The following problems ask you to prove some "obvious" claims about recursively-defined string functions. In each case, we want a self-contained, step-by-step induction proof that builds on formal definitions and prior results, *not* on intuition. In particular, your proofs must refer to the formal recursive definitions of string length and string concatenation:

$$|w| := \begin{cases} 0 & \text{if } w = \varepsilon \\ 1 + |x| & \text{if } w = ax \text{ for some symbol } a \text{ and some string } x \end{cases}$$

$$w \bullet z := \begin{cases} z & \text{if } w = \varepsilon \\ a \cdot (x \bullet z) & \text{if } w = ax \text{ for some symbol } a \text{ and some string } x \end{cases}$$

You may freely use the following results:

Lemma 1: $w \bullet \varepsilon = w$ for all strings w.

Lemma 2: $|w \bullet z| = |w| + |z|$ for all strings w and z.

Lemma 3: $(w \bullet y) \bullet z = w \bullet (y \bullet z)$ for all strings w, y, and z.

Inductive proofs of these lemmas (extracted directly from the lecture notes) appear on the following pages. Your inductive proofs should follow the general structure of these examples.

The ***reversal*** $\boldsymbol{w^R}$ of a string w is defined recursively as follows:

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = ax \text{ for some symbol } a \text{ and some string } x \end{cases}$$

For example, $\texttt{STRESSED}^R = \texttt{DESSERTS}$ and $\texttt{WTF374}^R = \texttt{473FTW}$.

1. Prove that $|w| = |w^R|$ for every string w.
2. Prove that $(w \bullet z)^R = z^R \bullet w^R$ for all strings w and z.
3. Prove that $(w^R)^R = w$ for every string w.

[Hint: The proof for problem 3 relies on problem 2, but it may be easier to solve problem 3 first.]

To think about later: Let $\#(a, w)$ denote the number of times symbol a appears in string w. For example, $\#(\texttt{X}, \texttt{WTF374}) = 0$ and $\#(\texttt{0}, \texttt{000010101010010100}) = 12$.

4. Give a formal recursive definition of $\#(a, w)$. (Your definition should have the same format as the definitions of $|w|$ and $w \bullet z$ at the top of this page.)
5. Prove that $\#(a, w \bullet z) = \#(a, w) + \#(a, z)$ for all symbols a and all strings w and z.
6. Prove that $\#(a, w^R) = \#(a, w)$ for all symbols a and all strings w.

Lemma 1. *$w \bullet \varepsilon = w$ for every string w.*

Proof: *Let w be an arbitrary string.*

Assume that $\boldsymbol{x \bullet \varepsilon = x}$ *for every string x such that $|x| < |w|$.*

There are two cases to consider:

- *Suppose $w = \varepsilon$.*

$$\begin{aligned} w \bullet \varepsilon &= \varepsilon \bullet \varepsilon && \text{because } w = \varepsilon, \\ &= \varepsilon && \text{by definition of } \bullet, \\ &= w && \text{because } w = \varepsilon. \end{aligned}$$

- *Suppose $w = ax$ for some symbol a and string x.*

$$\begin{aligned} w \bullet \varepsilon &= (a \cdot x) \bullet \varepsilon && \text{because } w = ax, \\ &= a \cdot (x \bullet \varepsilon) && \text{by definition of } \bullet, \\ &= a \cdot x && \text{by the inductive hypothesis,} \\ &= w && \text{because } w = ax. \end{aligned}$$

In both cases,

we conclude that $\boldsymbol{w \bullet \varepsilon = w}$. □

The nested boxes above try to emphasize this proof's *structure*. The *green italic* text is boilerplate for almost all string-induction proofs. The ***red bold*** text is the meat of the induction hypothesis and the result we're trying to prove. I'll use the same coloring in later proofs, but I'll omit the boxes.

We strongly recommend *writing* induction proofs "top down": Write all the boilerplate text in the larger boxes before thinking about what to write in smaller boxes. We also recommend writing the most general ("inductive") case before thinking about special ("base") cases, and writing the derivation for each case from both ends toward the middle.

Lemma 2. $|w \bullet z| = |w| + |z|$ *for all strings* w *and* z.

Proof: *Let* w *and* z *be arbitrary strings. Assume that* $|x \bullet z| = |x| + |z|$ *for every string* y *such that* $|x| < |w|$. (Notice that we are inducting only on w.) *There are two cases to consider:*

- *Suppose* $w = \varepsilon$.

$$
\begin{aligned}
|w \bullet z| &= |\varepsilon \bullet z| && \text{because } w = \varepsilon \\
&= |z| && \text{by definition of } \bullet \\
&= 0 + |z| && \text{by definition of } + \\
&= |\varepsilon| + |z| && \text{by definition of } |\cdot| \\
&= |w| + |z| && \text{because } w = \varepsilon
\end{aligned}
$$

- *Suppose* $w = ax$ *for some symbol* a *and string* x.

$$
\begin{aligned}
|w \bullet z| &= |ax \bullet z| && \text{because } w = ax \\
&= |a \cdot (x \bullet z)| && \text{by definition of } \bullet \\
&= 1 + |x \bullet z| && \text{by definition of } |\cdot| \\
&= 1 + |x| + |z| && \text{by the inductive hypothesis} \\
&= |ax| + |z| && \text{by definition of } |\cdot| \\
&= |w| + |z| && \text{because } w = ax
\end{aligned}
$$

In both cases, we conclude that $|w \bullet z| = |w| + |z|$. □

Lemma 3. $(w \bullet y) \bullet z = w \bullet (y \bullet z)$ *for all strings* w, y, *and* z.

Proof: *Let* w, y, *and* z *be arbitrary strings. Assume that* $\mathbf{(x \bullet y) \bullet z = x \bullet (y \bullet z)}$ *for every string* x *such that* $|x| < |w|$. (Notice again that we are inducting only on w.) *There are two cases to consider.*

- *Suppose* $w = \varepsilon$.

$$\begin{aligned}
(w \bullet y) \bullet z &= (\varepsilon \bullet y) \bullet z && \text{because } w = \varepsilon \\
&= y \bullet z && \text{by definition of } \bullet \\
&= \varepsilon \bullet (y \bullet z) && \text{by definition of } \bullet \\
&= w \bullet (y \bullet z) && \text{because } w = \varepsilon
\end{aligned}$$

- *Suppose* $w = ax$ *for some symbol* a *and some string* x.

$$\begin{aligned}
(w \bullet y) \bullet x &= (ax \bullet y) \bullet z && \text{because } w = ax \\
&= (a \cdot (x \bullet y)) \bullet z && \text{by definition of } \bullet \\
&= a \cdot ((x \bullet y) \bullet z) && \text{by definition of } \bullet \\
&= a \cdot (x \bullet (y \bullet z)) && \text{by the inductive hypothesis} \\
&= ax \bullet (y \bullet z) && \text{by definition of } \bullet \\
&= w \bullet (y \bullet z) && \text{because } w = ax
\end{aligned}$$

In both cases, we conclude that $\mathbf{(w \bullet y) \bullet z = w \bullet (y \bullet z)}$. □

Recall from lecture that a *regular expression* is compact notation for a language (that is, a set of strings). Formally, a regular language is one of the following:

- The symbol $\varnothing$ (representing the empty set)
- Any string (representing the set containing only that string)
- $\boldsymbol{R+S}$ for some regular expressions R and S (representing alternation / union)
- $\boldsymbol{R \cdot S}$ or $\boldsymbol{RS}$ for some regular expressions R and S (representing concatenation)
- $\boldsymbol{R}^*$ for some regular expression R (representing Kleene closure / unbounded repetition)

In the absence of parentheses, Kleene closure has highest precedence, followed by concatenation. For example, $1+01^* = \{0, 1, 01, 011, 0111, \ldots\}$, but $(1+01)^* = \{\varepsilon, 1, 01, 11, 011, 101, 111, 0101, \ldots\}$.

Give regular expressions for each of the following languages over the binary alphabet $\{0, 1\}$. (For extra practice, find multiple regular expressions for each language.)

0. All strings.
1. All strings containing the substring 000.
2. All strings *not* containing the substring 000.
3. All strings in which every run of 0s has length at least 3.
4. All strings in which every 1 appears before every substring 000.
5. All strings containing at least three 0s.
6. Every string except 000. *[Hint: Don't try to be clever.]*

More difficult problems to work on later:

7. All strings w such that *in every prefix of w*, the number of 0s and 1s differ by at most 1.

*8. All strings containing at least two 0s and at least one 1.

*9. All strings w such that *in every prefix of w*, the number of 0s and 1s differ by at most 2.

10. All strings in which every run has odd length. (For example, 0001 and 100000111 and the empty string ε are in this language, but 000000 and 001000 are not.)

★11. All strings in which the substring 000 appears an even number of times. (For example, 01100 and 000000 and the empty string ε are in this language, but 00000 and 001000 are not.)

Describe deterministic finite-state automata that accept each of the following languages over the alphabet $\Sigma = \{0, 1\}$. Give the states of your DFAs mnemonic names, and describe briefly *in English* the meaning or purpose of each state.

Either drawings or formal descriptions are acceptable, as long as the states Q, the start state s, the accept states A, and the transition function δ are all clear. Try not to use too many states, but *don't* try to use as few states as possible. Clarity is more important than brevity.

Yes, these are exactly the same languages that you saw last Friday.

