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*** 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 $\#(\texttt{0}, \texttt{00001010101010010100}) = 12$.

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 **$x \bullet \varepsilon = x$** 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 **$w \bullet \varepsilon = w$**.* □

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 $\emptyset$ (representing the empty set)
- Any string (representing the set containing only that string)
- $R + S$ for some regular expressions R and S (representing alternation / union)
- $R \cdot S$ or RS for some regular expressions R and S (representing concatenation)
- 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 `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 work on 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 $\Sigma = \{0, 1\}$. 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. $\left\{0^{2^n} \;\middle|\; n \geq 0\right\}$

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. $\left\{0^{n^2} \;\middle|\; n \geq 0\right\}$

*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 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:

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

   Prove that the language 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 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.

   $$\begin{aligned} Q' &= Q \\ s' &= s \\ A' &= A \\ \delta'(q, 0) &= \{\ \delta(q, 0)\ \} \\ \delta'(q, 1) &= \{\qquad\qquad\qquad\qquad\qquad\qquad\} \\ \delta'(q, \varepsilon) &= \{\qquad\qquad\qquad\qquad\qquad\qquad\} \end{aligned}$$

   ■

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.

$$\begin{aligned}
Q' &= Q \\
s' &= s \\
A' &= A \\
\delta'(q, 0) &= \{\ \delta(q, 0)\ \} \\
\delta'(q, 1) &= \{\qquad\qquad\qquad\qquad\qquad\qquad\} \\
\delta'(q, \varepsilon) &= \{\qquad\qquad\qquad\qquad\qquad\qquad\}
\end{aligned}$$

■

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

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

Prove that the language $\textsc{InsertOne1}(L)$ 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 $\textsc{InsertOne1}(L)$ 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 $(q, \textsc{False})$ means (the simulation of) M is in state q and M' has not yet skipped over a 1. State $(q, \textsc{True})$ means (the simulation of) M is in state q and M' has already skipped over a 1.

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

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

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

■

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

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

Prove that the language $\textsc{DeleteOne1}(L)$ 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 $\textsc{DeleteOne1}(L)$ 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 $(q, \textsc{False})$ means (the simulation of) M is in state q and M' has not yet reinserted a 1. State $(q, \textsc{True})$ means (the simulation of) M is in state q and M' has already reinserted a 1.

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

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

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

■

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, $evens(\texttt{THE⋄SNAIL}) = \texttt{H⋄NI}$ and $evens(\texttt{GROB⋄GOB⋄GLOB⋄GROD}) = \texttt{RBGBGO⋄RD}$.

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.

$$\begin{aligned} S &\to A \mid B & \{0^m 1^n \mid m \neq n\} \\ A &\to 0A \mid 0C & \{0^m 1^n \mid m > n\} \\ B &\to B1 \mid C1 & \{0^m 1^n \mid m < n\} \\ C &\to \varepsilon \mid 0C1 & \{0^m 1^n \mid m = n\} \end{aligned}$$

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

   [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. $\left\{w \in \{0,1\}^* \;\middle|\; \#(0,w) = 2 \cdot \#(1,w)\right\}$ — 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 $\Sigma = \{0, 1\}$. 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:

$$\textsc{Superstrings}(\{10010\}) = \{10010,\ 01010010,\ 1001011,\ 10010010010,\ \cdots\}$$

[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:

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

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:

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

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:

$$\mathit{flipOdds}(0000111101010100) = 1010010111111110$$

5. $\textsc{UnflipOdd1s}(L) := \{w \in \Sigma^* \mid \mathit{flipOdd1s}(w) \in L\}$, where the function *flipOdd1* inverts every other 1 bit of its input string, starting with the first 1. For example:

$$\mathit{flipOdd1s}(0000111100101010) = 0000010100001000$$

6. $\textsc{FlipOdd1s}(L) := \{\mathit{flipOdd1s}(w) \mid w \in L\}$, where the function *flipOdd1s* is defined in the previous problem.

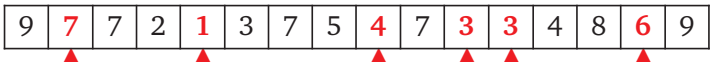
Here are several problems that are easy to solve in $O(n)$ time, essentially by brute force. Your task is to design algorithms for these problems that are significantly faster.

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:

   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 mod 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

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

   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

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

   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

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

   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

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

   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

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

   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:

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

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 $1+3(-2)(-5)+1-6+7=33$.

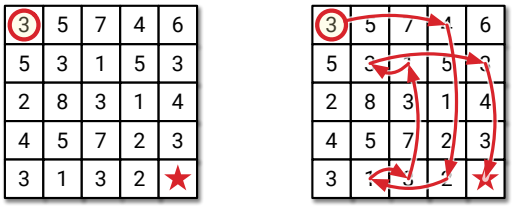
For each of the problems below, transform the input into a graph and apply a standard graph algorithm that you've seen in class. Whenever you use a standard graph algorithm, you ***must*** provide the following information. (I recommend actually using a bulleted list.)

- 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 $1 \leq s \leq n$ 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.]*

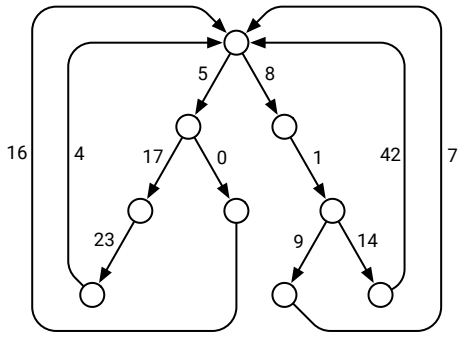
2. You just discovered your best friend from elementary school on Face𝕏 (formerly known as Twitbook). You both want to meet as soon as possible, but you live in two different cities that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop 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 $\boldsymbol{O(V^2)}$ ***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 $\boldsymbol{O(1)}$ ***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 $\boldsymbol{O(VE)}$ ***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 $4, 5^\star, 6, 7, 9^\star$.

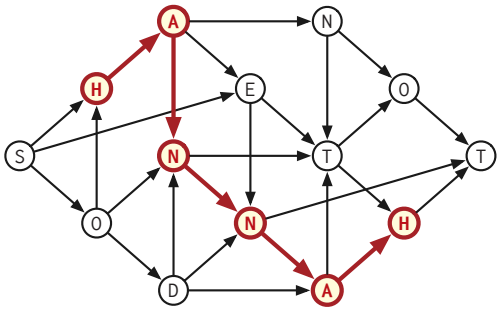
   (a) Describe an algorithm for this problem using dynamic programming.

   (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 $c : V \to \{1, 2, \ldots, k\}$, 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 $x_1, x_2, \ldots, x_n \in \{\text{TRUE}, \text{FALSE}\}$ 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 $x_1, x_2, \ldots, x_n \in \{\text{TRUE}, \text{FALSE}\}$ 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.

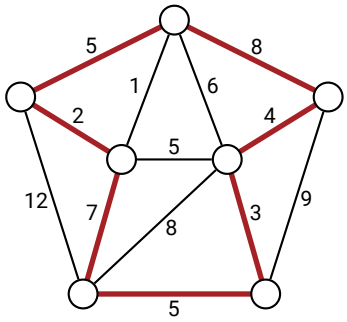
   (b) Prove that kColor problem is NP-hard for any $k \geq 3$.

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?

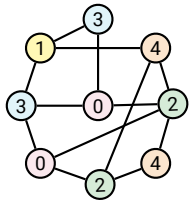
[Hint: Consider the corresponding problems with 1 or 2 in place of 374.]

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.]*

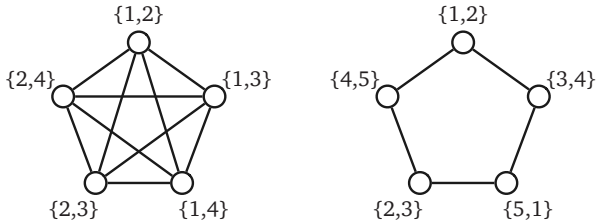


A careful 5-coloring.

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.

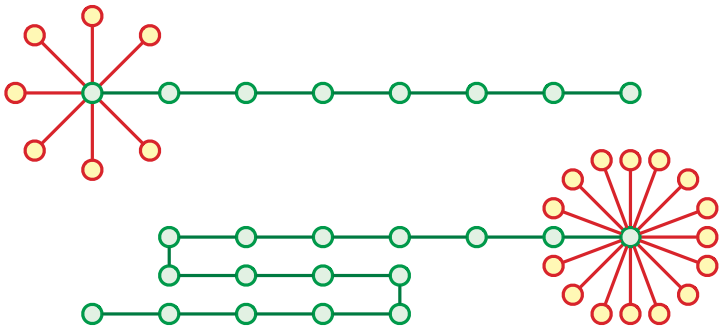
   (a) Prove that it is NP-hard to determine whether a given graph has a weak bicoloring with three colors. *[Hint: Reduce from the standard 3Color problem.]*

   (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 $\textsc{AcceptIn}(\mathcal{L}) := \{\langle M \rangle \mid \textsc{Accept}(M) \in \mathcal{L}\}$ is undecidable.

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

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

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

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

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.

CS/ECE 374 A ✧ Fall 2023

Homework 1

Due Tuesday, August 29, 2023 at 9pm Central Time

- **Submit your written solutions electronically to Gradescope as PDF files.** Submit a separate PDF file for each numbered problem. If you plan to typeset your solutions, you are welcome to use the LaTeX solution template on the course web site. If you must submit scanned handwritten solutions, please use a black pen on blank white paper and a high-quality scanner app (or an actual scanner).
- Groups of up to three people can submit joint solutions on Gradescope. ***Exactly*** one student in each group should upload the solution and indicate their other group members.
- **You may use any source at your disposal**—paper, electronic,[1] or human—but you ***must*** cite ***every*** source that you use,[2] you ***must*** write everything yourself in your own words, and you are responsible for any errors in the sources you use.[3] See the academic integrity policies on the course web site for more details.
- Written homework is normally due every Tuesday at 9pm. In addition, guided problem sets on PrairieLearn are normally due every **Monday** at 9pm; each student must do these individually. In particular, Guided Problem Set 1 is due Monday, August 28!
- Both guided problem sets and homework may be submitted up to 24 hours late for 50% partial credit, or for full credit with an approved extension. See the grading policies on the course web site for more details.
- Each homework will include at least one fully solved problem, similar to that week's assigned problems, together with the rubric we would use to grade this problem if it appeared in an actual homework or exam. These model solutions show our recommendations for structure, presentation, and level of detail in your homework solutions. (Obviously, the actual *content* of your solutions won't match the model solutions, because your problems are different!) Homeworks may also include additional practice problems.
- Standard grading rubrics for many problem types can be found on the course web page. For example, the problems in this week's homework will be graded using the standard induction rubric. (Weak induction makes the baby Jesus cry.)

