: {`a`, `b`}$^+$ → {`a`, `b`}*.
f(*w*) = if *w* = `a`*y*, *y* ∈ {`a`, `b`}*, then *y*
if *w* = `b`*y*, *y* ∈ {`a`, `b`}*, then *yy*^{R}

For example:
- On input *S*`aaab`*S*, your grammar should produce `aab`.
- On input *S*`baab`*S*, your grammar should produce `aabbaa`.

G = ({*S*, *P*, *V*, *W*, `a`, `b`}, {`a`, `b`}, *R*, *S*}, where *R* =

S `a` → *V*	/* Get rid of initial *S* and first character. Decide which case we have.
S `b` → *W*	
V `a` → `a` *V*	/* Case 1. Push *V* to the right and get rid of *V S*.
V `b` → `b` *V*	
V S → ε	
W `a` → `a` *W P* `a`	/* Case 2. Make a copy of *y*, one character at a time. Generate a
W `b` → `b` *W P* `b`	/* pusher *P*. Push each character to the right and over the *S* to reverse.
P `a` `a` → `a` *P* `a`	/* These next 4 rules push a character all the way to the right to the *S*.
P `a` `b` → `b` *P* `a`	
P `b` `a` → `a` *P* `b`	
P `b` `b` → `b` *P* `b`	
P `a` *S* → *S* `a`	/* Jump the wall and get rid of the pusher.
P `b` *S* → *S* `b`	
W S → ε	/* All done.

4) Show that {<*G*> : *G* is an unrestricted grammar that does not generate ε} is not semidecidable.

We prove the claim by describing *R*, a reduction from ¬H = {<*M*, *w*> : TM *M* does not halt on *w*} to *L*, defined as follows:

R(<*M*, *w*>) =

1. Construct the description of *M*#(*x*) that operates as follows:
    1.1. Erase the tape.
    1.2. Write *w* on the tape.
    1.3. Run *M* on *w*.
    1.4 Accept.
2. From *M*#, construct the description <*G*#> of a grammar *G*# where *L*(*G*#) = *L*(*M*#).
3. Return <*G*#>.

If *Oracle* exists and semidecides *L*, then *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:

- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *M*# always gets stuck in step 1.3. *M*# accepts nothing. ε ∉ *L*(*M*#). So neither is it in *L*(*G*#). So *Oracle* accepts.
- <*M*, *w*> ∉ ¬H: *M* halts on *w*, so *M*# always accepts. ε ∈ *L*(*M*#). So it is also in *L*(*G*#). So *Oracle* does not accept.

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

24 Context-Sensitive Languages and the Chomsky Hierarchy

25 Computable and Partially Computable Functions

26 Summary of Part IV

1) Consider the language L defined by the following grammar $G = (\{S, A, \mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, R, S)$, where R =

$S \rightarrow \mathtt{cc}A S \mathtt{b}$
$S \rightarrow \mathtt{cc}A \mathtt{b}$
$A\mathtt{c} \rightarrow \mathtt{c}A$
$A\mathtt{b} \rightarrow \mathtt{ab}$
$A\mathtt{a} \rightarrow \mathtt{aa}$

a) Give a concise description of L.

$\{\mathtt{a}^{2n}\mathtt{b}^n\mathtt{c}^n : n \geq 1\}$

b) Consider the following language classes: regular, context-free, context-sensitive, D, and SD. What is the smallest class to which L belongs? Prove that L is in the class you name and that it is not in the next smallest class.

L is context-sensitive since G contains no length-reducing rules. L is not context-free, which we show using the Pumping Theorem. Let $w = \mathtt{a}^{2k}\mathtt{b}^k\mathtt{c}^k$. If either v or y crosses into more than one region, pump in once and the resulting string will not be in L because the letters will be out of order. Otherwise, v and y can cover at most two of the three regions. Pump in once. At least one region changes length but they do not all change. Thus the resulting string is not in L.

2) True or False:
a) For every language L, $\varnothing^* L = L$.

True. $\varnothing^* = \{\varepsilon\}$ and $\{\varepsilon\}L = L$.

b) For every language L, $\varnothing^* L = \varnothing$.

False. $\varnothing^* = \{\varepsilon\}$. Suppose $L = \{\mathtt{a}\}$. Then $\{\varepsilon\}\{\mathtt{a}\} = \{\mathtt{a}\} \neq \varnothing$.

c) The context-free languages are closed under union with the regular languages.

True. Every regular language is also context- free and the context-free languages are closed under union.

d) If M_1 is a nondeterministic PDA, then there exists a nondeterministic Turing machine M_2 such that $L(M_1) = L(M_2)$, but it is possible that there does not exist a deterministic Turing machine such that $L(M_1) = L(M_2)$.

False. Since every context-free language is in D, there exists a deterministic TM that accepts $L(M_1)$.

e) If M_1 is a deterministic PDA, then there exists a nondeterministic Turing machine M_2 such that $L(M_1) = L(M_2)$

True. Since there exists a PDA that accepts it, $L(M_1)$ is context-free and thus in D. Since it is in D, there exists a deterministic TM M_2 that decides it. M_2 is also an NDTM.

f) If M_1 is a deterministic Turing machine and if there is a PDA M_2 such that $L(M_2) = L(M_1)$ then M_2 must be deterministic.

False. Every context-free language can be decided by some deterministic TM, but not every context-free language can be decided by a deterministic PDA.

g) The union of a context-free language and a regular language must be in D.

True. Since every regular language is context-free and the context-free languages are closed under union, the union of a context-free language and a regular one must be context-free. Every context-free language is in D.

h) If L is a semidecidable language and $\approx_L$ has 2 equivalence classes, then L must be decidable.

True. If $\approx_L$ has 2 equivalence classes, then L is regular. Every regular language is decidable.

i) Rice's Theorem tells us that $\{<M> : |L(M)| > 5\}$ is not in D.

True. The property in question is a nontrivial property of the SD languages.

j) Rice's Theorem tells us that $\{<M_1, M_2> : L(M_1) \subseteq L(M_2)\}$ is not in D.

False. This language is indeed not in D. But Rice's Theorem doesn't apply since the property in question is of an ordered pair of SD languages, not a single language.

k) Rice's Theorem tells us that $\{<M, w_1, w_2> : M$ accepts w_1 and rejects $w_2\}$ is not in D.

False. This language is indeed not in D. But Rice's Theorem doesn't apply since the property in question is of an ordered (language, string, string) triple, not a single language .

l) Rice's Theorem tells us that $\{<M> : M$ accepts all even length strings$\}$ is not in SD.

False. This language is indeed not in SD. But Rice's Theorem only tells us that it is not in D.

m) Rice's Theorem tells us that $\{<M_a, M_b> : L(M_a) = L(M_b)\}$ is not in D.