---

0. All strings.
1. All strings containing the substring `000`.
2. All strings *not* containing the substring `000`.
3. All strings in which every run of `0`s has length at least 3.
4. All strings in which every `1` appears before every substring `000`.
5. All strings containing at least three `0`s.
6. Every string except `000`. *[Hint: Don't try to be clever.]*

More difficult problems to think about later:

7. All strings w such that *in every prefix of w*, the number of `0`s and `1`s differ by at most 1.
8. All strings containing at least two `0`s and at least one `1`.
9. All strings w such that *in every prefix of w*, the number of `0`s and `1`s differ by at most 2.
10. All strings in which every run has odd length. (For example, `0001` and `100000111` and the empty string ε are in this language, but `000000` and `001000` are not.)

*11. All strings in which the substring `000` appears an even number of times. (For example, `01100` and `000000` and the empty string ε are in this language, but `00000` and `001000` are not.)

Describe deterministic finite-state automata that accept each of the following languages over the alphabet $\Sigma = \{0, 1\}$. You may find it easier to describe these DFAs formally than to draw pictures.

Either drawings or formal descriptions are acceptable, as long as the states Q, the start state s, the accept states A, and the transition function δ are all clear. Try to keep the number of states small.

1. All strings in which the number of 0s is even **and** the number of 1s is *not* divisible by 3.

2. All strings in which the number of 0s is even **or** the number of 1s is *not* divisible by 3.

3. All strings that are **both** the binary representation of an integer divisible by 3 **and** the ternary (base-3) representation of an integer divisible by 4.

   For example, the string 1100 is an element of this language, because it represents $2^3 + 2^2 = 12$ in binary and $3^3 + 3^2 = 36$ in ternary.

Harder problems to think about later:

4. All strings in which the subsequence 0101 appears an even number of times.

5. All strings w such that $\binom{|w|}{2} \bmod 6 = 4$.
   [Hint: Maintain both $\binom{|w|}{2} \bmod 6$ *and* $|w| \bmod 6$.*]*
   [Hint: $\binom{n+1}{2} = \binom{n}{2} + n$.*]*

*6. All strings w such that $F_{\#(10,w)} \bmod 10 = 4$, where $\#(10, w)$ denotes the number of times 10 appears as a substring of w, and F_n is the nth Fibonacci number:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

Prove that each of the following languages is ***not*** regular, first using fooling sets and then (for problems 3, 4, and 5) using a reduction argument. You may use the fact (proven in class and in the lecture notes) that the language $\{0^n1^n \mid n \geq 0\}$ is not regular. See the next page for a solved example showing both types of proof.

1. $\left\{0^{2^n} \mid n \geq 0\right\}$

2. $\{0^{2n}1^n \mid n \geq 0\}$

3. $\{0^m1^n \mid m \neq 2n\}$

   [Hint: There is a short reduction argument, but write the fooling set argument first.]

4. Strings over $\{0, 1\}$ where the number of 0s is exactly twice the number of 1s.

   [Hint: There is a short reduction argument, but write the fooling set argument first.]

5. Strings of properly nested parentheses (), brackets [], and braces {}. For example, the string ([]){} is in this language, but the string ([)] is not, because the left and right delimiters don't match.

   [Hint: There is a short reduction argument, but write the fooling set argument first.]

Harder problems to think about later:

6. Strings of the form $w_1\#w_2\#\cdots\#w_n$ for some $n \geq 2$, where each substring w_i is a string in $\{0, 1\}^*$, and some pair of substrings w_i and w_j are equal.

7. $\left\{0^{n^2} \mid n \geq 0\right\}$

*8. $\{w \in (0 + 1)^* \mid w$ is the binary representation of a perfect square$\}$

Solved problem:

9. Prove that the language $L = \{w \in (0+1)^* \mid \#(0,w) = \#(1,w)\}$ is ***not*** regular.

Solution (fooling set 0^*):

Consider the infinite set $F = \{0^n \mid n \geq 0\}$, or more simply $F = 0^*$.

We claim that every pair of distinct strings in F has a distinguishing suffix.

> Let x and y be arbitrary distinct strings in F.
> The definition of F implies $x = 0^i$ and $y = 0^j$ for some integers $i \neq j$.
> Let z be the string 1^i.
> Then $xz = 0^i 1^i \in L$.
> But $yz = 0^j 1^i \notin L$, because $i \neq j$.
> So z is a distinguishing suffix for x and y.

We conclude that F is a fooling set for L.

Because F is infinite, L cannot be regular. ■

*This is **exactly** the proof from the lecture notes for the canonical non-regular language $\{0^n 1^n \mid n \geq 0\}$. The inner box is a proof that every pair of district strings in F has a distinguishing suffix.*

Solution (fooling set 0^*):

For any natural number n, let $x_n = 0^n$, and let $F = \{x_n \mid n \geq 0\} = 0^*$.

Let i and j be arbitrary distinct natural numbers.

Let z_{ij} be the string 1^i.

Then $x_i z_{ij} = 0^i 1^i \in L$.

But $x_j z_{ij} = 0^j 1^i \notin L$, because $i \neq j$.

So z_{ij} is a distinguishing suffix for x_i and x_j.

We conclude that F is a fooling set for L.

Because F is infinite, L cannot be regular. ■

*This is another way of writing exactly the same proof that emphasizes the counter intuition; any algorithm that recognizes L **must** count 0s.*

Solution (reduction via closure): For the sake of argument, suppose L is regular.

Then the language $L \cap 0^*1^* = \{0^n 1^n \mid n \geq 0\}$ would also be regular, because regular languages are closed under intersection.

But we proved in class that $\{0^n 1^n \mid n \geq 0\}$ is not regular; we've reached a contradiction.

We conclude that L cannot be regular.

*And this is **why** the proof for $\{0^n 1^n \mid n \geq 0\}$ also works verbatim for this language.* ■

*Starred subproblems cover advanced material that will ***not*** appear on homeworks or exams.

1. Let $L = \{w \in \{0,1\}^* \mid w \text{ starts and ends with } 0\}$.

   (a) Construct an NFA for L with exactly three states.

   (b) Convert the NFA you just constructed into a DFA using the incremental subset construction. Draw the resulting DFA. Your DFA should have four states, all reachable from the start state.

   (c) Convert the DFA you constructed in part (b) into a regular expression using the state elimination algorithm.

   (d) Write a simpler regular expression for L.

2. Let L be the set of all strings that contain either 001 or 011 as a substring.

   (a) Construct an NFA for L with exactly four states.

   (b) Convert the NFA you just constructed into a DFA using the incremental subset construction. Draw the resulting DFA. Your DFA should have eight states, all reachable from the start state.

   (c) Convert the NFA you constructed in part (a) into a regular expression using the state elimination algorithm.

3. (a) Convert the regular expression $(0^*1 + 01^*)^*$ into an NFA using Thompson's algorithm.

   (b) Convert the NFA you just constructed into a DFA using the incremental subset construction. Draw the resulting DFA. Your DFA should have four states, all reachable from the start state. (Some of these states are obviously equivalent, but keep them separate.)

   (c) Convert the DFA you constructed in part (b) into a regular expression using the state elimination algorithm. You should not get the same regular expression you started with.

   *(d) **Work on this later:** Find the smallest DFA that is equivalent to your DFA from part (b), using Moore's algorithm (in Section 3.6 of the notes).

   *(e) **Work on this later:** Convert the minimal DFA from part (d) into a regular expression using the state elimination algorithm. Again, you should not get the same regular expression you started with.

   (f) What is this language?

4. **Work on this later:**

   (a) Convert the regular expression $(\varepsilon + (0 + 11)^*0)1(11)^*$ into an NFA using Thompson's algorithm.

   (b) Convert the NFA you just constructed into a DFA using the incremental subset construction. Draw the resulting DFA. Your DFA should have six states, all reachable from the start state. (Some of these states are obviously equivalent, but keep them separate.)

(c) Convert the DFA you constructed in part (b) into a regular expression using the state elimination algorithm. You should *not* get the same regular expression you started with.

*(d) Find the smallest DFA that is equivalent to your DFA from part (b), using Moore's algorithm (in Section 3.6 of the notes).

*(e) Convert the minimal DFA from part (d) into a regular expression using the state elimination algorithm. Again, you should *not* get the same regular expression you started with.

(f) What is this language?

Let L be an arbitrary regular language over the alphabet $\Sigma = \{0, 1\}$. Prove that the following languages are also regular. (You probably won't get to all of these during the lab session.)

1. Let $\textsc{InsertAny1s}(L)$ is the set of all strings that can be obtained from strings in L by inserting ***any number of*** 1s anywhere in the string. For example:

   $$\textsc{InsertAny1s}(\{\varepsilon,\ 1,\ 00\}) = \{\varepsilon,\ 1,\ 11,\ 111, \ldots,\ 00,\ 100,\ 0111110, 1110111111101111, \ldots\}$$

   Prove that the language $\textsc{InsertAny1s}(L)$ is regular.

   Solution: Let $M = (Q, s, A, \delta)$ be an arbitrary DFA that accepts the regular language L. We construct a new ***NFA with ε-transitions*** $M' = (Q', s', A', \delta')$ that accepts $\textsc{InsertAny1s}(L)$ as follows.

   Intuitively, M' *guesses* which 1s in the input string have been inserted, skips over those 1s, and simulates M on the original string w. M' has the same states and start state and accepting states as M, but it has a different transition function.

   $$Q' = Q$$
   $$s' = s$$
   $$A' = A$$

   $$\delta'(q, 0) = \{\ \delta(q, 0)\ \}$$
   $$\delta'(q, 1) = \{ \qquad\qquad\qquad\qquad \}$$
   $$\delta'(q, \varepsilon) = \{ \qquad\qquad\qquad\qquad \}$$

   ■

2. Let DeleteAny1s(L) is the set of all strings that can be obtained from strings in L by inserting ***any number of*** 1s anywhere in the string. For example:

$$\text{DeleteAny1s}(\{\varepsilon,\ 00,\ 1101\}) = \{\varepsilon, 0, 00, 01, 10, 101, 110, 1101\}$$

Prove that the language DeleteAny1s(L) is regular.