See the course web site for more information.

If you have any questions about these policies,
please don't hesitate to ask in class, in office hours, or on Piazza.

---

[1] Yes, including ChatGPT.
[2] Yes, including ChatGPT.
[3] Yes, including ChatGPT.

1. Consider the following recursively defined function:

$$stutter(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ aa \bullet stutter(x) & \text{if } w = ax \end{cases}$$

For example, $stutter(\texttt{MISSISSIPPI}) = \texttt{MMIISSSSIISSSSIIPPPPII}$.

(a) Prove that $|stutter(w)| = 2|w|$ for every string w.

(b) Prove that $stutter(x \bullet y) = stutter(x) \bullet stutter(y)$ for all strings x and y.

(c) ***Practice only. Do not submit solutions.***

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

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

For example, $\texttt{MISSIPPIPPI}^R = \texttt{IPPIPPISSIM}$.

Prove that $stutter(w)^R = stutter(w^R)$ for every string w.

You may freely use any result proved in lecture, in lab, or in the lecture notes. Otherwise your proofs must be formal and self-contained. In particular, your proofs *must* invoke the formal recursive definitions of string length and concatenation (and for part (c), reversal).

2. For each positive integer n, we define two strings p_n and v_n, respectively called the nth *Piṅgala string* and the nth *Virahāṇka string*. Piṅgala strings are defined by the following recurrence:

$$p_n = \begin{cases} \texttt{1} & \text{if } n = 1 \\ \texttt{0} & \text{if } n = 2 \\ p_{n-2} \bullet p_{n-1} & \text{otherwise} \end{cases}$$

For example:

$$p_7 = \overbrace{\texttt{10}}^{p_5}\overbrace{\texttt{0}}^{p_4}\overbrace{\texttt{10}}^{p_5}\overbrace{\texttt{0}\,\texttt{10}}^{p_6}\texttt{10}.$$

Virahāṇka strings are defined more indirectly as

$$v_n = \begin{cases} \texttt{1} & \text{if } n = 1 \\ \text{grow}(v_{n-1}) & \text{otherwise} \end{cases}$$

where the string function grow is defined as follows:

$$\text{grow}(w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \texttt{0} \cdot \text{grow}(x) & \text{if } w = \texttt{1}x \\ \texttt{10} \bullet \text{grow}(x) & \text{if } w = \texttt{0}x \end{cases}$$

For example:

$$\text{grow}(\texttt{01010010}) = \texttt{10} \bullet \texttt{0} \bullet \texttt{10} \bullet \texttt{0} \bullet \texttt{10} \bullet \texttt{10} \bullet \texttt{0} \bullet \texttt{10} = \texttt{1001001010010}$$

Finally, recall that the Fibonacci numbers are defined recursively as follows:

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

(a) Prove that $|p_n| = F_n$ for all $n \geq 1$.

(b) Prove that $\text{grow}(w \bullet z) = \text{grow}(w) \bullet \text{grow}(z)$ for all strings w and z.

(c) Prove that $p_n = v_n$ for all $n \geq 1$. *[Hint: Careful!]*

(d) ***Practice only. Do not submit solutions.***
Prove that $|v_n| = F_n$ for all $n \geq 1$.

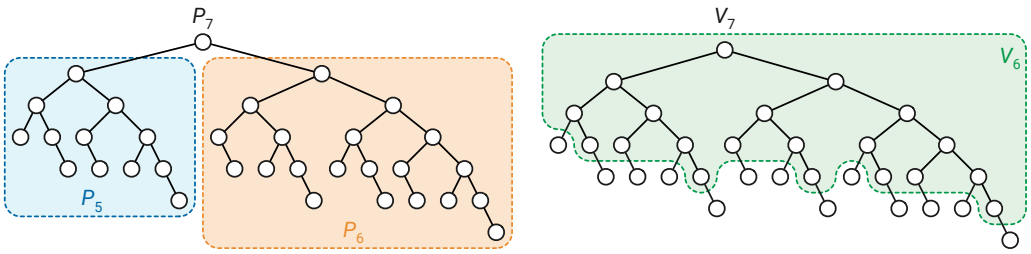
As in problem 1, you may freely use any result that proved in lecture, in lab, or in the lecture notes. Otherwise your proofs must be formal and self-contained. In particular, your proofs *must* invoke the formal recursive definitions of the strings p_n and v_n, the grow function, and the Fibonacci numbers F_n.

*3. ***Practice only. Do not submit solutions.***

For each non-negative integer n, we recursively define two binary trees P_n and V_n, called the nth *Piṅgala tree* and the nth *Virahāṇka tree*, respectively.

- P_0 and V_0 are empty trees, with no nodes.
- P_1 and V_1 each consist of a single node.
- For any integer $n \geq 2$, the tree P_n consists of a root with two subtrees; the left subtree is a copy of P_{n-1}, and the right subtree is a copy of P_{n-2}.
- For any integer $n \geq 2$, the tree L_n is obtained from L_{n-1} by attaching a new right child to every leaf and attaching a new left child to every node that has only a right child.

The following figure shows the recursive construction of these two trees when $n = 7$.



Recall that the Fibonacci numbers are defined recursively as follows:

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

(a) Prove that the tree P_n has exactly F_n leaves.

(b) Prove that the tree V_n has exactly F_n leaves.
[Hint: You need to prove a stronger result.]

(c) Prove that the trees P_n and V_n are identical, for all $n \geq 0$.
[Hint: The hardest part of this proof is developing the right language/notation.]

As in problem 1, you may freely use any result that proved in lecture, in lab, or in the lecture notes. Otherwise your proofs must be formal and self-contained. In particular, your proofs *must* invoke the formal recursive definitions of the trees P_n and V_n and the Fibonacci numbers F_n.

Solved Problems

3. For any string $w \in \{0,1\}^*$, let $swap(w)$ denote the string obtained from w by swapping the first and second symbols, the third and fourth symbols, and so on. For example:

$$swap(10\,11\,00\,01\,10\,1) = 01\,11\,00\,10\,01\,1.$$

The *swap* function can be formally defined as follows:

$$swap(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ w & \text{if } w = 0 \text{ or } w = 1 \\ ba \bullet swap(x) & \text{if } w = abx \text{ for some } a, b \in \{0,1\} \text{ and } x \in \{0,1\}^* \end{cases}$$

(a) Prove that $|swap(w)| = |w|$ for every string w.

Solution: Let w be an arbitrary string.
Assume $|swap(x)| = |x|$ for every string x that is shorter than w.
There are three cases to consider (mirroring the definition of *swap*):

- If $w = \varepsilon$, then

$$\begin{aligned} |swap(w)| &= |swap(\varepsilon)| && \text{because } w = \varepsilon \\ &= |\varepsilon| && \text{by definition of } swap \\ &= |w| && \text{because } w = \varepsilon \end{aligned}$$

- If $w = 0$ or $w = 1$, then

$$|swap(w)| = |w| \qquad \text{by definition of } swap$$

- Finally, if $w = abx$ for some $a, b \in \{0,1\}$ and $x \in \{0,1\}^*$, then

$$\begin{aligned} |swap(w)| &= |swap(abx)| && \text{because } w = abx \\ &= |ba \bullet swap(x)| && \text{by definition of } swap \\ &= |ba| + |swap(x)| && \text{because } |y \bullet z| = |y| + |z| \\ &= |ba| + |x| && \text{by the induction hypothesis} \\ &= 2 + |x| && \text{by definition of } |\cdot| \\ &= |ab| + |x| && \text{by definition of } |\cdot| \\ &= |ab \bullet x| && \text{because } |y \bullet z| = |y| + |z| \\ &= |abx| && \text{by definition of } \bullet \\ &= |w| && \text{because } w = abx \end{aligned}$$

In all cases, we conclude that $|swap(w)| = |w|$. ■

Rubric: 5 points: Standard induction rubric (scaled). This is more detail than necessary for full credit.

(b) Prove that $swap(swap(w)) = w$ for every string w.

Solution: Let w be an arbitrary string.
Assume $swap(swap(x)) = x$ for every string x that is shorter than w.
There are three cases to consider (mirroring the definition of *swap*):

- If $w = \varepsilon$, then

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

- If $w = \mathtt{0}$ or $w = \mathtt{1}$, then

$$\begin{aligned} swap(swap(w)) &= swap(w) && \text{by definition of } swap \\ &= w && \text{by definition of } swap \end{aligned}$$

- Finally, if $w = abx$ for some $a, b \in \{\mathtt{0}, \mathtt{1}\}$ and $x \in \{\mathtt{0}, \mathtt{1}\}^*$, then

$$\begin{aligned} swap(swap(w)) &= swap(swap(abx)) && \text{because } w = abx \\ &= swap(ba \bullet swap(x)) && \text{by definition of } swap \\ &= swap(ba \bullet z) && \text{where } z = swap(x) \\ &= swap(baz) && \text{by definition of } \bullet \\ &= ab \bullet swap(z) && \text{by definition of } swap \\ &= ab \bullet swap(swap(x)) && \text{because } z = swap(x) \\ &= ab \bullet x && \text{by the induction hypothesis} \\ &= abx && \text{by definition of } \bullet \\ &= w && \text{because } w = abx \end{aligned}$$

In all cases, we conclude that $swap(swap(w)) = w$. ■

Rubric: 5 points: Standard induction rubric (scaled). This is more detail than necessary for full credit.

4. The ***reversal*** w^R of a string w is defined recursively as follows:

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = a \cdot x \end{cases}$$

A ***palindrome*** is any string that is equal to its reversal, like AMANAPLANACANALPANAMA, RACECAR, POOP, I, and the empty string.

(a) Give a recursive definition of a palindrome over the alphabet Σ.

Solution: A string $w \in \Sigma^*$ is a palindrome if and only if either

- $w = \varepsilon$, or
- $w = a$ for some symbol $a \in \Sigma$, or
- $w = axa$ for some symbol $a \in \Sigma$ and some *palindrome* $x \in \Sigma^*$.

■

Rubric: 2 points = ½ for each base case + 1 for the recursive case. No credit for the rest of the problem unless this part is correct.

(b) Prove $w = w^R$ for every palindrome w (according to your recursive definition). You may assume the following facts about all strings x, y, and z:

- Reversal reversal: $(x^R)^R = x$
- Concatenation reversal: $(x \bullet y)^R = y^R \bullet x^R$
- Right cancellation: If $x \bullet z = y \bullet z$, then $x = y$.

Solution: Let w be an arbitrary palindrome.

Assume that $x = x^R$ for every palindrome x such that $|x| < |w|$.