False. This language is indeed not in SD. But Rice's Theorem doesn't apply since the property in question is of an ordered pair of SD languages, not a single language.

n) It is decidable, given two context-free grammars G_1 and G_2, whether $L(G_1) = L(G_2)$.

False. By Theorem 22.6.

o) It is decidable, given a context-free grammar G, whether $L(G) = \Sigma^*$.

False. By Theorem 22.5.

p) The semidecidable languages are closed under complement. .

False. By Theorem 20.5.

q) The decidable languages are closed under complement. .

True. By Theorem 20.4.

r) No semidecidable language is decidable. .

False. Every decidable language is also semidecidable.

s) Turing showed that there exists an algorithm to solve the Entscheidungsproblem. .

False. He showed that there does *not* exist such an algorithm.

t) The game of Life and the lambda calculus have equivalent computational power (i.e., any problem that can be solved by one of them can be solved by the other) .

True. Both formalisms can be used to describe exactly the semidecidable languages.

u) There exist functions that can be defined in the lambda calculus but that cannot be computed by any Turing machine. .

False. Turing showed the lambda calculus and Turing machines have equivalent computational power.

v) Unix-style regular expressions and Turing machines have equivalent computational power (i.e., any problem that can be solved by one of them can be solved by the other) .

True.

w) If L cannot be decided by a PDA with two stacks then L must not be in D.

True. We showed in Section 17.5.2 that a two-stack PDA has the same computational power as a Turing machine.

x) H is lexicographically Turing enumerable.

False. If it were, then it would be decidable, but it isn't.

y) If H were decidable then the semidecidable languages would be closed under complement.

True. If H were decidable, then every SD language would also be decidable and D is closed under complement.

z) The set of lexicographically Turing-enumerable languages is a subset of D. .

True. It is in fact exactly D.

aa) The set of lexicographically Turing-enumerable languages is a proper subset of D. .

False. It is the set SD. SD is not a subset of D since $H \in D$ but $H \notin SD$.

bb) D is a subset of the set of Turing-enumerable languages. .

True. The Turing-enumerable languages is the set SD. D is a subset of SD.

cc) The set of Turing-enumerable languages is a subset of D. .

False. The set of Turing-enumerable languages is SD and SD is not a subset of D.

dd) Let $L = \{<M> : L(M) \text{ is in SD}\}$. L is in D.

True. The definition of an SD language is that it is accepted by some Turing machine. So $L = \{<M> : L(M)\}$.

ee) If L_1 is in SD/D and L_2 is regular, then it is possible that $L_1 \cap L_2$ is regular.

True. Let L_1 be H and let L_2 be $\varnothing$. Then $L_1 \cap L_2 = \varnothing$, which is regular.

ff) If L_1 is in SD/D and L_2 is context-free, then it is not possible that $L_1 \cap L_2$ is regular.

False. Let L_1 be H and let L_2 be $\varnothing$. Then $L_1 \cap L_2 = \varnothing$, which is regular.

gg) The union of two context-free languages must be in D.

True. The context-free languages are closed under union. And every context-free language is in D.

hh) The intersection of two undecidable languages could be decidable.

True. Let L_1 be H and let L_2 be $\neg$H. $L_1 \cap L_2 = \varnothing$, which is decidable.

ii) The intersection of a context free language and a semidecidable one could be regular.

True. $\mathtt{a}^* \cap \mathtt{a}^*$ is regular. $\mathtt{a}^*$ is both -free and semidecidable.

jj) If $L_1 \cap L_2$ is semidecidable then both L_1 and L_2 must be semidecidable.

False. Let L_1 be $\neg$H and let L_2 be $\{\mathtt{a}\}$. Then $L_1 \cap L_2 = \varnothing$, which is semidecidable. But L_1 is not.

kk) If $L_1 \cap L_2$ is decidable then both L_1 and L_2 must be decidable.

False. Let L_1 be $\{\mathtt{a}\}$H (i.e., the string $\mathtt{a}$ concatened on the front of each string in H) and let L_2 be $\{\mathtt{b}\}$H. Neither L_1 nor L_2 is decidable. But $L_1 \cap L_2 = \varnothing$, which is decidable.

ll) If L is semidecidable and its complement is context-free, then L must be in D.

True. If L's complement is context-f ree, then it is also in SD. By Theorem 20.6, if L and its complement are both in SD then L is in D.

mm) It is possible that the union of an infinite number of decidable languages might also be decidable.

True. $\{\mathtt{a}\} \cup \{\mathtt{aa}\} \cup \{\mathtt{aaa}\} \cup \{\mathtt{aaaa}\} \cup \ldots$ is decidable.

nn) It is possible that, if $L_1 \cup L_2$ is regular, L_1 might not be regular.

True. Let $L_2 = \mathtt{a}^+$. Let $L_1 = \{\mathtt{a}^p : p \text{ is prime}\}$. $L_1 \cup L_2 = \mathtt{a}^+$, which is regular. But $L_1 = \{\mathtt{a}^p : p \text{ is prime}\}$ is not regular.

oo) If we subtract a finite number of strings from a decidable language, then it is possible that the result is regular.

True. Let $L = \mathtt{a}^+$. Subtract {a}, producing the set $\mathtt{aa}^+$, which is regular.

pp) If we add a finite number of strings to a semidecidable language, then it is possible that the result is decidable.

True. Let L be any decidable language. Then L is also semidecidable. The union of a decidable language and a finite one must be decidable.

qq) If L is semidecidable then its complement must not be decidable.

False. If L is semidecidable but not decidable, the claim is true. But L could be decidable. So, for example, let $L = \{\mathtt{a}\}$, which is regular, decidable, and semidecidable. Let $\Sigma = \{\mathtt{a}\}$. Then $\neg L = \varepsilon \cup \mathtt{aa}^+$, which is also regular and decidable.

rr) The complement of any semidecidable language must be decidable.

False. Let $L_1 = H$. $\neg H$ isn't even semidecidable, much less decidable.

ss) For all L_1 and L_2, if $L_1 \leq L_2$ and $L_2 \in D$ then $L_1 \in D$.

True. L_1 can be decided by composing the reduction with the decider for L_2.

tt) For all L_1 and L_2, if $L_1 \leq L_2$ and $L_2 \in SD$ then $L_1 \notin D$.

False. Let $L_1 = \{\mathtt{a}\}$ and let $L_2 = H$. $\{\mathtt{a}\} \leq H$. But $\{\mathtt{a}\}$ is in D.

uu) For all L_1 and L_2, if $L_1 \leq L_2$ and $L_2 \in SD/D$ then $L_1 \in SD/D$.