Solution: Let $M = (Q, s, A, \delta)$ be an arbitrary DFA that accepts the regular language L. We construct a new ***NFA with ε-transitions*** $M' = (Q', s', A', \delta')$ that accepts $DeleteAny1s(L)$ as follows.

Intuitively, M' *guesses* where 1s have been deleted from its input string, and simulates the original machine M on the guessed mixture of input symbols and 1s. M' has the same states and start state and accepting states as M, but a different transition function.

$$Q' = Q$$
$$s' = s$$
$$A' = A$$

$$\delta'(q, 0) = \{\ \delta(q, 0)\ \}$$
$$\delta'(q, 1) = \{\qquad\qquad\qquad\qquad\qquad\}$$
$$\delta'(q, \varepsilon) = \{\qquad\qquad\qquad\qquad\qquad\}$$

■

3. Let $\text{InsertOne1}(L) := \{x1y \mid xy \in L\}$ denote the set of all strings that can be obtained from strings in L by inserting ***exactly one*** 1. For example:

$$\text{InsertOne1}(\{\varepsilon, 00, 101101\}) = \{1, 100, 010, 001, 1101101, 1011101, 1011011\}$$

Prove that the language $\text{InsertOne1}(L)$ is regular.

Solution: Let $M = (Q, s, A, \delta)$ be an arbitrary DFA that accepts the regular language L. We construct a new ***NFA with ε-transitions*** $M' = (Q', s', A', \delta')$ that accepts $\text{InsertOne1}(L)$ as follows.

If the input string w does not contain a 1, then M' must rejects it; otherwise, intuitively, M' *guesses* which 1 was inserted into w, skips over that 1, and simulates M on the remaining string xy.

M' consists of two copies of M, one to process the prefix x and the other to process the suffix y. State (q, False) means (the simulation of) M is in state q and M' has not yet skipped over a 1. State (q, True) means (the simulation of) M is in state q and M' has already skipped over a 1.

$$Q' = Q \times \{\text{True}, \text{False}\}$$
$$s' = (s, \text{False})$$
$$A' =$$

$$\delta'((q, \text{False}), 0) = \{\ (\delta(q, 0), \text{False})\ \}$$
$$\delta'((q, \text{False}), 1) = \{\qquad\qquad\qquad\}$$
$$\delta'((q, \text{False}), \varepsilon) = \{\qquad\qquad\qquad\}$$

$$\delta'((q, \text{True}), 0) = \{\qquad\qquad\qquad\}$$
$$\delta'((q, \text{True}), 1) = \{\qquad\qquad\qquad\}$$
$$\delta'((q, \text{True}), \varepsilon) = \{\qquad\qquad\qquad\}$$

■

4. Let $\text{DeleteOne1}(L) := \{xy \mid x1y \in L\}$ denote the set of all strings that can be obtained from strings in L by deleting exactly one 1. For example:

$$\text{DeleteOne1}(\{\varepsilon, 00, 101101\}) = \{01101, 10101, 10110\}$$

Prove that the language $\text{DeleteOne1}(L)$ is regular.

Solution: Let $M = (\Sigma, Q, s, A, \delta)$ be a DFA that accepts the regular language L. We construct an ***NFA with ε-transitions*** $M' = (\Sigma, Q', s', A', \delta')$ that accepts $\text{DeleteOne1}(L)$ as follows.

Intuitively, M' *guesses* where the 1 was deleted from its input string. It simulates the original DFA M on the prefix x before the missing 1, then the missing 1, and finally the suffix y after the missing 1.

M' consists of two copies of M, one to process the prefix x and the other to process the suffix y. State (q, False) means (the simulation of) M is in state q and M' has not yet reinserted a 1. State (q, True) means (the simulation of) M is in state q and M' has already reinserted a 1.

$$Q' = Q \times \{\text{True}, \text{False}\}$$
$$s' = (s, \text{False})$$
$$A' =$$

$$\delta'((q, \text{False}), 0) = \{\ (\delta(q, 0), \text{False})\ \}$$
$$\delta'((q, \text{False}), 1) = \{\qquad\qquad\qquad\qquad\}$$
$$\delta'((q, \text{False}), \varepsilon) = \{\qquad\qquad\qquad\qquad\}$$

$$\delta'((q, \text{True}), 0) = \{\qquad\qquad\qquad\qquad\}$$
$$\delta'((q, \text{True}), 1) = \{\qquad\qquad\qquad\qquad\}$$
$$\delta'((q, \text{True}), \varepsilon) = \{\qquad\qquad\qquad\qquad\}$$

■

Work on these later: Consider the following recursively defined function on strings:

$$evens(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \varepsilon & \text{if } w = a \text{ for some symbol } a \\ b \cdot evens(x) & \text{if } w = abx \text{ for some symbols } a \text{ and } b \text{ and some string } x \end{cases}$$

Intuitively, $evens(w)$ skips over every other symbol in w, starting with the first symbol. For example, $evens(\texttt{THE}\diamond\texttt{SNAIL}) = \texttt{H}\diamond\texttt{NI}$ and $evens(\texttt{GROB}\diamond\texttt{GOB}\diamond\texttt{GLOB}\diamond\texttt{GROD}) = \texttt{RBGBGO}\diamond\texttt{RD}$.

Let L be an arbitrary regular language over the alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$.

5. Prove that the language $\textsc{Unevens}(L) := \{w \mid evens(w) \in L\}$ is regular.

6. Prove that the language $\textsc{Evens}(L) := \{evens(w) \mid w \in L\}$ is regular..

You saw the following context-free grammars in class on Thursday; in each example, the grammar itself is on the left; the explanation for each non-terminal is on the right.

- Properly nested strings of parentheses.

$$S \to \varepsilon \mid S\,(S) \qquad \text{properly nested parentheses}$$

  Here is a different grammar for the same language:

$$S \to \varepsilon \mid (S) \mid SS \qquad \text{properly nested parentheses}$$

- $\{0^m 1^n \mid m \neq n\}$. This is the set of all binary strings composed of some number of 0s followed by a *different* number of 1s.

$$\begin{array}{ll} S \to A \mid B & \{0^m1^n \mid m \neq n\} \\ A \to 0A \mid 0C & \{0^m1^n \mid m > n\} \\ B \to B1 \mid C1 & \{0^m1^n \mid m < n\} \\ C \to \varepsilon \mid 0C1 & \{0^m1^n \mid m = n\} \end{array}$$

Give context-free grammars for each of the following languages over the alphabet $\Sigma = \{0, 1\}$. For each grammar, describe the language for each non-terminal, either in English or using mathematical notation, as in the examples above. We probably won't finish all of these during the lab session.

1. All palindromes in Σ^*

2. All palindromes in Σ^* that contain an even number of 1s

3. All palindromes in Σ^* that end with 1

4. All palindromes in Σ^* whose length is divisible by 3

5. All palindromes in Σ^* that do not contain the substring 00

Harder problems to work on later:

6. $\{0^{2n}1^n \mid n \geq 0\}$

7. $\{0^m 1^n \mid m \neq 2n\}$

   [Hint: If $m \neq 2n$, then either $m < 2n$ or $m > 2n$. Extend the previous grammar, but pay attention to parity. This language contains the string 01.]

8. $\{0,1\}^* \setminus \{0^{2n}1^n \mid n \geq 0\}$

   [Hint: Extend the previous grammar. What's missing?]

9. $\{w \in \{0,1\}^* \mid \#(0,w) = 2 \cdot \#(1,w)\}$ — Binary strings where the number of 0s is exactly twice the number of 1s.

*10. $\{0,1\}^* \setminus \{ww \mid w \in \{0,1\}^*\}$.

   *[Anti-hint: The language $\{ww \mid w \in 0, 1^*\}$ is **not** context-free. Thus, the complement of a context-free language is not necessarily context-free!]*

Let L be an arbitrary regular language over the alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$. Prove that the following languages are also regular.

1. $\textsc{Superstrings}(L) := \{xyz \mid y \in L \text{ and } x, z \in \Sigma^*\}$. This language contains all superstrings of strings in L. For example:

$$\textsc{Superstrings}(\{\texttt{10010}\}) = \left\{\underline{\texttt{10010}},\ \texttt{010}\underline{\texttt{10010}},\ \underline{\texttt{10010}}\texttt{11},\ \texttt{100}\underline{\texttt{10010}}\texttt{010},\ \cdots\right\}$$

[Hint: This is much easier than it looks.]

2. $\textsc{Substrings}(L) := \{y \mid x, y, z \in \Sigma^* \text{ and } xyz \in L\}$. This language contains all substrings of strings in L. For example:

$$\textsc{Substrings}(\{\texttt{10010}\}) = \left\{\varepsilon,\ \texttt{0},\ \texttt{1},\ \texttt{00},\ \texttt{01},\ \texttt{10},\ \texttt{001},\ \texttt{010},\ \texttt{100},\ \texttt{0010},\ \texttt{1001},\ \texttt{10010}\right\}$$

3. $\textsc{Cycle}(L) := \{xy \mid x, y \in \Sigma^* \text{ and } yx \in L\}$. This language contains all strings that can be obtained by splitting a string in L into a prefix and a suffix and concatenating them in the wrong order. For example:

$$\textsc{Cycle}(\{\texttt{OOK!},\ \texttt{OOKOOK}\}) = \left\{\texttt{OOK!},\ \texttt{OK!O},\ \texttt{K!OO},\ \texttt{!OOK},\ \texttt{OOKOOK},\ \texttt{OKOOKO},\ \texttt{KOOKOO}\right\}$$

Work on these later.