There are three cases to consider (mirroring the definition of "palindrome"):

- If $w = \varepsilon$, then $w^R = \varepsilon$ by definition, so $w = w^R$.
- If $w = a$ for some symbol $a \in \Sigma$, then $w^R = a$ by definition, so $w = w^R$.
- Finally, if $w = axa$ for some symbol $a \in \Sigma$ and some palindrome $x \in P$, then

$$\begin{aligned} w^R &= (a \cdot x \bullet a)^R && \text{because w = axa} \\ &= (x \bullet a)^R \bullet a && \text{by definition of reversal} \\ &= a^R \bullet x^R \bullet a && \text{by concatenation reversal} \\ &= a \bullet x^R \bullet a && \text{by definition of reversal} \\ &= a \bullet x \bullet a && \text{by the inductive hypothesis} \\ &= w && \text{because w = axa} \end{aligned}$$

In all three cases, we conclude that $w = w^R$. ■

Rubric: 4 points: standard induction rubric (scaled)

(c) Prove that every string w such that $w = w^R$ is a palindrome (according to your recursive definition).

Again, you may assume the following facts about all strings x, y, and z:

- Reversal reversal: $(x^R)^R = x$
- Concatenation reversal: $(x \bullet y)^R = y^R \bullet x^R$
- Right cancellation: If $x \bullet z = y \bullet z$, then $x = y$.

Solution: Let w be an arbitrary string such that $w = w^R$.

Assume that every string x such that $|x| < |w|$ and $x = x^R$ is a palindrome.

There are three cases to consider (mirroring the definition of "palindrome"):

- If $w = \varepsilon$, then w is a palindrome by definition.
- If $w = a$ for some symbol $a \in \Sigma$, then w is a palindrome by definition.
- Otherwise, we have $w = ax$ for some symbol a and some *non-empty* string x.
  The definition of reversal implies that $w^R = (ax)^R = x^R a$.
  Because x is non-empty, its reversal x^R is also non-empty.
  Thus, $x^R = by$ for some symbol b and some string y.
  It follows that $w^R = bya$, and therefore $w = (w^R)^R = (bya)^R = ay^R b$.

  ⟨⟨*At this point, we need to prove that $a = b$ and that y is a palindrome.*⟩⟩

  Our assumption that $w = w^R$ implies that $bya = ay^R b$.
  The recursive definition of string equality immediately implies $a = b$.

  Because $a = b$, we have $w = ay^R a$ and $w^R = aya$.
  The recursive definition of string equality implies $y^R a = ya$.
  Right cancellation implies $y^R = y$.
  The inductive hypothesis now implies that y is a palindrome.

  We conclude that w is a palindrome by definition.

In all three cases, we conclude that w is a palindrome. ■

Rubric: 4 points: standard induction rubric (scaled).

5. Let $L \subseteq \{0,1\}^*$ be the language defined recursively as follows:

   - The empty string ε is in L.
   - For any string $x \in L$, the strings $0101x$ and $1010x$ are also in L.
   - For all strings x and y such that $xy \in L$, the strings $x00y$ and $x11y$ are also in L. (In other words, inserting two consecutive 0s or two consecutive 1s anywhere in a string in L yields another string in L.)
   - These are the only strings in L.

   Let EE denote the set of all strings $w \in \{0,1\}^*$ such that $\#(0,w)$ and $\#(1,w)$ are both even.

   In the following proofs, you may freely use any result proved in lecture, in lab, in the lecture notes, or earlier in your homework. Otherwise your proofs must be formal and self-contained; in particular, they must invoke the formal recursive definitions of $\#$ and L.

   (a) Prove that $L \subseteq EE$.

   Solution: Let w be an arbitrary string in L. We need to prove that $\#(0,w)$ and $\#(1,w)$ are both even. Here I will prove only that $\#(0,w)$ is even; the proof that $\#(1,w)$ is even is symmetric.

   Assume for every string $x \in L$ such that $|x| < |w|$ that $\#(0,x)$ is even.

   There are several cases to consider, mirroring the definition of L.

   - Suppose $w = \varepsilon$. Then $\#(0,w) = 0$, and 0 is even.
   - Suppose $w = 0101x$ or $w = 1010x$ for some string $x \in L$. The definition of $\#$ (applied four times) implies $\#(0,w) = \#(0,x)+2$. The inductive hypothesis implies $\#(0,x)$ is even. We conclude that $\#(0,w)$ is even.
   - Suppose $w = x00y$ for some strings x and y such that $xy \in L$. Then

   $$\begin{aligned}\#(0,w) &= \#(0,x00y)\\ &= \#(0,x)+\#(0,00)+\#(0,y)\\ &= \#(0,x)+\#(0,y)+\#(0,00)\\ &= \#(0,xy)+2\end{aligned}$$

   The induction hypothesis implies $\#(0,xy)$ is even. We conclude that $\#(0,w) = \#(0,xy)+2$ is also even.
   - Finally, suppose $w = x11y$ for some strings x and y such that $xy \in L$. Then

   $$\begin{aligned}\#(0,w) &= \#(0,x11y)\\ &= \#(0,x)+\#(0,11)+\#(0,y)\\ &= \#(0,x)+\#(0,y)\\ &= \#(0,xy)\end{aligned}$$

   The induction hypothesis implies $\#(0,w) = \#(0,xy)$ is even.

In all cases, we have shown that $\#(0, w)$ is even. Symmetric arguments imply that $\#(1, w)$ is even. We conclude that $w \in EE$. ■

Rubric: 5 points: standard induction rubric (scaled). Yes, this is enough detail for $\#(1, w)$. If explicit proofs are given for both $\#(0, w)$ and $\#(1, w)$, grade them independently, each for 2½ points.

(b) Prove that $EE \subseteq L$.

Solution: Let w be an arbitrary string in EE. We need to prove that $w \in L$.

Assume that for every string $x \in EE$ such that $|x| < |w|$, we have $x \in L$.

There are four (overlapping) cases to consider, depending on the first four symbols in w.

- Suppose $|w| < 4$. Then w must be one of the strings ε, 00, or 11; brute force inspection implies that every other string of length at most 3 (0, 1, 01, 10, 000, 001, 010, 011, 100, 101, 110, 111) has an odd number of 0s or an odd number of 1s (or both). All three strings ε, 00, and 11 are in L. In all other cases, we can assume that $|w| \geq 4$, so the "first four symbols of w" are well-defined.
- Suppose the first four symbols of w are 0000 or 0001 or 0010 or 0011 or 0100 or 1000 or 1001 or 1100. Then $w = x00y$ for some (possibly empty) strings x and y. Arguments in part (a) imply that $\#(0, xy) = \#(0, w) - 2$ and $\#(1, xy) = \#(1, w)$ are both even. Thus $xy \in EE$ by definition of EE. So the induction hypothesis implies $xy \in L$. We conclude that $w = x00y \in L$ by definition of L.
- Suppose the first four symbols of w are 0011 or 0110 or 0111 or 1011 or 1100 or 1101 or 1110 or 1111.) After swapping 0s and 1s, the argument in the previous case implies that $w \in L$.
- Finally, suppose the first four symbols of w are 0101 or 1010; in other words, suppose $w = 0101x$ or $w = 1010x$ for some (possibly empty) string x. Then $\#(0, x) = \#(0, w) - 2$ and $\#(1, x) = \#(1, w) - 2$ are both even, so $x \in EE$ by definition. The induction hypothesis implies $x \in L$. We conclude that $w \in L$ by definition of L.

Each of the 16 possible choices for the first four symbols of w is considered in at least one of the last three cases.

In all cases, we conclude that $w \in L$. ■

Rubric: 5 points: standard induction rubric (scaled). This is not the only correct proof. This is not the only correct way to express this particular case analysis.

CS/ECE 374 A ✧ Fall 2023

Homework 2

Due Wednesday, September 6, 2023 at 9pm Central Time

(One day later than usual because of Labor Day)

- Starting with this homework, please start your solution to each *lettered subproblem* ((a), (b), (c), etc.) on a new page. Yes, even if the previous subproblem is only one line long. Please also remember to tell Gradescope which page(s) are relevant for which subproblems.

---

1. For each of the following languages over the alphabet $\{0,1\}^*$, describe an equivalent regular expression, and briefly explain why your regular expression is correct. There are infinitely many correct answers for each language.

   (a) All strings in 1^*01^* whose length is a multiple of 3.

   (b) All strings that begin with the prefix 001, end with the suffix 100, and contain an odd number of 1s.

   (c) All strings that contain both 0011 and 1100 as substrings.

   (d) All strings that contain the substring 01 an odd number of times.

   (e) $\{0^a1^b0^c \mid a \geq 0 \text{ and } b \geq 0 \text{ and } c \geq 0 \text{ and } a \equiv b + c \pmod 2\}$.

2. For each of the following languages over the alphabet $\Sigma = \{0,1\}$, describe a DFA that accepts the language, and briefly describe the purpose of each state. You can describe your DFA using a drawing, or using formal mathematical notation, or using a product construction; see the standard DFA rubric.

   (a) All strings in 1^*01^* whose length is a multiple of 3.

   (b) All strings that represent a multiple of 5 in base 3. For example, this language contains the string 10100, because $10100_3 = 90_{10}$ is a multiple of 5. (Yes, base 3 allows the digits 0, 1, and 2, but your input string will never contain a 2.)

   (c) All strings containing the substring 01010010. (The required substring is $p_6 = v_6$ from Homework 1.)

   (d) All strings whose ninth-to-last symbol is 0, or equivalently, the set

   $$\left\{x0z \mid x \in \Sigma^* \text{ and } z \in \Sigma^8\right\}.$$

   (e) All strings w such that $(\#(0,w) \bmod 3) + (\#(1,w) \bmod 7) = (|w| \bmod 4)$.

   [Hint: Don't try to draw the last two.]

3. ***Practice only. Do not submit solutions.***

This question asks about strings over the set of *pairs* of bits, which we will write vertically. Let Σ_2 denote the set of all bit-pairs:

$$\Sigma_2 = \left\{\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\right\}$$

We can interpret any string w of bit-pairs as a $2 \times |w|$ matrix of bits; each row of this matrix is the binary representation of some non-negative integer, possibly with leading 0s. Let $hi(w)$ and $lo(w)$ respectively denote the *numerical values* of the top and bottom row of this matrix. For example, $hi(\varepsilon) = lo(\varepsilon) = 0$, and if

$$w = \begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}0011\\0101\end{bmatrix}$$

then $hi(w) = 3$ and $lo(w) = 5$.