False. $\{\mathtt{a}\} \leq H$. But $\{\mathtt{a}\}$ is in D.

vv) If $L_1 \leq L_2$ and $L_2 \in SD$ then $L_1 \notin D$.

False. $A^nB^n \leq H$, $H \in SD$, and $A^nB^n \in D$.

ww) Given any two undecidable languages L_1 and L_2, it must be true that $L_1 \cup L_2$ is undecidable.

False. Let L_1 be H. Let L_2 be $\neg H$. Then $L_1 \cup L_2$ is $\{<M, w>\}$, which can be decided by the simple procedure that checks for valid syntax.

xx) Let $L = \{w: w$ is a syntactically well-formed sentence in first-order logic and w is a tautology$\}$. L is in D.

False. See Section 22.4.

yy) Let $L = \{w: w$ is a syntactically well-formed sentence in Boolean logic and w is a tautology$\}$. L is in D.

True. A straightforward decision procedure simply enumerates the truth table for w.

zz) Let PCP = $\{<P> : P$ is an instance of the Post Correspondence problem and P has a solution$\}$. PCP $\in$ SD/D. .

True. See Section 22.2.

aaa) Consider the Post Correspondence Problem instance with X = (aa, bb) and Y = (ab, bb). This problem has a solution. .

True. One such solution is (2).

bbb) Consider the Post Correspondence Problem instance with X = (aa, bb) and Y = (aba, ba). This problem has a solution. .

False. 1 could not be the first element of a solution. Neither could 2.

ccc) (1, 2, 2) is a solution to the Post Correspondence Problem instance with X = (ab, bb) and Y = (ba, abb).

False. This PCP instance has no solutions.

ddd) (3, 2, 1) is a solution to the Post Correspondence Problem instance with X = (bba, ba, ba) and Y = (a, abb, bab).

True. The string that results is bababba.

eee) Let TILES = {<T> : any finite surface on the plane can be tiled, according to the rules described in the book, with the tile set T}. TILES $\in$ SD.

False.

fff) For every nondeterministic Turing machine there exists an equivalent deterministic one.

True. By Theorem 17.2.

ggg) For every deterministic Turing machine there exists an equivalent nondeterministic one.

True. Every DTM is also an NDTM.

hhh) Let M be a Turing machine that never reads more than 17 squares of its tape on any input. M must halt on all inputs.

False. There exists a decision procedure to decide whether M will halt on any given input This follows from the fact that there is a finite number of configurations of M if we consider only 17 squares on either side of the square the read/write head starts in. But it it possible that M loops reading the same squares over and over.

iii) Let M be a Turing machine that never reads more than 2 squares of its tape on any input. $L(M)$ is regular.

True. The number of configurations of M, if we consider only 2 squares on either side of the square on which the read/write head starts is finite. So we can simulate M with an FSM.

jjj) Let M be a Turing machine that never reads more than 2 squares of its tape on any input. $L(M)$ is finite.

False. Let M be a TM with Σ = {a} and that immediately halts and accepts. M never reads more than 2 square of its tape on any input. But $L(M) = \Sigma^*$, which is inifinite.

kkk) The set of semidecidable languages over the alphabet {a, b} is not countable.

False. Each such language is semidecided by some Turing machine. The set of TMs with Σ = {a, b} can be lexicographically enumerated and so is countably infinite.

lll) If M is a 4-tape Turing machine that decides some language L, there exists a 2-tape Turing machine that also decides L.

True. Using the encoding technique described in the proof of Theorem 17.1, such a machine can be constructed.

mmm) If M is a 4-tape Turing machine that decides some language L, there exists a 1-tape Turing machine with a 2-character tape alphabet that also decides L.

True. Using the encoding technique described in the proof of Theorem 17.1, a one-tape machine can be constructed. Then a new machine that uses only two tape symbols can be constructed by encoding everything in binary.

nnn) If a nondeterministic Turing machine decides some language L in $\mathcal{O}(|x|)$ steps, where x is the input string, then we know that there exists a deterministic Turing machine that decides L in $\mathcal{O}(|x|^2)$ steps.

False. We know that a deterministic machine exists, but the best technique we have for constructing it may result in a machine that takes exponential time. We cannot prove, however, that it isn't possible to do better.

ooo) If a k-tape Turing machine decides some language L in $\mathcal{O}(|n|)$ steps, then there must exist a 1-tape Turing machine that decides L in $\mathcal{O}(n^2)$ steps.

True. By Theorem 17.1.

ppp) If L can be decided by a 2-tape deterministic Turing machine M in $\mathcal{O}(n^2)$ steps, then we are guaranteed that L can be decided by a 1-tape deterministic Turing machine that, on input w, will never require more than $4|w|^2$ steps.

False. It may take $\mathcal{O}(n^2)$ steps.

qqq) If L is lexicographically Turing enumerable then L must be in SD.

True. In fact, it must be in D.

rrr) $H \leq \{a^n b^n : n \geq 0\}$. (Recall that $L_1 \leq L_2$ means that L_1 is reducible to L_2.)

False. Since $\{a^n b^n : n \geq 0\}$ is in D, if H were reducible to it, H would also be in D. But it is not.

sss) $\{a^n b^n : n \geq 0\} \leq H$. (Recall that $L_1 \leq L_2$ means that L_1 is reducible to L_2.)

True. Use the following reduction $R(w)$:
1. If $w \in a^n b^n$ then construct the simple TM M that immediately halts. Return $<M, \varepsilon>$.
2. Else contruct the simple TM M that immediately loops. Return $<M, \varepsilon>$.

ttt) If $H = \{<M, w> : \text{TM } M \text{ halts on input } w\}$ were in D, $\neg H$ would still be undecidable.

False. The class D is closed under complement.

uuu) $(\{S, R\}, \{a, b\}, \{S \rightarrow aSR, S \rightarrow \varepsilon, aR \rightarrow Ra, R \rightarrow b\}, S)$ is a context-sensitive grammar.

False. The second rule is length-reducing.

vvv) ({*S*, *R*}, {a, b}, {*S* → a*SR*, *S* → ab, a*R* → *R*a, *R* → b}, *S*) is a context-sensitive grammar.

True. There are no length-reducing rules.

www) It is theoretically possible to implement a Java compiler as a Turing machine.

True. It can be done using the technique that is sketched in Section 17.4.

xxx) It is theoretically possible to implement a Java compiler as a PDA .

False. If this were true, then the set of legal Java programs, including the type constraints, would be a context-free language and we showed that it is not.

yyy) There exists an unrestricted grammar that generates H.

True. Theorem 23.1 tells us that, if *L* is in SD, then there exists an unrestricted grammar that generates it. H is in SD. So there exists an unrestricted grammar that generates it.

zzz) Mapping reducibility is an equivalence relation.