4. $\textsc{FlipOdds}(L) := \{\mathit{flipOdds}(w) \mid w \in L\}$, where the function *flipOdds* inverts every odd-indexed bit in w. For example:

$$\mathit{flipOdds}(\underline{\texttt{0}}\texttt{0}\underline{\texttt{0}}\texttt{0}\underline{\texttt{1}}\texttt{1}\underline{\texttt{1}}\texttt{1}\underline{\texttt{0}}\texttt{1}\underline{\texttt{0}}\texttt{1}\underline{\texttt{0}}\texttt{1}\underline{\texttt{0}}\texttt{0}) = \underline{\texttt{1}}\texttt{0}\underline{\texttt{1}}\texttt{0}\underline{\texttt{0}}\texttt{1}\underline{\texttt{0}}\texttt{1}\underline{\texttt{1}}\texttt{1}\underline{\texttt{1}}\texttt{1}\underline{\texttt{1}}\texttt{1}\underline{\texttt{1}}\texttt{0}$$

5. $\textsc{UnflipOdd}\texttt{1}\textsc{s}(L) := \{w \in \Sigma^* \mid \mathit{flipOdd}\texttt{1}\mathit{s}(w) \in L\}$, where the function *flipOdd*1 inverts every other 1 bit of its input string, starting with the first 1. For example:

$$\mathit{flipOdd}\texttt{1}\mathit{s}(\texttt{0000}\underline{\texttt{1}}\texttt{1}\underline{\texttt{1}}\texttt{100}\underline{\texttt{1}}\texttt{010}\underline{\texttt{1}}\texttt{0}) = \texttt{0000}\underline{\texttt{0}}\texttt{1}\underline{\texttt{0}}\texttt{100}\underline{\texttt{0}}\texttt{010}\underline{\texttt{0}}\texttt{0}$$

6. $\textsc{FlipOdd}\texttt{1}\textsc{s}(L) := \{\mathit{flipOdd}\texttt{1}\mathit{s}(w) \mid w \in L\}$, where the function *flipOdd*1*s* is defined in the previous problem.

Here are several problems that are easy to solve in $O(n)$ time, essentially by brute force. Your task is to design algorithms for these problems that are significantly faster.

1. Suppose we are given an array $A[1..n]$ of n distinct integers, which could be positive, negative, or zero, sorted in increasing order so that $A[1] < A[2] < \cdots < A[n]$.

   (a) Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

   (b) Suppose we know in advance that $A[1] > 0$. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. *[Hint: This is **really** easy.]*

2. Suppose we are given an array $A[1..n]$ such that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a ***local minimum*** if both $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are exactly six local minima in the following array:

9	**7**	7	2	**1**	3	7	5	**4**	7	**3**	**3**	4	8	**6**	9
	▲			▲				▲		▲	▲			▲	

   Describe and analyze a fast algorithm that returns the index of one local minimum. For example, given the array above, your algorithm could return the integer 9, because $A[9]$ is a local minimum. *[Hint: With the given boundary conditions, any array **must** contain at least one local minimum. Why?]*

3. Suppose you are given two sorted arrays $A[1..n]$ and $B[1..n]$ containing distinct integers. Describe a fast algorithm to find the median (meaning the nth smallest element) of the union $A \cup B$. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1..8] = [2, 4, 5, 8, 17, 19, 21, 23]$$

   your algorithm should return the integer 9. *[Hint: What can you learn by comparing one element of A with one element of B?]*

Harder problem to think about later:

4. Now suppose you are given two sorted arrays $A[1..m]$ and $B[1..n]$ and an integer k. Describe a fast algorithm to find the kth smallest element in the union $A \cup B$. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1..5] = [2, 5, 7, 17, 19] \qquad k = 6$$

   your algorithm should return the integer 7.

In lecture on Thursday, we saw a divide-and-conquer algorithm, due to Karatsuba, that multiplies two n-digit integers using $O(n^{\lg 3})$ single-digit additions, subtractions, and multiplications. In this lab, we'll look at one application of Karatsuba's algorithm: converting a number from binary to decimal.

(The standard algorithm that computes one decimal digit of x at a time, by computing $x \bmod 10$ and then recursively converting $\lfloor x/10 \rfloor$, requires $\Theta(n^2)$ time.)

1. Consider the following recurrence, originally used by the Sanksrit prosodist Piṅgala in the second century BCE, to compute the number 2^n:

$$2^n = \begin{cases} 1 & \text{if } n = 0 \\ (2^{n/2})^2 & \text{if } n > 0 \text{ is even} \\ 2 \cdot (2^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is odd} \end{cases}$$

   We can use this algorithm to compute the decimal representation of 2^n, by representing all numbers using arrays of decimal digits, and implementing squaring and doubling using decimal arithmetic. Suppose we use Karatsuba's algorithm for decimal multiplication. What is the running time of the resulting algorithm?

2. We can use a similar algorithm to convert the binary representation of any integer into its decimal representation. Suppose we are given an integer x as an array of n bits (binary digits). Write $x = a \cdot 2^{n/2} + b$, where a is represented by the top $n/2$ bits of x, and b is represented by the bottom $n/2$ bits of x. Then we can convert x into decimal as follows:

   (a) Recursively convert a into decimal.
   (b) Recursively convert $2^{n/2}$ into decimal.
   (c) Recursively convert b into decimal.
   (d) Compute $x = a \cdot 2^{n/2} + b$ using decimal multiplication and addition.

   Now suppose we use Karatsuba's algorithm for decimal multiplication. What is the running time of the resulting algorithm? (For simplicity, you can assume n is a power of 2.)

3. Now suppose instead of converting $2^{n/2}$ to decimal by recursively calling the algorithm from problem 2, we use the specialized algorithm for powers of 2 from problem 1. Now what is the running time of the resulting algorithm (assuming we use Karatsuba's multiplication algorithm as before)?

Harder problem to think about about later:

4. In fact, it is possible to multiply two n-digit decimal numbers in $O(n \log n)$ time. Describe an algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(n \log^2 n)$ time.

> The cost is the effort to review the code before it can go on and the effort to maintain it forever after. Please don't do this.
> — Guido van Rossum (January 23, 2022)

A ***subsequence*** of a sequence (for example, an array, linked list, or string), obtained by removing zero or more elements and keeping the rest in the same sequence order. A subsequence is called a ***substring*** if its elements are contiguous in the original sequence. For example:

- `SUBSEQUENCE`, `UBSEQU`, and the empty string ε are all substrings (and therefore subsequences) of the string `SUBSEQUENCE`;
- `SBSQNC`, `SQUEE`, and `EEE` are all subsequences of `SUBSEQUENCE` but not substrings;
- `QUEUE`, `EQUUS`, and `DIMAGGIO` are not subsequences (and therefore not substrings) of `SUBSEQUENCE`.

Describe ***recursive backtracking*** algorithms for the following longest-subsequence problems. *Don't worry about running times.*

1. Given an array $A[1..n]$ of integers, compute the length of a longest ***increasing*** subsequence. A sequence $B[1..\ell]$ is *increasing* if $B[i] > B[i-1]$ for every index $i \geq 2$.

   For example, given the array

   $$\langle 3, \underline{\mathbf{1}}, \underline{\mathbf{4}}, 1, \underline{\mathbf{5}}, 9, 2, \underline{\mathbf{6}}, 5, 3, 5, \underline{\mathbf{8}}, 9, 7, \underline{\mathbf{9}}, 3, 2, 3, 8, 4, 6, 2, 7 \rangle$$

   your algorithm should return the integer 6, because $\langle 1, 4, 5, 6, 8, 9 \rangle$ is a longest increasing subsequence (one of many).

2. Given an array $A[1..n]$ of integers, compute the length of a longest ***decreasing*** subsequence. A sequence $B[1..\ell]$ is *decreasing* if $B[i] < B[i-1]$ for every index $i \geq 2$.

   For example, given the array

   $$\langle 3, 1, 4, 1, 5, \underline{\mathbf{9}}, 2, \underline{\mathbf{6}}, 5, 3, \underline{\mathbf{5}}, 8, 9, 7, 9, 3, 2, 3, 8, \underline{\mathbf{4}}, 6, \underline{\mathbf{2}}, 7 \rangle$$

   your algorithm should return the integer 5, because $\langle 9, 6, 5, 4, 2 \rangle$ is a longest decreasing subsequence (one of many).

3. Given an array $A[1..n]$ of integers, compute the length of a longest ***alternating*** subsequence. A sequence $B[1..\ell]$ is *alternating* if $B[i] < B[i-1]$ for every even index $i \geq 2$, and $B[i] > B[i-1]$ for every odd index $i \geq 3$.

   For example, given the array

   $$\langle \underline{\mathbf{3}}, \underline{\mathbf{1}}, \underline{\mathbf{4}}, \underline{\mathbf{1}}, \underline{\mathbf{5}}, 9, \underline{\mathbf{2}}, \underline{\mathbf{6}}, \underline{\mathbf{5}}, 3, 5, \underline{\mathbf{8}}, 9, \underline{\mathbf{7}}, \underline{\mathbf{9}}, \underline{\mathbf{3}}, 2, 3, \underline{\mathbf{8}}, \underline{\mathbf{4}}, \underline{\mathbf{6}}, \underline{\mathbf{2}}, \underline{\mathbf{7}} \rangle$$

   your algorithm should return the integer 17, because $\langle 3, 1, 4, 1, 5, 2, 6, 5, 8, 7, 9, 3, 8, 4, 6, 2, 7 \rangle$ is a longest alternating subsequence (one of many).