(a) Describe a DFA that accepts the language $L_{+1} = \{w \in \Sigma_2^* \mid hi(w) = lo(w) + 1\}$.
For example, $w = \begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1100\\1011\end{bmatrix} \in L_{+1}$, because $hi(w) = 12$ and $lo(w) = 11$.

(b) Describe a regular expression for L_{+1}.

(c) Describe a DFA that accepts the language $L_{\times 3} = \{w \in \Sigma_2^* \mid hi(w) = 3 \cdot lo(w)\}$.
For example, $w = \begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}1001\\0011\end{bmatrix} \in L_3$, because $hi(w) = 9$ and $lo(w) = 3$.

(d) Describe a regular expression for $L_{\times 3}$.

*(e) Describe a DFA that accepts the language $L_{\times 3/2} = \{w \in \Sigma_2^* \mid 2 \cdot hi(w) = 3 \cdot lo(w)\}$.
For example, $w = \begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}1001\\0110\end{bmatrix} \in L_{\times 3/2}$, because $hi(w) = 9$ and $lo(w) = 6$.

(Don't bother with the regular expression for this one.)

Solved problem

4. ***C comments*** are the set of strings over alphabet $\Sigma = \{*, /, \mathtt{A}, \diamond, \text{↲}\}$ that form a proper comment in the C program language and its descendants, like C++ and Java. Here ↲ represents the newline character, ◇ represents any other whitespace character (like the space and tab characters), and A represents any non-whitespace character other than * or /.[1] There are two types of C comments:

   - Line comments: Strings of the form //···↲
   - Block comments: Strings of the form /*···*/

   Following the C99 standard, we explicitly disallow ***nesting*** comments of the same type. A line comment starts with // and ends at the first ↲ after the opening //. A block comment starts with /* and ends at the the first */ completely after the opening /*; in particular, every block comment has at least two *s. For example, each of the following strings is a valid C comment:

   /***/ //◇//◇↲ /*///◇*◇↲**/ /*◇//◇↲◇*/

   On the other hand, *none* of the following strings is a valid C comment:

   /*/ //◇//◇↲◇↲ /*◇/*◇*/◇*/

   (Questions about C comments start on the next page.)

[1] The actual C commenting syntax is considerably more complex than described here, because of character and string literals.

- The opening /* or // of a comment must not be inside a string literal ("···") or a (multi-)character literal ('···').
- The opening double-quote of a string literal must not be inside a character literal ('"') or a comment.
- The closing double-quote of a string literal must not be escaped (\")
- The opening single-quote of a character literal must not be inside a string literal ("···'···") or a comment.
- The closing single-quote of a character literal must not be escaped (\')
- A backslash escapes the next symbol if and only if it is not itself escaped (\\) or inside a comment.

For example, the string "/*\\\"*/"/*"/*\"/*"*/ is a valid string literal (representing the 5-character string /*\"*/, which is itself a valid block comment!) followed immediately by a valid block comment. ***For this homework question, just pretend that the characters ', ", and \ don't exist.***

Commenting in C++ is even more complicated, thanks to the addition of *raw* string literals. Don't ask.

Some C and C++ compilers do support nested block comments, in violation of the language specification. A few other languages, like OCaml, explicitly allow nesting block comments.

(a) Describe a regular expression for the set of all C comments.

Solution:

$$//(/ + * + \mathtt{A} + \diamond)^*\text{↲} \quad + \quad /*\left(/ + \mathtt{A} + \diamond + \text{↲} + **^*(\mathtt{A} + \diamond + \text{↲})\right)^* \; *^**/$$

The first subexpression matches all line comments, and the second subexpression matches all block comments. Within a block comment, we can freely use any symbol other than $*$, but any run of $*$s must be followed by a character in $(\mathtt{A} + \diamond + \text{↲})$ or by the closing slash of the comment. ■

Rubric: Standard regular expression rubric. This is not the only correct solution.

(b) Describe a regular expression for the set of all strings composed entirely of blanks ($\diamond$), newlines (↲), and C comments.

Solution:

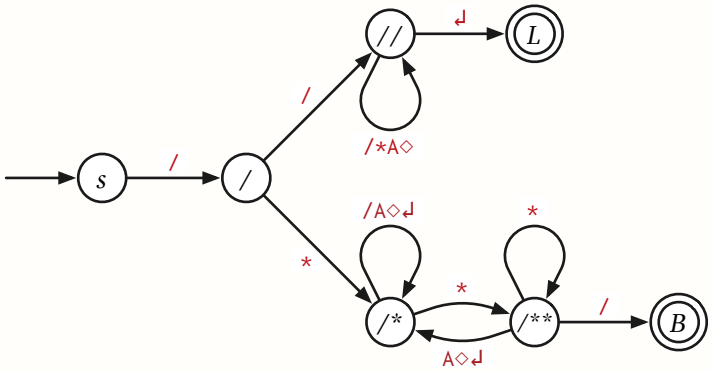
$$\left(\diamond + \text{↲} \; + \; //(/ + * + \mathtt{A} + \diamond)^*\text{↲} + /*(/ + \mathtt{A} + \diamond + \text{↲} + **^*(\mathtt{A} + \diamond + \text{↲}))^* \; **^*/\right)^*$$

This regular expression has the form $(\langle \text{whitespace} \rangle + \langle \text{comment} \rangle)^*$, where $\langle \text{whitespace} \rangle$ is the regular expression $\diamond + \text{↲}$ and $\langle \text{comment} \rangle$ is the regular expression from part (a). ■

Rubric: Standard regular expression rubric. This is not the only correct solution.

(c) Describe a DFA that accepts the set of all C comments.

Solution: The following eight-state DFA recognizes the language of C comments. All missing transitions lead to a hidden reject state.



The states are labeled mnemonically as follows:

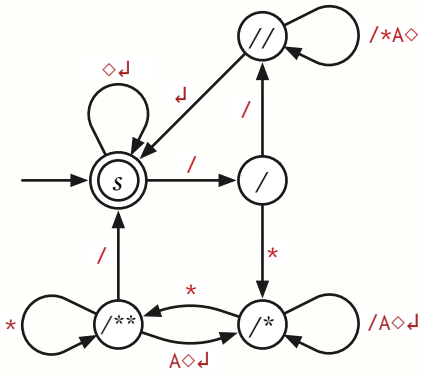
- s — We have not read anything.
- / — We just read the initial /.
- // — We are reading a line comment.
- L — We have just read a complete line comment.
- /* — We are reading a block comment, and we did not just read a * after the opening /*.
- /** — We are reading a block comment, and we just read a * after the opening /*.
- B — We have just read a complete block comment.

■

Rubric: Standard DFA design rubric. This is not the only correct solution, or even the simplest correct solution. (We don't need two distinct accepting states.)

(d) Describe a DFA that accepts the set of all strings composed entirely of blanks (◇), newlines (↲), and C comments.

Solution: By merging the accepting states of the previous DFA with the start state and adding white-space transitions at the start state, we obtain the following six-state DFA. Again, all missing transitions lead to a hidden reject state.



The states are labeled mnemonically as follows:

- *s* — We are between comments.
- / — We just read the initial / of a comment.
- // — We are reading a line comment.
- /* — We are reading a block comment, and we did not just read a * after the opening /*.
- /** — We are reading a block comment, and we just read a * after the opening /*.

■

Rubric: Standard DFA design rubric. This is not the only correct solution, but it is the simplest correct solution.

*5. Recall that the reversal w^R of a string w is defined recursively as follows:

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = a \cdot x \end{cases}$$

The reversal L^R of any *language* L is the set of reversals of all strings in L:

$$L^R := \left\{ w^R \;\middle|\; w \in L \right\}.$$

Prove that the reversal of every regular language is regular.

Solution: Let r be an arbitrary regular expression. We want to derive a regular expression r' such that $L(r') = L(r)^R$.

Assume for every regular expression s smaller than r that there is a regular expression s' such that $L(s') = L(s)^R$.

There are five cases to consider (mirroring the definition of regular expressions).

(a) If $r = \emptyset$, then we set $r' = \emptyset$, so that

$$\begin{aligned} L(r)^R &= L(\emptyset)^R && \text{because } r = \emptyset \\ &= \emptyset^R && \text{because } L(\emptyset) = \emptyset \\ &= \emptyset && \text{because } \emptyset^R = \emptyset \\ &= L(\emptyset) && \text{because } L(\emptyset) = \emptyset \\ &= L(r') && \text{because } r = \emptyset \end{aligned}$$

(b) If $r = w$ for some string $w \in \Sigma^*$, then we set $r' := w^R$, so that

$$\begin{aligned} L(r)^R &= L(w)^R && \text{because } r = w \\ &= \{w\}^R && \text{because } L(\langle \text{string} \rangle) = \{\langle \text{string} \rangle\} \\ &= \{w^R\} && \text{by definition of } L^R \\ &= L(w^R) && \text{because } L(\langle \text{string} \rangle) = \{\langle \text{string} \rangle\} \\ &= L(r') && \text{because } r = w^R \end{aligned}$$

(c) Suppose $r = s^*$ for some regular expression s. The inductive hypothesis implies a regular expressions s' such that $L(s') = L(s)^R$. Let $\boldsymbol{r' = (s')^*}$; then we have

$$\begin{aligned} L(r)^R &= L(s^*)^R && \text{because } r = s^* \\ &= (L(s)^*)^R && \text{by definition of } {}^* \\ &= (L(s)^R)^* && \text{because } (L^R)^* = (L^*)^R \\ &= (L(s'))^* && \text{by definition of } s' \\ &= L((s')^*) && \text{by definition of } {}^* \\ &= L(r') && \text{by definition of } r' \end{aligned}$$

(d) Suppose $r = s + t$ for some regular expressions s and t. The inductive hypothesis implies regular expressions s' and t' such that $L(s') = L(s)^R$ and $L(t') = L(t)^R$.