False. {a} is mapping reducible to H, but H is not mapping reducible to {a}. So the relation is not symmetric.

aaaa) There exist three languages L_1, L_2 and L_3 such that: $L_1 \subseteq L_2 \subseteq L_3$, $L_1 \notin$ D, $L_3 \notin$ D, and $L_2 \in$ D.

True. Let L_1 be EqTMs = $\{<M_a, M_b> : L(M_a) = L(M_b)\}$. Let L_2 be $\{M_a, M_b\}$. Let L_3 be $L_2 \cup$ H.

bbbb) It is possible for the intersection of two languages that are in SD/D to be regular.

True. PCP $\cap$ H = $\varnothing$, which is regular.

cccc) It is possible for the union of two languages that are not in SD to be regular.

True. Let $L_1 = H_{ALL} \cup$ {strings over the TM-encoding alphabet that are not legal TM descriptions}. Then the language:

$L_1 \cup \neg H_{ALL} = \Sigma^*$.

dddd) The complement of an SD language may be regular.

True. Let *L* be an arbitrary regular language. Then *L* is also SD. Since the regular languages are closed under complement, $\neg L$ must also be regular.

eeee) The intersection of a semidecidable language and a decidable language must be decidable.

False. Let Σ be any nonempty alphabet. Let $L_1 = \Sigma^*$. Let L_2 = H. $L_1 \cap L_2 = L_2$, which is not in D.

ffff) The intersection of a semidecidable language and a decidable language must be semidecidable.

True. Every decidable language is also semidecidable and the semidecidable languages are closed under intersection.

gggg) [Luay Nakhleh] We'll say that a real number x is computable iff there exists a Turing machine that enumerates (possibly infinitely) the digits of x after the decimal point. All real numbers are computable.

False. There are only countably infinitely many TMs, but there is an uncountably infinite number of real numbers. So there are not enough TMs to be able to do this.

hhhh) [Luay Nakhleh] $A_{ALL} \leq_M A_{ANY}$.

False. A_{ANY} is semidecidable (although it is not decidable). A_{ALL} is not in SD. But if it were mapping reducible to A_{ANY}, it would be in SD, since the composition of the reduction with the TM that semidecides A_{ANY} would semidecide A_{ALL}.

iiii) Every language that is accepted by some deterministic LBA is decided by some deterministic Turing machine.

True. If L is accepted by some deterministic LBA, then it is context-sensitive. By Theorem 24.2, every context-sensitive language is decidable.

3) For each of the following statements, circle all of the language classes for which it is true. Remember that if L is regular, then it is also context free, etc.

a) If L_1 and L_2 are languages in this class, then $L_1 \cap L_2$ must be a language in this class.

Regular Context Free D SD SD/D ¬SD

Note that $L_1 \cap L_2$ could be empty.

b) If L_1 and L_2 are languages in this class, then $L_1 \cup L_2$ must be a language in this class.

Regular Context Free D SD SD/D ¬SD

c) If L is in this class then there exists an infinite number of grammars for L.

Regular Context Free D SD SD/D ¬SD

d) If L is in this class, then there exists a Turing machine M that lexicographically enumerates the elements of L.

Regular Context Free D SD SD/D ¬SD

e) If L is in this class, with alphabet Σ, then there exists a decision procedure that determines whether an arbitrary string $w \in \Sigma^*$ is an element of L.

Regular Context Free D SD SD/D ¬SD

f) If L is in this class, then there must exist a deterministic TM that decides L.

Regular Context Free D SD SD/D ¬SD

g) Let L be the language defined by the grammar: $S \rightarrow S\mathtt{a}S$, $S \rightarrow \varepsilon$. L must be in this class.

Regular Context Free D SD SD/D ¬SD

Note that $L = \mathtt{a}^*$.

h) If L is in this class then $L \cup \{\varepsilon\}$ must be in this class.

Regular	Context Free	D	SD	SD/D	¬SD

i) If L is in this class then L^R must be in this class.

Regular	Context Free	D	SD	SD/D	¬SD

Part V: Complexity

27 Introduction to the Analysis of Complexity

1) [David Bunde] Let M be an arbitrary Turing machine.
   a) Suppose that $timereq(M) = 2n^2(n\text{-}1)$. Circle all of the following statements that are true:
      i) $timereq(M) \in \mathcal{O}(n^2)$. False
      ii) $timereq(M) \in \mathcal{O}(n^3\text{-}3n)$. True
      iii) $timereq(M) \in \mathcal{O}(n^4)$. True
      iv) $timereq(M) \in \Theta(n^5)$. False

   Circle ii and iii. The limit of $2n^2(n\text{-}1)$ over n^2 is infinity so the first does not hold. The second holds with $c = 3$ and $k = 3$. The third holds with $c = 2$ and $k = 1$. The fourth does not hold because the limit of $2n^2(n\text{-}1)$ over n^5 is zero.

   b) Suppose $timereq(M) = 2^n \cdot n^2$. Circle all of the following statements that are true:
      i) $timereq(M) \in \mathcal{O}(n^5)$. False
      ii) $timereq(M) \in \mathcal{O}(2^n)$. False
      iii) $timereq(M) \in \mathcal{O}(n!)$. True

   Circle iii. The limit of $2^n \cdot n^2$ over either of the first two is infinity. The third holds with c = 1 and k = 8. ($8! > 2^8 \cdot 8^2$ and increasing n by 1 for $n \geq 8$ increases the right-hand side by a factor of at most $2 \cdot (9/8)^2 < 8$ while the left-hand side increases by a factor of more than 8.)

2) [David Bunde] Let M be the Turing machine shown in Example 17.11. M duplicates a string. Analyze $timereq(M)$.

   Let n be the length of the input string w. M is described as two smaller machines, a copy machine and a shift machine. The copy machine repeatedly moves n+2 places to the right and then moves n+1 places to the left. It does this n times before halting on the space between the two copies. The shifting machine makes 3 moves to shift each of the n characters, for $3n$ total moves. The last thing M does is move back to the beginning of the input, which requires $2n$ moves. Thus, the total number of moves is $n(2n+3)+3n+2n = 2n^2+8n \in \mathcal{O}(n^2)$.

3) [David Bunde] Assume a computer that executes 10^{10} operations/second. Make the simplifying assumption that each operation of a program requires exactly one machine instruction. For each of the following programs P, defined by its time requirement, what is the largest size input on which P would be guaranteed to halt within an hour?
   a) $timereq(P) = 80n$.
   b) $timereq(P) = 3n^2$.
   c) $timereq(P) = 4^{n-1}$.