Harder problems to think about later:

4. Given an array $A[1..n]$ of integers, compute the length of a longest ***convex*** subsequence of A. A sequence $B[1..\ell]$ is *convex* if $B[i] - B[i-1] > B[i-1] - B[i-2]$ for every index $i \geq 3$.

   For example, given the array

   $$\langle \underline{\mathbf{3}}, \underline{\mathbf{1}}, 4, \underline{\mathbf{1}}, 5, 9, \underline{\mathbf{2}}, 6, 5, 3, \underline{\mathbf{5}}, 8, \underline{\mathbf{9}}, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7 \rangle$$

   your algorithm should return the integer 6, because $\langle 3, 1, 1, 2, 5, 9 \rangle$ is a longest convex subsequence (one of many).

5. Given an array $A[1..n]$, compute the length of a longest ***palindrome*** subsequence of A. Recall that a sequence $B[1..\ell]$ is a *palindrome* if $B[i] = B[\ell - i + 1]$ for every index i.

   For example, given the array

   $$\langle 3, 1, \underline{\mathbf{4}}, 1, 5, \underline{\mathbf{9}}, 2, 6, \underline{\mathbf{5}}, \underline{\mathbf{3}}, \underline{\mathbf{5}}, 8, 9, 7, \underline{\mathbf{9}}, 3, 2, 3, 8, \underline{\mathbf{4}}, 6, 2, 7 \rangle$$

   your algorithm should return the integer 7, because $\langle 4, 9, 5, 3, 5, 9, 4 \rangle$ is a longest palindrome subsequence (one of many).

A ***subsequence*** of a sequence (for example, an array, a linked list, or a string), obtained by removing zero or more elements and keeping the rest in the same sequence order. A subsequence is called a ***substring*** if its elements are contiguous in the original sequence. For example:

- `SUBSEQUENCE`, `UBSEQU`, and the empty string ε are all substrings of `SUBSEQUENCE`;
- `SBSQNC`, `UEQUE`, and `EEE` are all subsequences of `SUBSEQUENCE` but not substrings;
- `QUEUE`, `SSS`, and `FOOBAR` are not subsequences of `SUBSEQUENCE`.

Describe and analyze ***dynamic programming*** algorithms for the following longest-subsequence problems. Use the recurrences you developed on Wednesday.

1. Given an array $A[1..n]$ of integers, compute the length of a longest ***increasing*** subsequence of A. A sequence $B[1..\ell]$ is *increasing* if $B[i] > B[i-1]$ for every index $i \geq 2$.

2. Given an array $A[1..n]$ of integers, compute the length of a longest ***decreasing*** subsequence of A. A sequence $B[1..\ell]$ is *decreasing* if $B[i] < B[i-1]$ for every index $i \geq 2$.

3. Given an array $A[1..n]$ of integers, compute the length of a longest ***alternating*** subsequence of A. A sequence $B[1..\ell]$ is *alternating* if $B[i] < B[i-1]$ for every even index $i \geq 2$, and $B[i] > B[i-1]$ for every odd index $i \geq 3$.

4. Given an array $A[1..n]$ of integers, compute the length of a longest ***convex*** subsequence of A. A sequence $B[1..\ell]$ is *convex* if $B[i] - B[i-1] > B[i-1] - B[i-2]$ for every index $i \geq 3$.

5. Given an array $A[1..n]$, compute the length of a longest ***palindrome*** subsequence of A. Recall that a sequence $B[1..\ell]$ is a *palindrome* if $B[i] = B[\ell - i + 1]$ for every index i.

Basic steps in developing a dynamic programming algorithm

1. **Formulate the problem recursively.** This is the hard part. There are two distinct but equally important things to include in your formulation.

   (a) **Specification.** First, give a clear and precise English description of the problem you are claiming to solve. Not *how* to solve the problem, but *what* the problem actually is. Omitting this step in homeworks or exams will cost you significant points.

   (b) **Solution.** Second, give a clear recursive formula or algorithm for the whole problem in terms of the answers to smaller instances of *exactly* the same problem. It generally helps to think in terms of a recursive definition of your inputs and outputs. If you discover that you need a solution to a *similar* problem, or a slightly *related* problem, you're attacking the wrong problem; go back to step 1.

   (c) **Don't optimize prematurely.** It may be tempting to ignore "obviously" suboptimal choices, because that will yield an "obviously" faster algorithm, but it's usually a bad idea, for two reasons. First, the optimization may not actually improve the running time of the final dynamic programming algorithm. But more importantly, many "obvious" optimizations are actually incorrect! ***First* make it work; *then* optimize.**

2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order.

   (a) **Identify the subproblems.** What are all the different ways can your recursive algorithm call itself, starting with some initial input?

   (b) **Analyze running time.** Add up the running times of all possible subproblems, *ignoring the recursive calls*.

   (c) **Choose a memoization data structure.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. But some problems need a more complicated data structure.

   (d) **Identify dependencies.** Except for the base cases, every recursive subproblem depends on other subproblems—which ones? Draw a picture of your data structure, pick a generic element, and draw arrows from each of the other elements it depends on. Then formalize your picture.

   (e) **Find a good evaluation order.** Order the subproblems so that each subproblem comes *after* the subproblems it depends on. Typically, you should consider the base cases first, then the subproblems that depends only on base cases, and so on. ***Be careful!***

   (f) **Write down the algorithm.** You know what order to consider the subproblems, and you know how to solve each subproblem. So do that! If your data structure is an array, this usually means writing a few nested for-loops around your original recurrence.

3. **Try to improve.** What's the bottleneck in your algorithm? Can you find a faster algorithm by modifying the recurrence? Can you tighten the time analysis? *Now* is the time to think about removing "obviously" redundant or suboptimal choices. (But always make sure that your optimizations are correct!!)

Amy Nancato, the founding director of the new Parisa Tabriz School of Computing and Data Science, has commissioned a series of snow ramps on the south slope of the Orchard Downs sledding hill[1] and challenged Erhan Hajek, head of the Department of Electrical and Computer Engineering, to a sledding contest. Erhan and Amy will both sled down the hill, each trying to maximize their air time. The winner gets to expand their unit into Siebel Center, the ECE Building, *and* the new Campus Instructional Facility; the loser has to move their entire unit under the Boneyard bridge behind Everitt Lab.

Whenever Amy or Erhan reaches a ramp *while on the ground*, they can either use that ramp to jump through the air, possibly flying over one or more ramps, or sled past that ramp and stay on the ground. Obviously, if someone flies over a ramp, they cannot use that ramp to extend their jump.

1. Suppose you are given a pair of arrays $Ramp[1..n]$ and $Length[1..n]$, where $Ramp[i]$ is the distance from the top of the hill to the ith ramp, and $Length[i]$ is the distance that any sledder who takes the ith ramp will travel through the air.

   Describe and analyze an algorithm to determine the maximum *total* distance that Erhan or Amy can travel through the air.

2. Uh-oh. The university lawyers heard about Amy and Erhan's little bet and immediately objected. To protect the university from both lawsuits and sky-rocketing insurance rates, they impose an upper bound on the number of jumps that either sledder can take.

   Describe and analyze an algorithm to determine the maximum total distance that Amy or Erhan can spend in the air *with at most k jumps*, given the original arrays $Ramp[1..n]$ and $Length[1..n]$ and the integer k as input.

Harder problem to think about later:

3. When the lawyers realized that imposing their restriction didn't immediately shut down the contest, they added yet another restriction: No ramp may be used more than once! Disgusted by all the legal interference, Erhan and Amy give up on their bet and decide to cooperate to put on a good show for the spectators.

   Describe and analyze an algorithm to determine the maximum total distance that Amy and Erhan can spend in the air, each taking at most k jumps (so at most $2k$ jumps total), and with each ramp used at most once.

[1]The north slope is faster, but too short for an interesting contest.

1. A ***basic arithmetic expression*** is composed of characters from the set $\{1, +, \times\}$ and parentheses. Almost every integer can be represented by more than one basic arithmetic expression. For example, all of the following basic arithmetic expression represent the integer 14:

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1$$
$$((1+1)\times(1+1+1+1+1))+((1+1)\times(1+1))$$
$$(1+1)\times(1+1+1+1+1+1+1)$$
$$(1+1)\times(((1+1+1)\times(1+1))+1)$$

Describe and analyze an algorithm to compute, given an integer n as input, the minimum number of 1's in a basic arithmetic expression whose value is equal to n. The number of parentheses doesn't matter, just the number of 1's. For example, when $n = 14$, your algorithm should return 8, for the final expression above. The running time of your algorithm should be bounded by a small polynomial function of n.

Harder problem to think about later:

2. Suppose you are given a sequence of integers separated by + and − signs; for example:

$$1+3-2-5+1-6+7$$

You can change the value of this expression by adding parentheses in different places. For example:

$$1+3-2-5+1-6+7=-1$$
$$(1+3-(2-5))+(1-6)+7=9$$
$$(1+(3-2))-(5+1)-(6+7)=-17$$

Describe and analyze an algorithm to compute, given a list of integers separated by + and − signs, the maximum possible value the expression can take by adding parentheses. Parentheses must be used only to group additions and subtractions; in particular, do not use them to create implicit multiplication as in $1+3(-2)(-5)+1-6+7=33$.

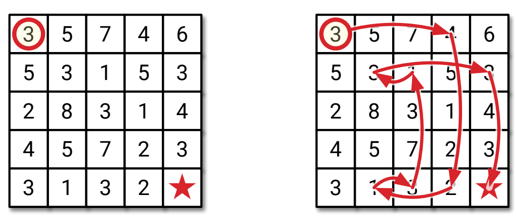
For each of the problems below, transform the input into a graph and apply a standard graph algorithm that you've seen in class. Whenever you use a standard graph algorithm, you ***must*** provide the following information. (I recommend actually using a bulleted list.)