Set $r' := s' + t'$; then we have

$$\begin{aligned}
L(r)^R &= L(s+t)^R && \text{because } r = s+t \\
&= (L(s) \cup L(t))^R && \text{by definition of } + \\
&= \{w^R \mid w \in (L(s) \cup L(t))\} && \text{by definition of } L^R \\
&= \{w^R \mid w \in L(s) \text{ or } w \cup L(t)\} && \text{by definition of } \cup \\
&= \{w^R \mid w \in L(s)\} \cup \{w^R \mid w \cup L(t)\} && \text{by definition of } \cup \\
&= L(s)^R \cup L(t)^R && \text{by definition of } L^R \\
&= L(s') \cup L(t') && \text{by definition of } s' \text{ and } t' \\
&= L(s'+t') && \text{by definition of } + \\
&= L(r') && \text{by definition of } r'
\end{aligned}$$

(e) Suppose $r = s \bullet t$ for some regular expressions s and t. The inductive hypothesis implies regular expressions s' and t' such that $L(s') = L(s)^R$ and $L(t') = L(t)^R$. Set $r' = t' \bullet s'$; then we have

$$\begin{aligned}
L(r)^R &= L(st)^R && \text{because } r = s+t \\
&= (L(s) \bullet L(t))^R && \text{by definition of } \bullet \\
&= \{w^R \mid w \in (L(s) \bullet L(t))\} && \text{by definition of } L^R \\
&= \{(x \bullet y)^R \mid x \in L(s) \text{ and } y \in L(t)\} && \text{by definition of } \bullet \\
&= \{y^R \bullet x^R \mid x \in L(s) \text{ and } y \in L(t)\} && \text{concatenation reversal} \\
&= \{y' \bullet x' \mid x' \in L(s)^R \text{ and } y' \in L(t)^R\} && \text{by definition of } L^R \\
&= \{y' \bullet x' \mid x' \in L(s') \text{ and } y' \in L(t')\} && \text{by definition of } s' \text{ and } t' \\
&= L(t') \bullet L(s') && \text{by definition of } \bullet \\
&= L(t' \bullet s') && \text{by definition of } \bullet \\
&= L(r') && \text{by definition of } r'
\end{aligned}$$

In all five cases, we have found a regular expression r' such that $L(r') = L(r)^R$. It follows that $L(r)^R$ is regular. ■

Rubric: Standard induction rubric!!

CS/ECE 374 A ✧ Fall 2023

Homework 3

Due Tuesday, September 12, 2023 at 9pm Central Time

- Please start your solution to each *lettered subproblem* ((a), (b), (c), etc.) on a new page. Please also remember to tell Gradescope which page(s) are relevant for which subproblems.

---

1. Prove that the following languages over the alphabet $\Sigma = \{0, 1\}$ are *not* regular.

   (a) $\{0^a 1 0^b 1 0^c \mid 2b = a + c\}$.

   (b) The set of all palindromes in Σ^* whose lengths are divisible by 7.

   (c) $\{1^m 0^n \mid m + n > 0 \text{ and } \gcd(m, n) = 1\}$

   Here $\gcd(m, n)$ denotes the *greatest common divisor* of m and n: the largest integer d such that both m/d and n/d are integers. In particular, $\gcd(1, n) = 1$ and $\gcd(0, n) = n$ for every positive integer n.

2. For each of the following languages over the alphabet $\Sigma = \{0, 1\}$, either prove that the language is regular (by constructing an appropriate DFA, NFA, or regular expression) or prove that the language is not regular (by constructing an infinite fooling set). Recall that Σ^+ denotes the set of all *nonempty* strings over Σ.

   (a) Strings in which the substrings `01` and `10` appear the same number of times. For example, $1100011 \in L$ because both substrings appear once, but $01000011 \notin L$.

   (b) Strings in which the substrings `00` and `11` appear the same number of times. For example, $1100011 \in L$ because both substrings appear twice, but $01000011 \notin L$.

   (c) $\{xyyx \mid x, y \in \Sigma^+\}$

   (d) $\{xyyz \mid x, y, z \in \Sigma^+\}$

   [Hint: Exactly two of these languages are regular.]

*3. ***Practice only. Do not submit solutions.***

A ***Moore machine*** is a variant of a finite-state automaton that produces output; Moore machines are sometimes called finite-state *transducers*. For purposes of this problem, a Moore machine formally consists of six components:

- A finite set Σ called the input alphabet
- A finite set Γ called the output alphabet
- A finite set Q whose elements are called states
- A start state $s \in Q$
- A transition function $\delta : Q \times \Sigma \to Q$
- An output function $\omega : Q \to \Gamma$

More intuitively, a Moore machine is a graph with a special start vertex, where every node (state) has one outgoing edge labeled with each symbol from the input alphabet, and each node (state) is additionally labeled with a symbol from the output alphabet.

The Moore machine reads an input string $w \in \Sigma^*$ one symbol at a time. For each symbol, the machine changes its state according to the transition function δ, and then outputs the symbol $\omega(q)$, where q is the new state. Formally, we recursively define a *transducer* function $\omega^* : Q \times \Sigma^* \to \Gamma^*$ as follows:

$$\omega^*(q, w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \omega(\delta(q,a)) \cdot \omega^*(\delta(q,a), x) & \text{if } w = ax \end{cases}$$

Given input string $w \in \Sigma^*$, the machine outputs the string $\omega^*(w, s) \in \Gamma^*$. The ***output language $L^\circ(M)$*** of a Moore machine M is the set of all strings that the machine can output:

$$L^\circ(M) := \{\omega^*(s, w) \mid w \in \Sigma^*\}$$

(a) Let M be an arbitrary Moore machine. Prove that $L^\circ(M)$ is a regular language.

(b) Let M be an arbitrary Moore machine whose input alphabet Σ and output alphabet Γ are identical. Prove that the language

$$L^=(M) = \{w \in \Sigma^* \mid w = \omega^*(s, w)\}$$

is regular. $L^=(M)$ consists of all strings w such that M outputs w when given input w; these are also called *fixed points* for the transducer function ω^*.

[Hint: These problems are easier than they look!]

Solved problems

4. For each of the following languages, either prove that the language is regular (by constructing an appropriate DFA, NFA, or regular expression) or prove that the language is not regular (by constructing an infinite fooling set).

   Recall that a *palindrome* is a string that equals its own reversal: $w = w^R$. Every string of length 0 or 1 is a palindrome.

   (a) Strings in $(0+1)^*$ in which no prefix of length at least 2 is a palindrome.

   Solution: Regular: $\varepsilon + 01^* + 10^*$. Call this language L_a.

   Let w be an arbitrary non-empty string in $(0+1)^*$. Without loss of generality, assume $w = 0x$ for some string x. There are two cases to consider.

   - If x contains a 0, then we can write $w = 01^n0y$ for some integer n and some string y. The prefix 01^n0 is a palindrome of length at least 2. Thus, $w \notin L_a$.
   - Otherwise, $x \in 1^*$. Every non-empty prefix of w is equal to 01^n for some non-negative integer $n \le |x|$. Every palindrome that starts with 0 also ends with 0, so the only palindrome prefixes of w are ε and 0, both of which have length less than 2. Thus, $w \in L_a$.

   We conclude that $0x \in L_a$ if and only if $x \in 1^*$. A similar argument implies that $1x \in L_a$ if and only if $x \in 0^*$. Finally, trivially, $\varepsilon \in L_a$. ■

   Rubric: 2½ points = ½ for "regular" + 1 for regular expression + 1 for justification. This is more detail than necessary for full credit.

   (b) Strings in $(0+1+2)^*$ in which no prefix of length at least 2 is a palindrome.

   Solution: Not regular. Call this language L_b.

   Consider the set $F = (012)^+$.

   Let x and y be arbitrary distinct strings in F.

   Then $x = (012)^i$ and $y = (012)^j$ for some positive integers $i \neq j$.

   Without loss of generality, assume $i < j$.

   Let z be the suffix $(210)^i$.

   - $xz = (012)^i(210)^i$ is a palindrome of length $6i \ge 2$, so $xz \notin L_b$.
   - $yz = (012)^j(210)^i$ has no palindrome prefixes except ε and 0, because $i < j$, so $yz \in L_b$.

   Thus, z is a distinguishing suffix for x and y.

   We conclude that F is a fooling set for L_b.

   Because F is infinite, L_b cannot be regular. ■

   Rubric: 2½ points = ½ for "not regular" + 2 for fooling set proof (standard rubric, scaled).

(c) Strings in $(0+1)^*$ in which no prefix of length at least 3 is a palindrome.

Solution: Not regular. Call this language L_c.

Consider the set $F = (001101)^+$.

Let x and y be arbitrary distinct strings in F.

Then $x = (001101)^i$ and $y = (001101)^j$ for some positive integers $i \neq j$.

Without loss of generality, assume $i < j$.

Let z be the suffix $(101100)^i$.

- $xz = (001101)^i(101100)^i$ is a palindrome of length $12i \geq 2$, so $xz \notin L_b$.
- $yz = (001101)^j(101100)^i$ has no palindrome prefixes except ε and 0 and 00, because $i < j$, so $yz \in L_b$.

Thus, z is a distinguishing suffix for x and y.

We conclude that F is a fooling set for L_c.

Because F is infinite, L_c cannot be regular. ■

Rubric: 2½ points = ½ for "not regular" + 2 for fooling set proof (standard rubric, scaled).

(d) Strings in $(0+1)^*$ in which no *substring* of length at least 3 is a palindrome.

Solution: Regular. Call this language L_d.

Every palindrome of length at least 3 contains a palindrome substring of length 3 or 4. Thus, the complement language $\overline{L_d}$ is described by the regular expression

$$(0+1)^*(000+010+101+111+0110+1001)(0+1)^*$$

Thus, $\overline{L_d}$ is regular, so its complement L_d is also regular. ■

Solution: Regular. Call this language L_d.

In fact, L_d is *finite*! Appending either 0 or 1 to any of the underlined strings creates a palindrome suffix of length 3 or 4.

$$\varepsilon + 0 + 1 + 00 + 01 + 10 + 11 + 001 + \underline{011} + \underline{100} + 110 + \underline{0011} + \underline{1100}$$

■

Rubric: 2½ points = ½ for "regular" + 2 for proof:

- 1 for expression for $\overline{L_d}$ + 1 for applying closure
- 1 for regular expression + 1 for justification

CS/ECE 374 A ✧ Fall 2023

Homework 4

Due Tuesday, September 19, 2023 at 9pm Central Time

This is the last homework before Midterm 1.

---

1. Recall the following string functions from Homework 1:

$$stutter(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ aa \bullet stutter(x) & \text{if } w = ax \end{cases} \qquad \mathsf{grow}(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \mathtt{0} \cdot \mathsf{grow}(x) & \text{if } w = \mathtt{1}x \\ \mathtt{10} \bullet \mathsf{grow}(x) & \text{if } w = \mathtt{0}x \end{cases}$$

For example, $stutter(\mathtt{1001}) = \mathtt{11000011}$, and $\mathsf{grow}(\mathtt{1001}) = \mathtt{0} \bullet \mathtt{10} \bullet \mathtt{10} \bullet \mathtt{0} = \mathtt{010100}$.

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

(a) $\textsc{Stutter}(L) = \{stutter(w) \mid w \in L\}$

(b) $\textsc{Unstutter}(L) = \{w \mid stutter(w) \in L\}$

(c) $\textsc{Grow}(L) = \{grow(w) \mid w \in L\}$

(d) $\textsc{Ungrow}(L) = \{w \mid grow(w) \in L\}$

2. Give context-free grammars for the following languages, and clearly explain how they work and the set of strings generated by each nonterminal. Grammars with unclear or missing explanations may receive little or no credit. On the other hand, we do *not* want formal proofs of correctness.