   There are $60 \times 60 = 3600$ seconds in an hour so we are looking for the largest value of n that is at most 3600×10^{10}.
   a) Dividing the number of seconds by 80 gives 45×10^{10} as the largest possible value of n.
   b) Dividing the number of seconds by 3 gives 1200×10^{10} as the largest possible value of n^2. The largest possible value of n is 3464101 since $3464101^2 = 11999995738201$ and $3464102^2 = 12000002666404$. (You can get in the right neighborhood by taking the square root of 1200 and then multiplying by 10^5.)
   c) Calculating some powers of n shows that $4^{21} = 4398046511104$ while $4^{22} = 17592186044416$. Thus, the largest possible value of n is 22 (recall that $timereq(P) = 4^{n-1}$, not 4^n). (A somewhat more mathematically-satisfying technique than trying values is to use a logarithm base 4.)

4) [David Bunde] Let each line of the following table correspond to a problem for which two algorithms, *A* and *B*, exist. The table entries correspond to *timereq* for each of those algorithms. Determine, for each problem, the smallest value of *n* (the length of the input) such that algorithm *B* runs faster than algorithm *A*.

A	***B***
n^2	$1000000n$
n^3	$1000000n$
2^n	$1000000n$
$n!$	$1000000n$

- Row 1: At $n = 1000000$, the two algorithms have equal running time and at $n = 1000001$, algorithm *B* runs faster.
- Row 2: At $n = 100$, the two algorithms have equal running time and at $n = 101$, algorithm *B* runs faster.
- Row 3: Algorithm *B* first runs faster at $n = 25$ because $2^{24} = 16777216 < 24000000$ and $2^{25} = 33554432 > 25000000$.
- Row 4: Algorithm *B* first runs faster at $n = 11$ because $10! = 3628800 < 10000000$ and $11! = 39916800 > 11000000$.

5) Show that $L = \{<M>: M \text{ is a Turing machine and } timereq(M) \in \mathcal{O}(n)\}$ is not in D.

The basic idea is that *timereq*(*M*) is only defined if *M* halts on all inputs. We prove that *L* is not in D by reduction from H. Define:

R(<*M*, *w*>) =
1. Construct the description <*M#*> of a new Turing machine *M#*(*x*) that, on input *x*, operates as follows:
   1.1. Run *M* on *w* for |*x*| steps or until it halts naturally.
   1.2. If *M* would have halted naturally, then loop.
   1.3. Else halt.
2. Return <*M#*>.

{*R*, ¬} is a reduction from H to *L*. If *Oracle* exists and decides *L*, then $C = \neg Oracle(R(<M, w>))$ semidecides H:

- If $<M, w> \in$ H: *M* halts on *w*. So, on some (long) inputs, *M#* will notice the halting. On those inputs, it fails to halt. So *timereq*(*M#*) is undefined and thus not in $\mathcal{O}(n)$. *Oracle*(<*M#*>) rejects and *C* accepts.
- If $<M, w> \notin$ H: *M* does not halt on *w*, so *M#* will never discover that *M* would have halted. So, on all inputs, *M#* halts in $\mathcal{O}(n)$ steps (since the number of simulation steps it runs is *n*). So $timereq(M\#) \in \mathcal{O}(n)$ and *Oracle*(<*M#*>) accepts. So *C* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

6) True or False:

a) $n \in \mathcal{O}(n^4)$ — True
b) $2^n = \mathcal{O}(2^n)$ — False
c) $2n + 2^n \in \mathcal{O}(2n)$ — False
d) $n^3 + 2n + 3 \in \Theta(n^4)$ — False
e) $n^3 + 2n + 3 \in \mathcal{O}(n^3)$ — True
f) $n^3 \in \mathcal{O}(3n^3)$ — True
g) $n^2 = \mathcal{O}(n^3)$ — False
h) $2n^3 + 2n + 3 \in \Theta(n^5)$ — False
i) $2^n + 2n \in \mathcal{O}(n!/2)$ — True

j) $7n^4 + 2n + 3 \in \mathcal{O}(n^7)$ True

28 Time Complexity Classes

1) Prove that P is closed under:
   a) difference (where the difference between two languages L_1 and L_2 is $L_1 - L_2$).

   If L_1 and L_2 are in P, then there exist deterministic, polynomial-time Turing machines M_1 and M_2 that decide them. We show a new, deterministic, polynomial-time Turing machine M that decides $L_1 - L_2$. On input w, run M_1 on w. If it rejects, reject. Otherwise, run M_2 on w. If it accepts, reject. Otherwise accept.

   b) reverse.

   If L_1 is in P, then there exists a deterministic, polynomial-time Turing machine M_1 that decides it. We show a new, deterministic, polynomial-time Turing machine M that decides L^R. On input w, first reverse w. (This can be done in polynomial time.) Then run M_1 on w^R. Accept if it accepts; reject if it rejects.

2) Prove that NP is closed under:
   a) reverse.

   If L_1 is in NP, then there exists a nondeterministic, polynomial-time Turing machine M_1 that decides it. We show a new, nondeterministic, polynomial-time Turing machine M that decides L^R. On input w, first reverse w. (This can be done deterministically in polynomial time.) Then run M_1 on w^R. Accept if it accepts; reject if it rejects.

3) Show that each of the following languages is NP-complete by first showing that it is in NP and then showing that it is NP-hard:
   a) CHROMATIC-NUMBER = {<G, k> : G is an undirected graph whose chromatic number is no more than k}. Recall that the chromatic number of a graph G is the smallest number of colors required to color its vertices, subject to the constraint that no two adjacent vertices may be assigned the same color.

   We first show that CHROMATIC-NUMBER is in NP by exhibiting *Ver*, a polynomial-time verifier for it. Let c be a certificate: a list of vertices and, for each, a color. On input <G, k, c>, *Ver* operates as follows. Let V be the vertices of G and E be its edges. Then:

   1. Check that c contains exactly one entry for every vertex in V. If it does not, halt and reject.
   2. Count the number of colors that are used in c. If it is greater than k, halt and reject.
   3. For each vertex v in V do:
      3.1. For each vertex v' that is connected to v by an edge in E do:
         Check that v and v' are colored differently in c. If not, halt and reject.
   4. Halt and accept.

   Ver runs in polynomial time. Step 1 can be easily implemented to run in $\mathcal{O}(|V|^2)$ time. Step 2 runs in $\mathcal{O}(|V|)$ time. Step 3 runs in $\mathcal{O}(|V| \cdot |E|)$ time.

4) [David Bunde] Let ONE-CLIQUE ={<*G*, *k*>: *G* is an undirected graph with vertices *V*, *k* is an integer satisfying $1 \leq k \leq |V|$, and *G* contains exactly one *k*-clique}. Does the following nondeterministic, polynomial-time algorithm decide ONE-CLIQUE? Explain your answer.