- What are the vertices? What does each vertex represent?
- What are the edges? Are they directed or undirected?
- If the vertices and/or edges have associated values, what are they?
- **What is the precise relationship between your graph and the stated problem?**
- What standard graph algorithm are you using to solve the problem?
- What is the running time of your entire algorithm, *including* the time to build the graph, *as a function of the original input parameters*?

Finally, it is crucial to remember that even when you are explicitly given a graph as part of the input, that may not be the graph you actually want to search!

---

1. A ***number maze*** is an $n \times n$ grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner. On each turn, you are allowed to move the token up, down, left, or right; the distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three steps down, three steps left, or three steps right. However, you are never allowed to move the token off the edge of the board.



A 5 × 5 number maze that can be solved in eight moves.

Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution. For example, given the number maze shown above, your algorithm should return the integer 8. Your input is a two-dimensional array $M[1..n, 1..n]$ of positive integers.

The remaining problems (starting on the next page) consider variants of problem 1, where the sequence of moves must satisfy certain constraints to be considered a valid solution. For each problem, your goal is to describe and analyze an algorithm that either returns the minimum number of moves in a *valid* solution to a given number maze, or reports correctly that no valid solution exists.

2. Suppose a sequence of moves is considered *valid* if and only if the moves alternate between horizontal and vertical. That is, a valid move sequence never has two horizontal moves in a row or two vertical moves in a row.

   Describe and analyze an algorithm that either returns the minimum number of moves in a valid solution to a given number maze, or reports correctly that no valid solution exists.

3. Suppose a sequence of moves is considered *valid* if and only if its length (the number of moves) is a multiple of 5.

   Describe and analyze an algorithm that either returns the minimum number of moves in a valid solution to a given number maze, or reports correctly that no valid solution exists.

Harder problems to think about later:

4. Now suppose a sequence of moves is considered *valid* if and only if it does not contain two adjacent moves in opposite directions. In other words, a sequence of moves is valid if and only if it contains no U-turns.

   Describe and analyze an algorithm that either returns the minimum number of moves in a valid solution to a given number maze, or reports correctly that no valid solution exists.

5. Now suppose a sequence of moves is considered *valid* if and only if each move in the sequence is longer than the previous move (if any). In other words, a sequence of moves is valid if and only if the sequence of move *lengths* is increasing.

   Describe and analyze an algorithm that either returns the minimum number of moves in a valid solution to a given number maze, or reports correctly that no valid solution exists.

6. Finally, suppose a sequence of moves is considered *valid* if and only if the sequence of move lengths is a palindrome. (A palindrome is any sequence that is equal to its reversal.)

   Describe and analyze an algorithm that either returns the minimum number of moves in a valid solution to a given number maze, or reports correctly that no valid solution exists.

For each of the problems below, transform the input into a graph and apply a standard graph algorithm that you've seen in class. Whenever you use a standard graph algorithm, you ***must*** provide the following information. (I recommend actually using a bulleted list.)

- What are the vertices? What does each vertex represent?
- What are the edges? Are they directed or undirected?
- If the vertices and/or edges have associated values, what are they?
- **What is the precise relationship between your graph and the stated problem?**
- What standard graph algorithm are you using to solve the problem?
- What is the running time of your entire algorithm, *including* the time to build the graph, *as a function of the original input parameters*?

Finally, it is crucial to remember that even when you are explicitly given a graph as part of the input, that may not be the graph you actually want to search!

---

1. Suppose you decide to organize a Snakes and Ladders competition with n participants. In this competition, each game of Snakes and Ladders involves three players. After the game is finished, they are ranked first, second, and third. Each player may be involved in any (non-negative) number of games, and the number need not be equal among players.

   At the end of the competition, m games have been played. You realize that you forgot to implement a proper rating system, and therefore decide to produce the overall ranking of all n players as you see fit. However, to avoid being too suspicious, if player A ranked better than player B in at least one game, then A must rank better than B in the overall ranking.

   You are given the list of players and their rankings in each of the m games. Describe and analyze an algorithm that produces an overall ranking of the n players that is consistent with the individual game rankings, or correctly reports that no such ranking exists.

2. There are n galaxies connected by m intergalactic teleport-ways. Each teleport-way joins two galaxies and can be traversed in both directions. However, the company that runs the teleport-ways has established an extremely lucrative cost structure: Anyone can teleport *further* from their home galaxy at no cost whatsoever, but teleporting *toward* their home galaxy is prohibitively expensive.

   Judy has decided to take a sabbatical tour of the universe by visiting as many galaxies as possible, starting at her home galaxy. To save on travel expenses, she wants to teleport away from her home galaxy at every step, except for the very last teleport home.

   Describe and analyze an algorithm to compute the maximum number of galaxies that Judy can visit. Your input consists of an undirected graph G with n vertices and m edges describing the teleport-way network, an integer $1 \le s \le n$ identifying Judy's home galaxy, and an array $D[1..n]$ containing the distances of each galaxy from s.

Harder problems to think about later:

3. Just before embarking on her universal tour, Judy wins the space lottery, giving her just enough money to afford *two* teleports toward her home galaxy. Describe and analyze a new algorithm to compute the maximum number of galaxies Judy can visit; if she visits the same galaxy twice, that counts as two visits. After all, argues the travel agent, who can see an entire galaxy in just one visit?

*4. Judy replies angrily to the travel agent that *she* can see an entire galaxy in just one visit, because 99% of every galaxy is exactly the same glowing balls of plasma and lifeless chunks of rock and McDonalds and Starbucks and prefab "Irish" pubs and overpriced souvenir shops and Peruvian street-corner musicians as every other galaxy.

   Describe and analyze an algorithm to compute the maximum number of *distinct* galaxies Judy can visit. She is still *allowed* to visit the same galaxy more than once, but only the first visit counts toward her total.

1. Describe and analyze an algorithm to compute the shortest path from vertex s to vertex t in a directed graph with weighted edges, where exactly *one* edge $u \to v$ has negative weight. Assume the graph has no negative cycles. *[Hint: Modify the input graph and run Dijkstra's algorithm.] [Hint: Alternatively,* ***don't*** *modify the input graph, but run Dijkstra's algorithm anyway.]*

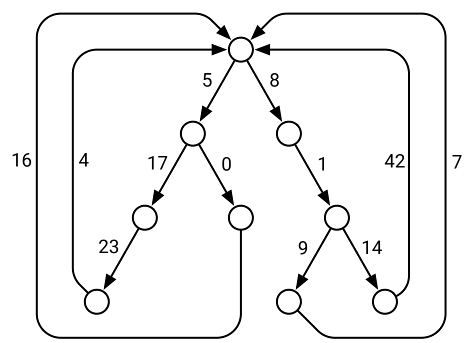
2. You just discovered your best friend from elementary school on Face𝕏 (formerly known as Twitbook). You both want to meet as soon as possible, but you live in two different cities that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving toward each other. But where *exactly* should you meet?

   You are given a weighted graph $G = (V, E)$, where the vertices V represent cities and the edges E represent roads that directly connect cities. Each edge e has a weight $w(e)$ equal to the time required to travel between the two cities. You are also given a vertex p, representing your starting location, and a vertex q, representing your friend's starting location.

   Describe and analyze an algorithm to find the target vertex t that allows you and your friend to meet as soon as possible, assuming both of you leave home *right now*.

Think about later:

3. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has a non-negative weight.



A looped tree.

   (a) How much time would Dijkstra's algorithm require to compute the shortest path between two vertices u and v in a looped tree with n nodes?

   (b) Describe and analyze a faster algorithm.

1. Suppose that you have just finished computing the array $dist[1..V, 1..V]$ of shortest-path distances between ***all*** pairs of vertices in an edge-weighted directed graph G. Unfortunately, you discover that you incorrectly entered the weight of a single edge $u \to v$, so all that precious CPU time was wasted. Or was it? Maybe your distances are correct after all!

   In each of the following problems, let $w(u \to v)$ denote the weight that you used in your distance computation, and let $w'(u \to v)$ denote the correct weight of $u \to v$.

   (a) Suppose $w(u \to v) > w'(u \to v)$; that is, the weight you used for $u \to v$ was *larger* than its true weight. Describe an algorithm that repairs the distance array in ***$O(V^2)$ time*** under this assumption. *[Hint: For every pair of vertices x and y, either $u \to v$ is on the shortest path from x to y or it isn't.]*

   (b) Maybe even that was too much work. Describe an algorithm that determines whether your original distance array is actually correct in ***$O(1)$ time***, again assuming that $w(u \to v) > w'(u \to v)$. *[Hint: Either $u \to v$ is the shortest path from u to v or it isn't.]*

   (c) **To think about later:** Describe an algorithm that determines in ***$O(VE)$ time*** whether your distance array is actually correct, even if $w(u \to v) < w'(u \to v)$.

   (d) **To think about later:** Argue that when $w(u \to v) < w'(u \to v)$, repairing the distance array *requires* recomputing shortest paths from scratch, at least in the worst case.

2. You—yes, *you*—can cause a major economic collapse with the power of graph algorithms![1] The *arbitrage* business is a money-making scheme that takes advantage of differences in currency exchange. In particular, suppose that 1 US dollar buys 120 Japanese yen; 1 yen buys 0.01 euros; and 1 euro buys 1.2 US dollars. Then, a trader starting with \$1 can convert their money from dollars to yen, then from yen to euros, and finally from euros back to dollars, ending with \$1.44! The cycle of currencies $\$ \to ¥ \to € \to \$$ is called an ***arbitrage cycle***. Of course, finding and exploiting arbitrage cycles before the prices are corrected requires extremely fast algorithms.