(a) $\left\{\mathtt{0}^a\mathtt{1}\mathtt{0}^b\mathtt{1}\mathtt{0}^c \;\middle|\; b = 2a + 2c\right\}$.

(b) $\left\{\mathtt{0}^a\mathtt{1}\mathtt{0}^b\mathtt{1}\mathtt{0}^c \;\middle|\; 2b = a + c\right\}$.

(c) The set of all palindromes in Σ^* whose lengths are divisible by 7.

*(d) ***Practice only. Do not submit solutions.***
Strings in which the substrings $\mathtt{00}$ and $\mathtt{11}$ appear the same number of times. For example, $\mathtt{1100011} \in L$ because both substrings appear twice, but $\mathtt{01000011} \notin L$.

Yes, you've seen most of these languages before.

*3. ***Practice only. Do not submit solutions.***

Let L_1 and L_2 be arbitrary regular languages over the alphabet $\Sigma = \{0, 1\}$. Prove that the following languages are also regular.

(a) $\textsc{Faro}(L_1, L_2) := \{faro(x, z) \mid x \in L_1 \text{ and } z \in L_2 \text{ with } |x| = |z|\}$, where

$$faro(x, z) := \begin{cases} z & \text{if } x = \varepsilon \\ a \cdot faro(z, y) & \text{if } x = ay \end{cases}$$

For example, $faro(0011, 0101) = 00011011$ and $\textsc{Faro}(0^*, 1^*) = (01)^*$.

(b) $\textsc{Shuffles}(L_1, L_2) := \bigcup_{w \in L_1, y \in L_2} shuffles(w, y)$, where $shuffles(w, y)$ is the set of all strings obtained by shuffling w and y, or equivalently, all strings in which w and y are complementary subsequences. Formally:

$$shuffles(w, y) = \begin{cases} \{y\} & \text{if } w = \varepsilon \\ \{w\} & \text{if } y = \varepsilon \\ \{a\} \bullet shuffles(x, y) \cup \{b\} \bullet shuffles(w, z) & \text{if } w = ax \text{ and } y = bz \end{cases}$$

For example, $shuffles(001, 1) = \{0011, 0101, 1001\}$ and $shuffles(00, 11) = \{0011, 0101, 0110, 1001, 1010, 1100\}$. Finally, $\textsc{Shuffles}(0^*, 1^*) = (0 + 1)^*$.

Both of these names are taken from methods of mixing a deck of playing cards. A *shuffle* divides the deck into two smaller stacks, and then interleaves those two stacks arbitrarily. A *Faro shuffle* or *perfect shuffle* divides the pack of cards exactly in half, and then interleaves them perfectly; the final deck alternates between cards from one half and cards from the other half. Faro shuffles are the basis of several card tricks.

Solved problems

4. (a) Fix an arbitrary regular language L. Prove that the language $\mathit{half}(L) := \{w \mid ww \in L\}$ is also regular.

Solution: Let $M = (\Sigma, Q, s, A, \delta)$ be an arbitrary DFA that accepts L. We define a new NFA $M' = (\Sigma, Q', s', A', \delta')$ with ε-transitions that accepts $\mathit{half}(L)$, as follows:

$$\begin{aligned} Q' &= (Q \times Q \times Q) \cup \{s'\} \\ s' &\text{ is an explicit state in } Q' \\ A' &= \{(h,h,q) \mid h \in Q \text{ and } q \in A\} \\ \delta'(s', \varepsilon) &= \{(s,h,h) \mid h \in Q\} \\ \delta'(s', a) &= \emptyset \\ \delta'((p,h,q), \varepsilon) &= \emptyset \\ \delta'((p,h,q), a) &= \{(\delta(p,a), h, \delta(q,a))\} \end{aligned}$$

M' reads its input string w and simulates M reading the input string ww. Specifically, M' simultaneously simulates two copies of M, one reading the left half of ww starting at the usual start state s, and the other reading the right half of ww starting at some intermediate state h.

- The new start state s' non-deterministically guesses the "halfway" state $h = \delta^*(s, w)$ without reading any input; this is the only non-determinism in M'.
- State (p,h,q) means the following:
  - The left copy of M (which started at state s) is now in state p.
  - The initial guess for the halfway state is h.
  - The right copy of M (which started at state h) is now in state q.
- M' accepts if and only if the left copy of M ends at state h (so the initial non-deterministic guess $h = \delta^*(s, w)$ was correct) and the right copy of M ends in an accepting state.

■

Solution (smartass): A complete solution is given in the lecture notes. ■

Rubric: 5 points: standard langage transformation rubric (scaled). Yes, the smartass solution would be worth full credit.

(b) Describe a regular language L such that the language $double(L) := \{ww \mid w \in L\}$ is *not* regular. Prove your answer is correct.

Solution: Consider the regular language $L = 0^*1$.

Expanding the regular expression lets us rewrite $L = \{0^n1 \mid n \geq 0\}$. It follows that $double(L) = \{0^n10^n1 \mid n \geq 0\}$. I claim that this language is not regular.

Let x and y be arbitrary distinct strings in L.

Then $x = 0^i1$ and $y = 0^j1$ for some integers $i \neq j$.

Then x is a distinguishing suffix of these two strings, because

- $xx \in double(L)$ by definition, but
- $yx = 0^i10^j1 \notin double(L)$ because $i \neq j$.

We conclude that L is a fooling set for $double(L)$.

Because L is infinite, $double(L)$ cannot be regular. ■

Solution: Consider the regular language $L = \Sigma^* = (0 + 1)^*$.

I claim that the language $double(\Sigma^*) = \{ww \mid w \in \Sigma^*\}$ is not regular.

Let F be the infinite language 01^*0.

Let x and y be arbitrary distinct strings in F.

Then $x = 01^i0$ and $y = 01^j0$ for some integers $i \neq j$.

The string $z = 1^i$ is a distinguishing suffix of these two strings, because

- $xz = 01^i01^i = ww$ where $w = 01^i$, so $xz \in double(\Sigma^*)$, but
- $yx = 01^j01^i \notin double(\Sigma^*)$ because $i \neq j$.

We conclude that F is a fooling set for $double(\Sigma^*)$.

Because F is infinite, $double(\Sigma^*)$ cannot be regular. ■

Rubric: 5 points:

- 2 points for describing a regular language L such that $double(L)$ is not regular.
- 3 point for the fooling set proof (standard fooling set rubric, scaled and rounded)

These are not the only correct solutions. These are not the only fooling sets for these languages.

5. Give context-free grammars for the following languages over the alphabet $\Sigma = \{0, 1\}$. Clearly explain how they work and the set of strings generated by each nonterminal. Grammars with unclear or missing explanations may receive little or no credit; on the other hand, we do *not* want formal proofs of correctness.

   (a) In any string, a ***run*** is a maximal non-empty substring of identical symbols. For example, the string $0111000011001 = 0^1 1^3 0^4 1^2 0^2 1^1$ consists of six runs.

   Let L_a be the set of all strings in Σ^* that contain two runs of 0s of equal length. For example, L_a contains the strings 01101111 and 01001011100010 (because each of those strings contains more than one run of 0s of length 1) but L_a does not contain the strings 000110011011 and 00000000111.

Solution:

$S \to ACB$	strings with two blocks of 0s of same length
$A \to \varepsilon \mid X1$	empty or ends with 1
$B \to \varepsilon \mid 1X$	empty or starts with 1
$C \to 0C0 \mid 0D0$	$0^n y 0^n$, where y starts and ends with 1
$D \to 1 \mid 1X1$	starts and ends with 1
$X \to \varepsilon \mid 1X \mid 0X$	all strings: $(0+1)^*$

Every string in L has the form $x0^n y0^n z$, where x is either empty or ends with 1, y starts and ends with 1, and z is either empty or begins with 1. Nonterminal A generates the prefix x; non-terminal B generates the suffix z; nonterminal C generates the matching runs of 0s, and nonterminal D generates the interior string y.

The same decomposition can be expressed more compactly as follows:

$S \to B \mid B1A \mid A1B \mid A1B1A$	strings with two blocks of 0s of same length
$A \to 1A \mid 0A \mid \varepsilon$	all strings: $(0+1)^*$
$B \to 0B0 \mid 010 \mid 01A10$	$0^n y 0^n$, where y starts and ends with 1

■

Rubric: 5 points = 3 for clearly correct grammar + 2 for clear explanation. These are not the only correct solutions.

(b) $L_b = \{w \in \Sigma^* \mid w \text{ is } \textit{not} \text{ a palindrome}\}$.

Solution:

$S \to 0S0 \mid 0S1 \mid 1S0 \mid 1S1 \mid A$	non-palindromes
$A \to 0B1 \mid 1B0$	start and end with different symbols
$B \to 0B \mid 1B \mid \varepsilon$	all strings

Every non-palindrome w can be decomposed as either $w = x0y1z$ or $w = x1y0z$, for some substrings x, y, z such that $|x| = |z|$. Non-terminal S generates the prefix x and matching-length suffix z; non-terminal A generates the distinct symbols, and non-terminal B generates the interior substring y. ■

Solution:

$S \to 0S0 \mid 1S1 \mid A$	non-palindromes
$A \to 0B1 \mid 1B0$	start and end with different symbols
$B \to 0B \mid 1B \mid \varepsilon$	all strings

Every non-palindrome w must have a prefix x and a substring y such that either $w = x0y1x^R$ or $w = x1y0x^R$. Specifically, x is the longest common prefix of w and w^R. In the first case, the grammar generates w as follows:

$$S \rightsquigarrow^* xAx^R \rightsquigarrow x0B1x^R \rightsquigarrow^* x0y1x^R = w$$

The derivation for $w = x1y0x^R$ is similar. ■

Rubric: 5 points = 3 for clearly correct grammar + 2 for clear explanation. These are not the only correct solutions.

CS/ECE 374 A ✧ Fall 2023

Homework 5

Due Tuesday, October 3, 2023 at 9pm Central Time

---

1. In the lab on Wednesday, you'll see an algorithm that finds a local minimum in a one-dimensional array in $O(\log n)$ time. This question asks you to consider two higher-dimensional versions of this problem.

   (a) Suppose we are given a two-dimensional array $A[1..n, 1..n]$ of distinct integers. An array element $A[i, j]$ is called a ***local minimum*** if it is smaller than its four immediate neighbors:

   $$A[i, j] < \min\{A[i-1, j],\ A[i+1, j],\ A[i, j-1],\ A[i, j+1]\}$$

   To avoid edge cases, we assume all cells in row 1, row n, column 1, and column n have value $+\infty$.