*decide*ONE-CLIQUE(<*G*,*k*>) =
1. Nondeterministically select a set *S* of *k* vertices of *G*.
2. If *S* does not form a clique, reject.
3. Else nondeterministically select another set $T \neq S$ of *k* vertices.
4. If *T* forms a clique, reject.
5. Else accept.

No. The problem is that a nondeterministic program will accept if even one path accepts. Thus, this algorithm will accept if there is a *k*-clique and another group of *k* vertices that do not form a clique. The algorithm designer probably intended step 3 to select another *k*-clique if one exists, but nothing forces that to happen.

5) [David Bunde] Show that, if P = NP, then there exists a deterministic, polynomial-time algorithm that finds a Hamiltonian path in a graph if one exists.

Since HAMILTONIAN-PATH is NP-complete, it is in NP. If P = NP, then HAMILTONIAN-PATH is also in P and therefore has a deterministic, polynomial-time decision algorithm *A*. We will use *A* as a subroutine in our procedure for actually finding a Hamiltonian path.

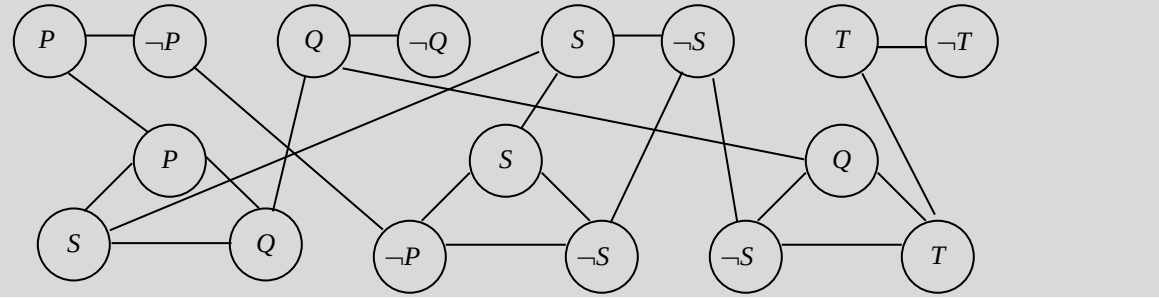
We begin by calling *A* on the input graph *G*. If *A* rejects <*G*>, then our algorithm says there is no Hamiltonian path and exits. If *A* accepts <*G*>, then we need to find which edges it uses. To do this, we make a series of calls to *A*, giving it slight modifications of the graph *G*. Specifically, for each edge *e* of *G*, we remove *e* and call *A* on the modified graph. If *A* accepts the modified graph, then we know that edge *e* is not necessary for a Hamiltonian path so we leave it out of the graph used in subsequent calls. If *A* rejects the modified graph, edge *e* must appear in every remaining Hamiltonian path so we return it to the graph.

The modified graph resulting from this procedure contains at least one Hamiltonian path because *G* started with a Hamiltonian path and *A* accepts the modified graph at the end of each time through the loop (after edge *e* is added back if necessary). The modified graph also cannot contain any edge whose removal leaves a Hamiltonian path in the remaining graph or that edge would have been removed when selected by the loop. Therefore, the modified graph contains exactly the edges of a Hamiltonian path.

How long does this procedure take? It makes one call to *A* for each edge of *G* plus one extra. Since each edge is listed in the input, the number of calls is proportional to the length of the input. Each of these calls is on a graph no larger than *G* so each takes time polynomial in the input length. Therefore, the total time devoted to calls to *A* is polynomial in the input length. The rest of the procedure also runs in polynomial-time since it only makes simple changes to the graph.

6) [David Bunde] Let R be the reduction from 3-SAT to VERTEX-COVER that we defined in the proof of Theorem 28.20. Show the graph that R builds when given the Boolean formula:

$$(P \vee S \vee Q) \wedge (S \vee \neg P \vee \neg S) \wedge (Q \vee \neg S \vee T)$$



7) [David Bunde] Recall that a wff in Boolean logic is said to be valid iff it is true for any assignment of truth values to its variables. We will say that a wff is non-valid iff some assignment of truth values makes it false. Show that the following language is NP-complete:

- NON-VALID = {<w>: w is a wff in Boolean logic and w is not valid}

We must prove that NON-VALID is in NP and that it is NP-hard.

We first prove that NON-VALID is in NP: We describe *Ver*, a deterministic, polynomial-time verifier for it. For w, a wff in Boolean logic, our certificate c will be an assignment of truth values to the variables. *Ver* can take this assignment and, using the procedure described in the proof of Theorem 28.12, verify that w evaluates to *False* given the assignments provided by c.

Next we show that NON-VALID is NP-hard: We give a deterministic polynomial-time reduction R from SAT. To perform this reduction, R uses the input <w> to create <$\neg(w)$>. This clearly runs in polynomial time. To show that it is correct, we must show that <w> $\in$ NON-VALID iff <$\neg(w)$> $\in$ SAT.

We first show that <w> $\in$ NON-VALID $\rightarrow$ <$\neg(w)$> $\in$ SAT. If <w> $\in$ NON-VALID, then w is non-valid and there is some assignment of truth values that makes w evaluate to *False*. This same assignment of truth values makes $\neg(w)$ evaluate to *True* so <$\neg(w)$> $\in$ SAT.

Next, we show that <$\neg(w)$> $\in$ SAT $\rightarrow$ <w> $\in$ NON-VALID. If <$\neg(w)$> $\in$ SAT, then $\neg(w)$ is satisfiable and some assignment of truth values makes $\neg(w)$ evaluate to *True*. This same assignment of truth values makes w evaluate to *False* so <w> $\in$ NON-VALID.

8) [David Bunde] Show that INTEGER-PROGRAMMING is NP-hard by reduction from VERTEX-COVER.

We give a deterministic polynomial-time reduction R that takes <G, k>, the encoding of graph $G = (V, E)$ and desired size of the vertex cover k. Reduction R creates a set of linear inequalities. For each vertex v in G, the set of linear inequalities has a variable x_v. It has the following groups of inequalities:

- $-x_v \leq 0$ and $x_v \leq 1$ for each variable x_v
- $-x_u - x_v \leq -1$ for each edge $(u, v) \in E$
- $\Sigma_{v \in V} x_v \leq k$

There are two inequalities for each vertex in V, one for each edge in E, plus one other. All the inequalities have length at most proportional to the number of vertices. Thus, the total length of these inequalities is polynomial in the input size. Since each inequality is straightforward to compute from G, they can also be computed in polynomial-time.