   Suppose n different currencies are traded in your currency market. You are given a two-dimensional array $R[1..n, 1..n]$ containing exchange rates between every pair of currencies; for each i and j, one unit of currency i can be traded for $R[i, j]$ units of currency j. (Do *not* assume that $R[i, j] \cdot R[j, i] = 1$.)

   (a) Describe an algorithm that returns an array $V[1..n]$, where $V[i]$ is the maximum amount of currency i that you can obtain by trading, starting with one unit of currency 1, assuming there are no arbitrage cycles.

   (b) Describe an algorithm to determine whether the given matrix of currency exchange rates creates an arbitrage cycle.

   *(c) **To think about later:** Modify your algorithm from part (b) to actually return an arbitrage cycle, if such a cycle exists.

[1] No, you can't.

1. Suppose you are given an array of numbers, some of which are marked as *icky*, and you want to compute the length of the longest increasing subsequence of A that includes at most k icky numbers. Your input consists of the integer k, the number array $A[1..n]$, and another boolean array $Icky[1..n]$.

   For example, suppose your input consists of the integer $k = 2$ and the following array (with icky numbers are indicated by stars):

$3^\star$	$1^\star$	4	$1^\star$	$5^\star$	9	$2^\star$	6	5	$3^\star$	5	9	7	$9^\star$	3	2	3	$8^\star$	4	$6^\star$	2	$6^\star$

   Then your algorithm should return the integer 5, which is the length of the increasing subsequence $4, 5^\star, 6, 7, 9^\star$.

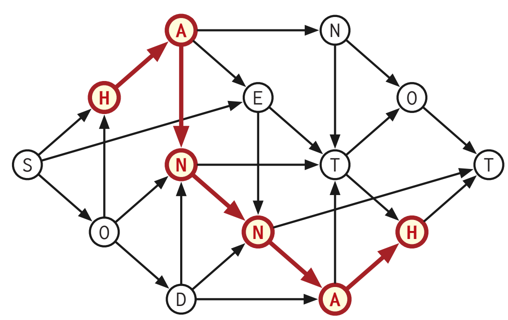
   (a) Describe an algorithm for this problem using dynamic programming.

   (b) Describe an algorithm for this problem by reducing it to a standard graph problem.

Think about later:

2. Let G be a directed acyclic graph whose vertices have labels from some fixed alphabet. Any directed path in G has a label, which is a string obtained by concatenating the labels of its vertices. Recall that a *palindrome* is a string that is equal to its reversal.

   Describe and analyze an algorithm to find the length of the longest palindrome that is the label of a path in G. For example, given the dag below, your algorithm should return the integer 6, which is the length of the palindrome HANNAH.



   (a) Describe an algorithm for this problem using dynamic programming.

   (b) Describe an algorithm for this problem by reducing it to a standard graph problem.

1. Suppose you are given a magic black box that somehow answers the following decision problem *in polynomial time*:

   - Input: A directed graph G and a positive integer L. (The edges of G are not weighted, and G is not necessarily a dag.)
   - Output: True if G contains a (simple) path of length L, and False otherwise.[1]

   (a) Using this black box as a subroutine, describe algorithms that solves the following optimization problem *in polynomial time*:

   - Input: A directed graph G.
   - Output: The length of the longest path in G.

   (b) Using this black box as a subroutine, describe algorithms that solves the following search problem *in polynomial time*:

   - Input: A directed graph G.
   - Output: The longest path in G

   [Hint: You can use the magic box more than once.]

2. An ***independent set*** in a graph G is a subset S of the vertices of G, such that no two vertices in S are connected by an edge in G. Suppose you are given a magic black box that somehow answers the following decision problem *in polynomial time*:

   - Input: An undirected graph G and an integer k.
   - Output: True if G has an independent set of size k, and False otherwise.[2]

   (a) Using this black box as a subroutine, describe algorithms that solves the following optimization problem *in polynomial time*:

   - Input: An undirected graph G.
   - Output: The size of the largest independent set in G.

   (b) Using this black box as a subroutine, describe algorithms that solves the following search problem *in polynomial time*:

   - Input: An undirected graph G.
   - Output: An independent set in G of maximum size.

   [Hint: You can use the magic box more than once.]

[1] You already know how to solve this problem in polynomial time *when the input graph G is a dag*, but this magic box works for *every* input graph.

[2] It is not hard to solve this problem in polynomial time via dynamic programming when the input graph G is a *tree*, but this magic box works for *every* input graph.

To think about later:

3. Formally, a ***proper coloring*** of a graph $G = (V, E)$ is a function $c: V \to \{1, 2, \dots, k\}$, for some integer k, such that $c(u) \neq c(v)$ for all $uv \in E$. Less formally, a valid coloring assigns each vertex of G a color, such that every edge in G has endpoints with different colors. The ***chromatic number*** of a graph is the minimum number of colors in a proper coloring of G.

   Suppose you are given a magic black box that somehow answers the following decision problem *in polynomial time*:

   - Input: An undirected graph G and an integer k.
   - Output: True if G has a proper coloring with k colors, and False otherwise.[3]

   Using this black box as a subroutine, describe an algorithm that solves the following ***coloring problem*** *in polynomial time*:

   - Input: An undirected graph G.
   - Output: A valid coloring of G using the minimum possible number of colors.

   *[Hint: You can use the magic box more than once. The input to the magic box is a graph and **only** a graph, meaning **only** vertices and edges.]*

4. Suppose you are given a magic black box that somehow answers the following decision problem in *polynomial time*:

   - Input: A boolean circuit K with n inputs and one output.
   - Output: True if there are input values $x_1, x_2, \dots, x_n \in \{\text{True}, \text{False}\}$ that make K output True, and False otherwise.

   Using this black box as a subroutine, describe an algorithm that solves the following related search problem *in polynomial time*:

   - Input: A boolean circuit K with n inputs and one output.
   - Output: Input values $x_1, x_2, \dots, x_n \in \{\text{True}, \text{False}\}$ that make K output True, or None if there are no such inputs.

   [Hint: You can use the magic box more than once.]

[3] Again, it is not hard to solve this problem in polynomial time via dynamic programming when the input graph G is a *tree*, but this magic box works for *every* input graph.

Proving that a problem X is NP-hard requires several steps:

- Choose a problem Y that you already know is NP-hard (because we told you so in class).
- Describe an algorithm to solve Y, using an algorithm for X as a subroutine. Typically this algorithm has the following form: Given an instance of Y, transform it into an instance of X, and then call the magic black-box algorithm for X.
- ***Prove*** that your algorithm is correct. This always requires two separate steps, which are usually of the following form:
  - ***Prove*** that your algorithm transforms "good" instances of Y into "good" instances of X.
  - ***Prove*** that your algorithm transforms "bad" instances of Y into "bad" instances of X. Equivalently: Prove that if your transformation produces a "good" instance of X, then it was given a "good" instance of Y.
- Argue that your algorithm for Y runs in polynomial time. (This is usually trivial.)

---

1. Recall the following kColor problem: Given an undirected graph G, can its vertices be colored with k colors, so that every edge touches vertices with two different colors?

   (a) Describe a direct polynomial-time reduction from 3Color to 4Color.

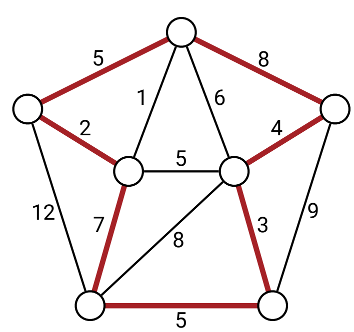
   (b) Prove that kColor problem is NP-hard for any $k \geq 3$.

2. A *Hamiltonian cycle* in a graph G is a cycle that goes through every vertex of G exactly once. Deciding whether an arbitrary graph contains a Hamiltonian cycle is NP-hard.

   A ***tonian cycle*** in a graph G is a cycle that goes through at least *half* of the vertices of G. Prove that deciding whether a graph contains a tonian cycle is NP-hard.

To think about later:

3. Let G be an undirected graph with weighted edges. A Hamiltonian cycle in G is ***heavy*** if the total weight of edges in the cycle is at least half of the total weight of all edges in G. Prove that deciding whether a graph contains a heavy Hamiltonian cycle is NP-hard.



A heavy Hamiltonian cycle. The cycle has total weight 34; the graph has total weight 67.

Prove that each of the following problems is NP-hard.

1. Given an undirected graph G, does G contain a simple path that visits all but 374 vertices?

2. Given an undirected graph G, does G have a spanning tree in which every vertex has degree at most 374?

3. Given an undirected graph G, does G have a spanning tree with at most 374 leaves?

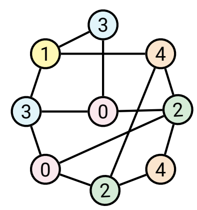
[Hint: Consider the corresponding problems with 1 or 2 in place of 374.]

Recall that a proper k-coloring of a graph G is a function that assigns each vertex of G a "color" from the set $\{0, 1, 2, \ldots, k-1\}$ (or less formally, from any set of size k), such that for any edge uv, vertices u and v are assigned different "colors". The *chromatic number* of G is the smallest integer k such that G has a proper k-coloring.