   Describe and analyze an algorithm to find a local minimum in A as quickly as possible. (Remember that faster algorithms are worth more points, but only if they are correct.)

   [Hint: Suppose $A[i, j]$ is the smallest element in row i. If $A[i, j]$ is smaller than both of its vertical neighbors $A[i-1, j]$ and $A[i+1, j]$, we are clearly done. But what if $A[i, j] > A[i+1, j]$?]

   *[Hint: This problem is more subtle than it appears at first glance; many published solutions for this problem on the internet are incorrect. The main issue is that a local minimum in a rectangular subarray is not necessarily a local minimum in the original array. Design a recursive algorithm for the following more general problem: Given a two-dimensional array that contains a local minimum **whose value is less than the value of every border cell**, find such a local minimum.]*

   (b) Now suppose we are given a three-dimensional array $A[1..n, 1..n, 1..n]$ of distinct integers. An array element $A[i, j, k]$ is called a ***local minimum*** if it is smaller than its six immediate neighbors:

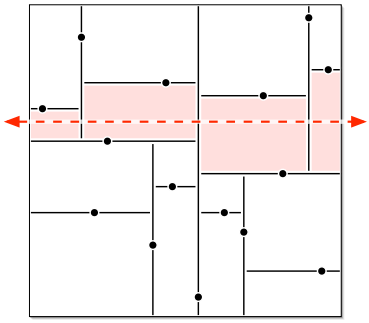
   $$A[i, j] < \min\left\{\begin{array}{l} A[i-1, j, k],\ A[i+1, j, k], \\ A[i, j-1, k],\ A[i, j+1, k], \\ A[i, j, k-1],\ A[i, j, k+1] \end{array}\right\}$$

   To avoid edge cases, we assume all cells on the boundary of the array have value $+\infty$.

   Describe and analyze an algorithm to find a local minimum in A as quickly as possible.

(Remember that faster algorithms are worth more points, but only if they are correct.)

2. Suppose we have n points scattered inside a two-dimensional box. A *kd-tree* recursively subdivides the points as follows. First we split the box into two smaller boxes with a *vertical* line, then we split each of those boxes with *horizontal* lines, and so on, always alternating between horizontal and vertical splits. Each time we split a box, the splitting line partitions the rest of the interior points *as evenly as possible* by passing through a median point in the interior of the box (*not* on its boundary). If a box doesn't contain any points, we don't split it any more; these final empty boxes are called *cells*.



A kd-tree for 15 points. The dashed line crosses the four shaded cells.

(a) How many cells does the kd-tree have, as a function of n? Prove that your answer is correct.

(b) In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n? Prove that your answer is correct. Assume that $n = 2^k - 1$ for some integer k. *[Hint: There is more than one function f such that $f(15) = 4$.]*

(c) Suppose we have n points stored in a kd-tree. Describe and analyze an algorithm that counts the number of points above a given horizontal line (such as the dashed line in the figure) as quickly as possible. *[Hint: Use part (b).]*

I should have specified that the following information is stored in each internal node v in the kd-tree:

- $v.x$ and $v.y$: The coordinates of the point defining the cut at v
- $v.dir \in \{vertical, horizontal\}$: The direction of the cut at v.
- $v.left$ and $v.right$: The children of v if $v.dir = vertical$
- $v.up$ and $v.down$: The children of v if $v.dir = horizontal$
- $v.size$: the number of points=cuts in the subtree rooted at v.

Instead I allowed arbitrary information to be computed in preprocessing; that freedom allows a much simpler and more efficient query algorithm!

(d) Describe and analyze an efficient algorithm that counts, given a kd-tree storing n points, the number of points that lie inside a given rectangle R with horizontal and vertical sides. *[Hint: Use part (c).]*

Assume that all x-coordinates and y-coordinates are distinct; that is, no two points lie on the same horizontal line or the same vertical line, no point lies on the query line in part (c), and no point lies on the boundary of the query rectangle in part (d).

*3. ***Practice only. Do not submit solutions.***

The following variant of the infamous StoogeSort algorithm[1] was discovered by the British actor Patrick Troughton during rehearsals for the 20th anniversary *Doctor Who* special "The Five Doctors".[2]

```
WhoSort(A[1..n]):
  if n < 13
      sort A by brute force
  else
      k = ⌈n/5⌉
      WhoSort(A[1..3k])          ⟨⟨Hartnell⟩⟩
      WhoSort(A[2k+1..n])        ⟨⟨Troughton⟩⟩
      WhoSort(A[1..3k])          ⟨⟨Pertwee⟩⟩
      WhoSort(A[k+1..4k])        ⟨⟨Davison⟩⟩
```

(a) Prove by induction that WhoSort correctly sorts its input. *[Hint: Where can the smallest k elements be?]*

(b) Would WhoSort still sort correctly if we replaced "if $n < 13$" with "if $n < 4$"? Justify your answer.

(c) Would WhoSort still sort correctly if we replaced "$k = \lceil n/5 \rceil$" with "$k = \lfloor n/5 \rfloor$"? Justify your answer.

(d) What is the running time of WhoSort? (Set up a running-time recurrence and then solve it, ignoring the floors and ceilings.)

(e) Forty years later, 15th Doctor Ncuti Gatwa discovered the following optimization to WhoSort, which uses the standard Merge subroutine from mergesort, which merges two sorted arrays into one sorted array.

```
NuWhoSort(A[1..n]):
  if n < 13
      sort A by brute force
  else
      k = ⌈n/5⌉
      NuWhoSort(A[1..3k])                ⟨⟨Grant⟩⟩
      NuWhoSort(A[2k+1..n])              ⟨⟨Whittaker⟩⟩
      Merge(A[1..2k], A[2k+1..4k])       ⟨⟨Tennant⟩⟩
```

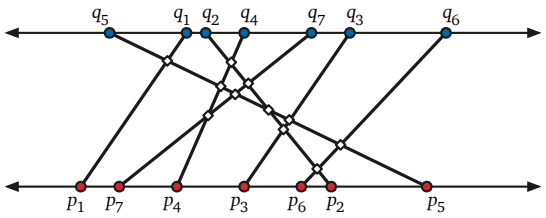
What is the running time of NuWhoSort?

[1] https://en.wikipedia.org/wiki/Stooge_sort

[2] Tom Baker, the fourth Doctor, declined to return for the reunion; hence, only four Doctors appeared in "The Five Doctors". (Well, okay, technically the BBC used excerpts of the unfinished episode "Shada" to include Baker, but he wasn't really *there*—to the extent that any fictional character in a television show about a time traveling wizard arguing with several other versions of himself about immortality can be said to be "really" "there".)

Solved problems

4. Suppose we are given two sets of n points, one set $\{p_1, p_2, \ldots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \ldots, q_n\}$ on the line $y = 1$. Consider the n line segments connecting each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time. See the example below.



Seven segments with endpoints on parallel lines, with 11 intersecting pairs.

Your input consists of two arrays $P[1..n]$ and $Q[1..n]$ of x-coordinates; you may assume that all $2n$ of these numbers are distinct. No proof of correctness is necessary, but you should justify the running time.

Solution: We begin by sorting the array $P[1..n]$ and permuting the array $Q[1..n]$ to maintain correspondence between endpoints, in $O(n \log n)$ time. Then for any indices $i < j$, segments i and j intersect if and only if $Q[i] > Q[j]$. Thus, our goal is to compute the number of pairs of indices $i < j$ such that $Q[i] > Q[j]$. Such a pair is called an ***inversion***.

We count the number of inversions in Q using the following extension of mergesort; as a side effect, this algorithm also sorts Q. If $n < 100$, we use brute force in $O(1)$ time. Otherwise:

- Color the elements in the Left half $Q[1..\lfloor n/2 \rfloor]$ bLue.
- Color the elements in the Right half $Q[\lfloor n/2 \rfloor + 1..n]$ Red.
- Recursively count inversions in (and sort) the blue subarray $Q[1..\lfloor n/2 \rfloor]$.
- Recursively count inversions in (and sort) the red subarray $Q[\lfloor n/2 \rfloor + 1..n]$.
- Count red/blue inversions as follows:
  - MERGE the sorted subarrays $Q[1..n/2]$ and $Q[n/2+1..n]$, maintaining the element colors.
  - For each blue element $Q[i]$ of the now-sorted array $Q[1..n]$, count the number of smaller red elements $Q[j]$.

The last substep can be performed in $O(n)$ time using a simple for-loop:

```
CountRedBlue(A[1..n]):
  count ← 0
  total ← 0
  for i ← 1 to n
      if A[i] is red
          count ← count + 1
      else
          total ← total + count
  return total
```

Merge and CountRedBlue each run in $O(n)$ time. Thus, the running time of our inversion-counting algorithm obeys the mergesort recurrence $T(n) = 2T(n/2) + O(n)$. (We can safely ignore the floors and ceilings in the recursive arguments.) We conclude that the overall running time of our algorithm is $O(n \log n)$, as required.

Rubric: This is enough for full credit.

In fact, we can execute the third merge-and-count step directly by modifying the Merge algorithm, without any need for "colors". Here changes to the standard Merge algorithm are indicated in red.

```
MergeAndCount(A[1..n], m):
  i ← 1; j ← m + 1; count ← 0; total ← 0
  for k ← 1 to n
      if j > n
          B[k] ← A[i]; i ← i + 1; total ← total + count
      else if i > m
          B[k] ← A[j]; j ← j + 1; count ← count + 1
      else if A[i] < A[j]
          B[k] ← A[i]; i ← i + 1; total ← total + count
      else
          B[k] ← A[j]; j ← j + 1; count ← count + 1
  for k ← 1 to n
      A[k] ← B[k]
  return total
```

We can further optimize MergeAndCount by observing that *count* is always equal to $j - m - 1$, so we don't need an additional variable. (Proof: Initially, $j = m + 1$ and $count = 0$, and we always increment j and *count* together.)

```
MergeAndCount2(A[1..n], m):
  i ← 1;  j ← m + 1;  total ← 0
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i + 1;  total ← total + j − m − 1
      else if i > m
          B[k] ← A[j];  j ← j + 1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i + 1;  total ← total + j − m − 1
      else
          B[k] ← A[j];  j ← j + 1
  for k ← 1 to n
      A[k] ← B[k]
  return total
```

MergeAndCount2 still runs in $O(n)$ time, so the overall running time is still $O(n \log n)$, as required. ■

Rubric: 10 points = 2 for base case + 2 for divide (split and recurse) + 4 for conquer (merge and count) + 2 for time analysis. This is neither the only way to correctly describe this algorithm nor the only correct $O(n \log n)$-time algorithm. No proof of correctness is required.

Max 3 points for a correct $O(n^2)$-time algorithm.