It remains to show that $<G, k> \in$ VERTEX-COVER iff $R(<G, k>) \in$ INTEGER-PROGRAMMING. We first show that $<G, k> \in$ VERTEX-COVER $\rightarrow R(<G, k>) \in$ INTEGER-PROGRAMMING. Let S be the set of vertices in the vertex cover. We create an integer vector by setting the variables corresponding to members of S to one and all other variables to zero. The first group of inequalities are satisfied because all variables have value either 0 or 1. Each inequality in the second group is satisfied because each edge is incident to at least one vertex in the vertex cover (by definition of a vertex cover) and the variable corresponding to that vertex therefore contributes -1 to the left-hand side of that inequality. The inequality from the third group is satisfied because at most k vertices are included in the vertex cover and only the variables corresponding to those vertices contribute to the sum.

We complete the proof by showing that $R(<G, k>) \in$ INTEGER-PROGRAMMING $\rightarrow <G, k> \in$ VERTEX-COVER by taking an assignment of values to the variables and creating a vertex cover. In particular, the vertex cover includes all vertices corresponding to variables given value one. The first two groups of inequalities imply that every edge is incident to at least one vertex in the vertex cover. The first and third groups of inequalities imply that at most k vertices are included.

9) [David Bunde] The night manager of a university print center is responsible for seeing that all handouts are printed for the next day's classes. At the beginning of the evening, he receives copies of all the handouts to be printed along with the number of copies of each that are required. (The time to make copies of a handout is proportional to the number of copies required.) He tries to divide the handouts between two copy machines and makes the copies; print center policy forbids using both copy machines on the same handout. His goal is to minimize the completion time of the last machine to finish since he goes home once all the handouts have been made.
   a) Convert this optimization problem to a language recognition problem.

PRINT-QUICK = {$<S, k>$: S is a multiset of integers, k is an integer, and there is a way to divide S into two subsets A and S-A such that the sum of the elements in each subset is at most k}

   b) Make the strongest statement you can about the complexity of the resulting language.

PRINT-QUICK is NP-complete. The members of A form the certificate used by a deterministic verifier. To see that PRINT-QUICK is NP-hard, we give a reduction from SET-PARTITION to it. (Observe that SET-PARITION is an almost identical problem; it lacks the parameter k and always tries to divide the set into equal parts.) Our reduction takes an instance $<S>$ of SET-PARTITION, adds up all the values in S, and creates an instance $<S, k>$ of PRINT-QUICK with k equal to half the sum of values in S. (If the sum is odd, we round down; S cannot be partitioned and the resulting word will not be in PRINT-QUICK.) Since we are just rephrasing the problem, $<S> \in$ SET-PARTITION iff $<S, k> \in$ PRINT-QUICK.

Notice that PRINT-QUICK is the two-processor version of MULTIPROCESSOR-SCHEDULING, described in [Garey and Johnson 1979].

10) [David Bunde] Define the following language:

- SUBSET-UNSUM = {$<S, k>$: S is a multiset (i.e., duplicates are allowed) of integers, k is an integer, and there does *not* exist a subset of S whose elements sum to k}.

Explain why it is generally believed that SUBSET-UNSUM is not NP-complete.

SUBSET-UNSUM is the complement of the NP-complete language SUBSET-SUM. If SUBSET-UNSUM is NP-complete, then Theorem 28.26 tells us that NP = Co-NP. Although this is not known to be incorrect, these two complexity classes are widely believed to be different; despite substantial effort, no one has been able to find an NP-complete language whose complement is in NP.

11) True or False:

a) For every nondeterministic Turing machine, there is known to exist an equivalent deterministic one that runs in no more than twice the time.

False. There exists an equivalent deterministic one but the best guarantee that we know that we can make is that it take exponentially more time.

b) For every polynomial-time nondeterministic Turing machine an equivalent polynomial-time deterministic one is known to exist.

False. There exists an equivalent deterministic one but the best guarantee that we know that we can make is that it take exponentially more time.

c) There exists a polynomial-time nondeterministic Turing machine for which it is known that no equivalent deterministic polynomial-time Turing machine exists.

False. While we cannot prove that an equivalent polynomial-time deterministic Turing machine exists, neither can we prove that one might not.

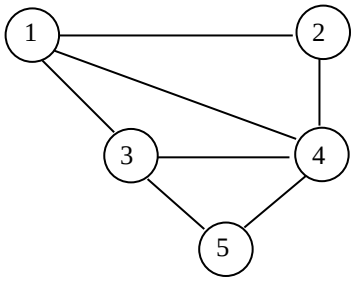
d) Let TILES = {<*T*> : any finite surface on the plane can be tiled, according to the rules described in the book, with the tile set *T*}. There exists an exponential-time algorithm to decide TILES.

False. TILES isn't decidable by any algorithm.

e) For every context-free language, there exists a deterministic, polynomial-time verifier.

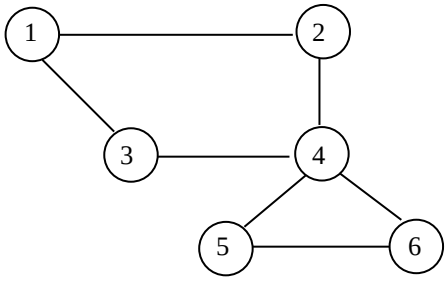
True. By Theorem 28.3, every context-free language is in P. So it is also in NP.

f) There exists an Eulerian circuit through the following graph:



False. Vertices 1 and 3 have odd degree.

g) There exists an Eulerian circuit through the following graph:



True. All vertices have even degree.

h) Consider the string S: $(P \wedge Q) \vee \neg T$. S is in SAT.

True. A satisfying assignment is P = *True*, Q = *True*, T = *False*.

i) Consider the string S: $(P \wedge Q) \vee \neg T$. S is in 3-SAT.

False. It does not have the correct syntax.

j) To show that a language L is NP-complete, it suffices to show that 3-SAT $\leq_P L$.

False. L might not be in NP.

k) VERTEX-COVER $\leq_P$ 3-SAT.

True. This must be so since VERTEX-COVER is in NP and 3-SAT is NP-complete.

29 Space Complexity Classes

1) [David Bunde] Prove that $A^nB^nC^n = \{a^nb^nc^n: n \geq 0\}$ is in L.