1. A proper k-coloring of a graph G is ***balanced*** if each color is assigned to exactly the same number of vertices. Prove that it is NP-hard to decide whether a given graph G has a *balanced* 3-coloring. *[Hint: Reduce from the standard 3Color problem.]*

2. Prove that the following problem is NP-hard: Given an undirected graph G, find *any* integer $k > 374$ such that G has a proper coloring with k colors but G does not have a proper coloring with $k - 374$ colors. For example, if the chromatic number of G is 10000, then any integer between 10000 and 10373 is a correct answer.

3. A 5-coloring is ***careful*** if the colors assigned to adjacent vertices are not only distinct, but differ by more than 1 (mod 5). Prove that deciding whether a given graph has a careful 5-coloring is NP-hard. *[Hint: Reduce from the standard 5Color problem.]*

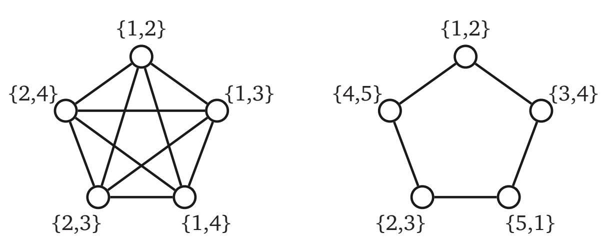


A careful 5-coloring.

4. A ***bicoloring*** of an undirected graph assigns each vertex a set of *two* colors. There are two types of bicoloring: In a *weak* bicoloring, the endpoints of each edge must use *different* sets of colors; however, these two sets may share one color. In a *strong* bicoloring, the endpoints of each edge must use *distinct* sets of colors; that is, they must use four colors altogether. Every strong bicoloring is also a weak bicoloring.

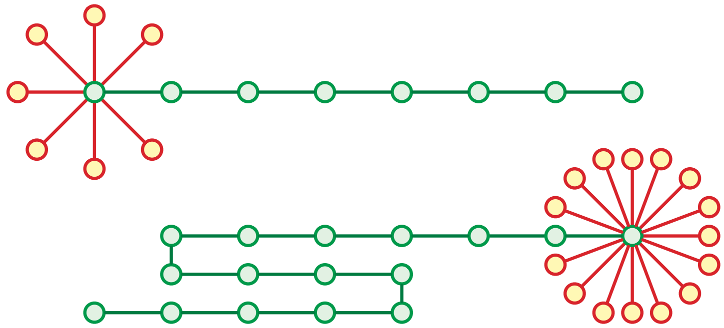
   (a) Prove that it is NP-hard to determine whether a given graph has a weak bicoloring with three colors. *[Hint: Reduce from the standard 3Color problem.]*

   (b) Prove that it is NP-hard to determine whether a given graph has a strong bicoloring with ***five*** colors. *[Hint: Reduce from the standard 3Color (sic) problem!]*



Left: A weak bicoloring of a 5-clique with four colors.
Right A strong bicoloring of a 5-cycle with five colors.

1. LONGEST DANDELION: A *dandelion of length* ℓ consists of a path of length ℓ, with exactly ℓ new edges attached to one end. Prove that it is NP-hard to find the longest dandelion subgraph of a given undirected graph.



Two dandelions, one of length 7 and the other of length 15.

2. HIGH-DEGREE INDEPENDENT SET: Suppose we are given a graph G and an integer k. Prove that it is NP-hard to decide whether G contains an independent set of k vertices, each of which has degree at least k.

   *[Hint: Reduce from the **decision** version of the INDEPENDENTSET problem: Given a graph G and an integer k, does G contain an independent set of size k?]*

3. HALF-CLIQUE: Suppose we are given a graph G with $2n$ vertices, for some integer n. Prove that it is NP-hard to decide whether G contains a complete subgraph with n vertices?

   *[Hint: Reduce from the **decision** version of the CLIQUE problem: Given a graph G and an integer k, does G contain a clique of size k?]*

Rice's Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\textsc{Accept}(Y) \in \mathcal{L}$.*
- *There is a Turing machine N such that $\textsc{Accept}(N) \notin \mathcal{L}$.*

The language $\textsc{AcceptIn}(\mathcal{L}) := \left\{ \langle M \rangle \;\middle|\; \textsc{Accept}(M) \in \mathcal{L} \right\}$ *is undecidable.*

Prove that the following languages are undecidable *using Rice's Theorem*:

1. $\textsc{AcceptRegular} := \left\{ \langle M \rangle \;\middle|\; \textsc{Accept}(M) \text{ is regular} \right\}$
2. $\textsc{AcceptIllini} := \left\{ \langle M \rangle \;\middle|\; M \text{ accepts the string } \texttt{ILLINI} \right\}$
3. $\textsc{AcceptPalindrome} := \left\{ \langle M \rangle \;\middle|\; M \text{ accepts at least one palindrome} \right\}$
4. $\textsc{AcceptThree} := \left\{ \langle M \rangle \;\middle|\; M \text{ accepts exactly three strings} \right\}$
5. $\textsc{AcceptUndecidable} := \left\{ \langle M \rangle \;\middle|\; \textsc{Accept}(M) \text{ is undecidable} \right\}$

To think about later. Which of the following are undecidable? How would you prove that?

1. $\textsc{Accept}\{\{\varepsilon\}\} := \left\{ \langle M \rangle \;\middle|\; M \text{ accepts only the string } \varepsilon\text{; that is, } \textsc{Accept}(M) = \{\varepsilon\} \right\}$
2. $\textsc{Accept}\{\varnothing\} := \left\{ \langle M \rangle \;\middle|\; M \text{ does not accept any strings; that is, } \textsc{Accept}(M) = \varnothing \right\}$
3. $\textsc{Accept=Reject} := \left\{ \langle M \rangle \;\middle|\; \textsc{Accept}(M) = \textsc{Reject}(M) \right\}$
4. $\textsc{Accept}{\neq}\textsc{Reject} := \left\{ \langle M \rangle \;\middle|\; \textsc{Accept}(M) \neq \textsc{Reject}(M) \right\}$
5. $\textsc{Accept}{\cup}\textsc{Reject} := \left\{ \langle M \rangle \;\middle|\; \textsc{Accept}(M) \cup \textsc{Reject}(M) = \Sigma^* \right\}$

Proving that a language L is undecidable by reduction requires several steps. (These are the essentially the same steps you already use to prove that a problem is NP-hard.)

- Choose a language L' that you already know is undecidable (because we told you so in class). The simplest choice is usually the standard halting language

$$\textsc{Halt} := \{\langle M, w\rangle \mid M \text{ halts on } w\}$$

- Describe an algorithm that decides L', using an algorithm that decides L as a black box. Typically your reduction will have the following form:

  Given an arbitrary string x, construct a special string y,
  such that $y \in L$ if and only if $x \in L'$.

  In particular, if $L = \textsc{Halt}$, your reduction will have the following form:

  Given the encoding $\langle M, w\rangle$ of a Turing machine M and a string w,
  construct a special string y, such that
  $y \in L$ if and only if M halts on input w.

- Prove that your algorithm is correct. This proof almost always requires two separate steps:
  - Prove that if $x \in L'$ then $y \in L$.
  - Prove that if $x \notin L'$ then $y \notin L$.

Very important: Name every object in your proof, and *always* refer to objects by their names. Never *ever* refer to "the Turing machine" or "the algorithm" or "the code" or "the input string" or (gods forbid) "it" or "this", even in casual conversation, even if you're "just" explaining your intuition, even when you're "just" *thinking* about the reduction to yourself.

Prove that the following languages are undecidable.

1. $\textsc{AcceptIllini} := \{\langle M\rangle \mid M \text{ accepts the string } \texttt{ILLINI}\}$
2. $\textsc{AcceptThree} := \{\langle M\rangle \mid M \text{ accepts exactly three strings}\}$
3. $\textsc{AcceptPalindrome} := \{\langle M\rangle \mid M \text{ accepts at least one palindrome}\}$
4. $\textsc{AcceptOnlyPalindromes} := \{\langle M\rangle \mid \text{Every string accepted by } M \text{ is a palindrome}\}$

A solution for problem 1 appears on the next page; don't look at it until you've thought a bit about the problem first.

Solution (for problem 1): For the sake of argument, suppose there is an algorithm DecideAcceptIllini that correctly decides the language AcceptIllini. Then we can solve the halting problem as follows:

```
DecideHalt(⟨M, w⟩):
  Encode the following Turing machine M′:
      M′(x):
        ⟨⟨ignore the input string x⟩⟩
        run M on input w
        ⟨⟨ignore the output of M⟩⟩
        return True
  if DecideAcceptIllini(⟨M′⟩)
      return True
  else
      return False
```

We prove this reduction correct as follows:

$\Longrightarrow$ Suppose M halts on input w.

Then M' accepts *every* input string x.

In particular, M' accepts the string `ILLINI`.

So DecideAcceptIllini accepts the encoding $\langle M' \rangle$.

So DecideHalt correctly accepts the encoding $\langle M, w \rangle$.

$\Longleftarrow$ Suppose M does not halt on input w.

Then M' diverges on *every* input string x.

In particular, M' does not accept the string `ILLINI`.

So DecideAcceptIllini rejects the encoding $\langle M' \rangle$.

So DecideHalt correctly rejects the encoding $\langle M, w \rangle$.

In both cases, DecideHalt is correct. But that's impossible, because Halt is undecidable. We conclude that the algorithm DecideAcceptIllini does not exist. ■

As usual for undecidablility proofs, this proof invokes *four* distinct Turing machines:

- The hypothetical algorithm DecideAcceptIllini.
- The new algorithm DecideHalt that we construct in the solution.
- The arbitrary machine M whose encoding is part of the input to DecideHalt.
- The special machine M' whose encoding DecideHalt constructs (from the encoding of M and w) and then passes to DecideAcceptIllini.