We give a deterministic Turing machine *M* to decide this language using logarithmic space. *M* uses its work tape to store three counts in binary, separated by # signs. Each count stores the number of times one type of character occurs. *M* reads the input, incrementing the first counter as long as the character `a` is read. Once the character `b` is read, then *M* begins incrementing the second count. Similarly, once a `c` is read, *M* begins incrementing the third count. If a character corresponding to an earlier count is ever read, *M* rejects immediately because the input word does not have the correct form. Once the input word ends, *M* begins to check whether the counts have the same value. To do this, it first examines the leftmost digit in each, replacing these digits with # signs. Then it compares the second digits by looking at the first digit after each group of # signs, followed by the third digits (found the same way) and so on. If there are ever fewer than three digits occurring after groups of # signs, then one of the counts was too small and *M* rejects immediately. Similarly, if any of the corresponding digits ever differ, *M* rejects immediately because the counts differ. *M* only accepts if the non-blank part of its work tape becomes entirely # signs.

Because the counts are stored in binary, each has only logarithmic length with respect to the input. Since *M* uses three counts (a constant number) and only a constant other amount of space for the separating # signs, the total space used is still logarithmic and $A^nB^nC^n \in L$.

30 Practical Solutions for Hard Problems

1) [David Bunde] Give a randomized polynomial-time algorithm that takes a wff of Boolean logic in 3-conjunctive normal form and outputs an assignment of truth values to variables that satisfies at least half the clauses in expectation. (*Hint:* Analyze the algorithm that makes a random assignment of truth values to variables.)

As suggested in the hint, we consider the algorithm that makes a random assignment of truth values. Each literal separately is satisfied by half the truth assignments (those assigning the appropriate truth value to the variable occurring in the literal). Since a clause is satisfied if at least one of its literals is satisfied, each clause is satisfied by at least half the truth assignments. (Exactly one half is achieved by a clause where the same literal appears three times, though clauses containing distinct literals are satisfied by a higher proportion of truth assignments.) Thus, any single clause in the formula has at least a 50% probability of being satisfied by the random assignment. By linearity of expectation, this implies that the expected number of satisfied clauses is at least half the total number.

2) [David Bunde] A popular word puzzle requires us to try to find a path between a pair of words, where each step changes only one letter at a time and results in a valid word. For example, consider the following path between the words GAME and HARD:

GAME
TAME
TOME
TORE
BORE
BARE
BARD
HARD

Suppose you want to write a program using A* to find paths between pairs of words. Your program takes an initial word and a goal word, plus a dictionary listing all the valid words. Each node of the search is a word reached from the initial word. As you expand the search tree from the initial word, the function $g = f + h$ gives the distance of each node from the initial word. Which of the following functions h are admissible for this search?

a) $h_1(n) = 0$.
b) $h_2(n)$ = the number of words in the dictionary occurring alphabetically between n and the goal word.
c) $h_2(n)$ = the number of words in the dictionary occurring alphabetically between n and the goal word.
d) $h_4(n)$ = the number of letters in which n and the initial word differ.

A function $h(n)$ is admissible iff it never overestimates the number of words that must be visited on a path from n and the goal word. The functions h_1 and h_3 satisfy this definition. Function h_1 is admissible because the actual distance between any pair of words is at least zero. Function h_3 is admissible because each step changes only one letter so the number of different letters is a lower bound on the number of steps between a pair of words. Function h_2 can overestimate the distance because changing a single letter can move a considerable distance through the dictionary. (For example, the move BARD $\rightarrow$ HARD above skips over GAME, BORE, and BARE.) Function h_4 can overestimate because it has no particular relationship with the distance to the goal word. For example, h_4(BARD) = 3 in the sample above even though this word is one step from the goal word.

31 Summary and References

1) True or False:
   a) $L = \{<M>: M \text{ is a Turing machine and } timereq(M) \in \mathcal{O}(n)\}$ is in D.

      False. *timereq*(*M*) is defined only on TMs that always halt and that question is undecidable.

   b) $L = \{<M>: M \text{ is a Turing machine and } timereq(M) \in \mathcal{O}(2^n)\}$ is in SD.

      False. *timereq*(*M*) is defined only on TMs that always halt and that question is not semidecidable.

Appendix A: Review of Mathematical Background

1) Let *GW* (*GoesWith*) be the following relation:

{(chocolate, sugar), (sugar, cinnamon), (cinnamon, nutmeg), (onions, cheese), (cheese, peppers)}

a) Let *GWRT* be the reflexive transitive closure of *GW*. List the elements of *GWRT*.

{(chocolate, sugar), (sugar, cinnamon), (cinnamon, nutmeg), (onions, cheese), (onions, peppers),
(chocolate, chocolate), (sugar, sugar), (cinnamon, cinnamon), (nutmeg, nutmeg), (onions, onions),
(cheese, cheese), (peppers, peppers),
(chocolate, cinnamon), (chocolate, nutmeg),
(sugar, nutmeg),
(onions, peppers)}

b) Is *GWRT* an equivalence relation? Prove your answer.

No, because it isn't symmetric. To be symmetric, it would have to contain, for example, (sugar, chocolate). But it doesn't.

Appendix B: The Theory: Working with Logical Formulas

1) Consider the following Boolean formula F:

   $\neg(P \vee R) \rightarrow (W \vee \neg S \vee \neg T)$ $\rightarrow$

   a) Using the procedure *conjunctiveBoolean* (described in the proof of Theorem B.1), construct a new formula F' that is equivalent to F and that is in conjunctive normal form.

   $\neg(P \vee R) \rightarrow (W \vee \neg S \vee \neg T)$
   $(\neg P \wedge \neg R) \vee (W \vee \neg S \vee \neg T)$
   $(\neg P \vee W \vee \neg S \vee \neg T) \wedge (\neg R \vee W \vee \neg S \vee \neg T)$

   b) Using the procedure *3-conjunctiveBoolean* (described in the proof of Theorem B.2), construct a formula F'' that is in 3-conjunctive normal form and that is satisfiable iff F' is.

   $(\neg P \vee W \vee \neg S \vee \neg T) \wedge (\neg R \vee W \vee \neg S \vee \neg T)$
   $(\neg P \vee W \vee Z_1) \wedge (\neg Z_1 \vee \neg S \vee \neg T) \wedge (\neg R \vee W \vee \neg S \vee \neg T)$
   $(\neg P \vee W \vee Z_1) \wedge (\neg Z_1 \vee \neg S \vee \neg T) \wedge (\neg R \vee W \vee Z_2) \wedge (\neg Z_2 \vee \neg S \vee \neg T)$

2) Show an example that proves that *conjunctiveBoolean* does not run in polynomial time. You can do this by describing a family of Boolean formulas with the property that the length of the output of *conjunctiveBoolean* grows exponentially with the length of the input.

   Let w be any formula of the form $(X_1 \wedge Y_1) \vee (X_2 \wedge Y_2) \vee (X_3 \wedge Y_3) \vee \ldots \vee (X_n \wedge Y_n)$. The output of *conjunctiveBoolean* on input w will have 2^n clauses.