Notice that each boxed algorithm is preceded by a clear English description of the task that algorithm performs—not how the algorithm works, but the relationship between its input and its output. **Each English description is worth 25% of the credit for that algorithm** (rounding to the nearest half-point). For example, the CountRedBlue algorithm is worth 4 points ("conquer"); the English description alone ("For each blue element $Q[i]$ of the now-sorted array $Q[1..n]$, count the number of smaller red elements $Q[j]$.") is worth 1 point.

CS/ECE 374 A ✧ Fall 2023

Homework 6

Due Tuesday, October 10, 2023 at 9pm Central Time

Please make sure that you read and understand the standard dynamic programming rubric.

---

1. Satya is in charge of establishing a new testing center for the Standardized Awesomeness Test (SAT), and found an old conference hall that is perfect. The conference hall has n rooms of various sizes along a single long hallway, numbered in order from 1 through n. Satya knows exactly how many students fit into each room, and he wants to use a subset of the rooms to host as many students as possible for testing.

   Unfortunately, there have been several incidents of students cheating at other testing centers by tapping secret codes through walls. To prevent this type of cheating, Satya can use two adjacent rooms only if he demolishes the wall between them. The city's chief architect has determined that demolishing the walls on both sides of the same room would threaten the building's structural integrity. For this reason, Satya can never host students in three consecutive rooms.

   Describe an efficient algorithm that computes the largest number of students that Satya can host for testing without using three consecutive rooms. The input to your algorithm is an array $S[1..n]$, where each $S[i]$ is the (non-negative integer) number of students that can fit in room i.

2. As a typical overworked college student, you occasionally pull all-nighters to get more work done. Painful experience has taught you that the longer you stay awake, the less productive you are.

   Suppose there are n days left in the semester. For each of the next n days, you can either stay awake and work, or you can sleep. You have an array $Score[1..n]$, where $Score[i]$ is the (always positive) number of points you will earn on day i *if you are awake and well-rested*.

   However, staying awake for several days in a row has a price: Each consecutive day you stay awake cuts the quality of your work in half. Thus, if you are awake on day i, and you most recently slept on day $i-k$, then you will actually earn $Score[i]/2^{k-1}$ points on day i. (You've already decided to sleep on day 0.)

   For example, suppose $n = 6$ and $Score = [3, 7, 4, 3, 9, 1]$.

   - If you work on all six days, you will earn $3 + \frac{7}{2} + \frac{4}{4} + \frac{3}{8} + \frac{9}{16} + \frac{1}{32} = 8.46875$ points.
   - If you work only on days 1, 3, and 5, you will earn $3 + 4 + 9 = 16$ points.
   - If you work only on days 2, 3, 5, and 6, you will earn $7 + \frac{4}{2} + 9 + \frac{1}{2} = 18.5$ points.

   Design and analyze an algorithm that computes the maximum number of points you can earn, given the array $Score[1..n]$ as input. For example, given the input array $[3, 7, 4, 3, 9, 1]$, your algorithm should return the number 18.5.

   VERY IMPORTANT: Do not actually do this in real life!

3. ***Practice only. Do not submit solutions.***

(a) Any string can be decomposed into a sequence of palindromes. For example, the string BUBBASEESABANANA ("Bubba sees a banana.") can be broken into palindromes in the following ways (and 65 others):

BUB • BASEESAB • ANANA

B • U • BB • ASEESA • B • ANANA

BUB • B • A • SEES • ABA • N • ANA

B • U • BB • A • S • EE • S • A • B • A • NAN • A

B • U • B • B • A • S • E • E • S • A • B • A • N • A • N • A

Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string BUBBASEESABANANA, your algorithm should return 3.

(b) A ***metapalindrome*** is a decomposition of a string into a sequence of palindromes, such that the sequence of palindrome lengths is itself a palindrome. For example, the string BOBSMAMASEESAUKULELE ("Bob's mama sees a ukulele") has the following metapalindromes (among others):

BOB • S • MAM • ASEESA • UKU • L • ELE

B • O • B • S • M • A • M • A • S • E • E • S • A • U • K • U • L • E • L • E

The length sequences of these metapalindromes are $(3, 1, 3, 6, 3, 1, 3)$ and $(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$; notice that both of these sequences are themselves palindromes.

Describe and analyze an efficient algorithm to find the smallest number of palindromes in any metapalindrome for a given string. For example, given the input string BOBSMAMASEESAUKULELE, your algorithm should return 7.

Solved Problems

3. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings **BANANA** and ANANAS in several different ways.

BANANAANANAS **BAN**ANA**ANA**NAS **B**AN**AN**A**NA**N**A**S

Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of the strings **DYNAMIC** and PROGRAMMING:

PRO**D**G**Y**R**NAM**AMMI**I**N**C**G **DY**PRO**N**G**A**R**M**AMM**IC**ING

(a) Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

Solution: We define a boolean function $Shuf(i, j)$, which is TRUE if and only if the prefix $C[1..i+j]$ is a shuffle of the prefixes $A[1..i]$ and $B[1..j]$. We need to compute $Shuf(m, n)$. The function $Shuf$ satisfies the following recurrence:

$$Shuf(i,j) = \begin{cases} \text{TRUE} & \text{if } i = j = 0 \\ Shuf(0, j-1) \wedge (B[j] = C[j]) & \text{if } i = 0 \text{ and } j > 0 \\ Shuf(i-1, 0) \wedge (A[i] = C[i]) & \text{if } i > 0 \text{ and } j = 0 \\ \big(Shuf(i-1, j) \wedge (A[i] = C[i+j])\big) & \\ \quad \vee \big(Shuf(i, j-1) \wedge (B[j] = C[i+j])\big) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $Shuf[0..m][0..n]$. Each array entry $Shuf[i, j]$ depends only on the entries immediately above and immediately to the left: $Shuf[i-1, j]$ and $Shuf[i, j-1]$. Thus, we can fill the array in standard row-major order in $O(mn)$ *time*. ■

Solution: The following algorithm runs in $O(mn)$ *time*.

```
IsShuffle?(A[1..m], B[1..n], C[1..m+n]):
  Shuf[0,0] ← True
  for j ← 1 to n
      Shuf[0,j] ← Shuf[0,j−1] ∧ (B[j] = C[j])
  for i ← 1 to n
      Shuf[i,0] ← Shuf[i−1,0] ∧ (A[i] = B[i])
  for j ← 1 to n
      Shuf[i,j] ← False
      if A[i] = C[i+j]
          Shuf[i,j] ← Shuf[i−1,j]
      if B[i] = C[i+j]
          Shuf[i,j] ← Shuf[i,j] ∨ Shuf[i,j−1]
  return Shuf[m,n]
```

Here $Shuf(i, j) = \text{TRUE}$ if and only if the prefix $C[1..i+j]$ is a shuffle of the

prefixes $A[1..i]$ and $B[1..j]$. ■

Rubric: 5 points, standard dynamic programming rubric. **Each of these solutions is separately worth full credit.** These are not the only correct solutions. −½ for reporting running time as $O(n^2)$. 3 points for a slower polynomial-time algorithm; scale partial credit accordingly.

(b) Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine *the number of different ways* that A and B can be shuffled to obtain C.

Solution: Let $\#Shuf(i,j)$ denote the number of different ways that the prefixes $A[1..i]$ and $B[1..j]$ can be shuffled to obtain the prefix $C[1..i+j]$. We need to compute $\#Shuf(m,n)$.

The $\#Shuf$ function satisfies the following recurrence. Here I am using Iverson bracket notation to convert booleans to integers: For any proposition P, the expression $[P]$ is equal to 1 if P is true and 0 if P is false.

$$\#Shuf(i,j) = \begin{cases} 1 & \text{if } i = j = 0 \\ \#Shuf(0,j-1) \cdot \big[B[j] = C[j]\big] & \text{if } i = 0 \text{ and } j > 0 \\ \#Shuf(i-1,0) \cdot \big[A[i] = C[i]\big] & \text{if } i > 0 \text{ and } j = 0 \\ \big(\#Shuf(i-1,j) \cdot \big[A[i] = C[i]\big]\big) & \\ \qquad + \big(\#Shuf(i,j-1) \cdot \big[B[j] = C[j]\big]\big) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $\#Shuf[0..m][0..n]$. As in part (a), we can fill this array in standard row-major order in ***O(mn) time***. ■

Solution: The following algorithm runs in ***O(mn) time***:

```
NumShuffles(A[1..m], B[1..n], C[1..m+n]):
  #Shuf[0,0] ← 1
  for j ← 1 to n
      #Shuf[0,j] ← 0
      if (B[j] = C[j])
          #Shuf[0,j] ← #Shuf[0,j−1]
  for i ← 1 to n
      #Shuf[0,j] ← 0
      if (A[i] = B[i])
          #Shuf[0,j] ← #Shuf[i−1,0]
  for j ← 1 to n
      #Shuf[i,j] ← 0
      if A[i] = C[i+j]
          #Shuf[i,j] ← #Shuf[i−1,j]
      if B[i] = C[i+j]
          #Shuf[i,j] ← #Shuf[i,j] + #Shuf[i,j−1]
  return Shuf[m,n]
```

Here $\#Shuf[i, j]$ stores the number of different ways that the prefixes $A[1..i]$ and $B[1..j]$ can be shuffled to obtain the prefix $C[1..i+j]$. ■

Rubric: 5 points, standard dynamic programming rubric. **Again, each of these solutions is separately worth full credit.** These are not the only correct solutions. −½ for reporting running time as $O(n^2)$. 3 points for a slower polynomial-time algorithm; scale partial credit accordingly.

CS/ECE 374 A ✧ Fall 2023

Homework 7

Due Tuesday, October 17, 2023 at 9pm Central Time

---

1. The City Council of Sham-Poobanana needs to partition Purple Street into voting districts. A total of n people live on Purple Street, at consecutive addresses $1, 2, \ldots, n$. Each voting district must be a contiguous interval of addresses $i, i+1, \ldots, j$ for some $1 \le i < j \le n$. By law, each Purple Street address must lie in exactly one district, and the number of addresses in each district must be between k and $2k$, where k is a positive integer parameter.

   Every election in Sham-Poobanana is between two rival factions: Oceania and Eurasia. A majority of the current City Council are from Oceania, so they consider a district to be *good* if more than half the residents of that district voted for Oceania in the previous election. Naturally, the City Council has complete voting records for all n residents.

   For example, the figure below shows a legal partition of 22 addresses (of which 9 are good and 13 are bad) into 4 good districts and 3 bad districts, where $k = 2$ (so each district contains either 2, 3, or 4 addresses). Each **O** indicates a vote for Oceania, and each **X** indicates a vote for Eurasia.

   Describe an algorithm to find the largest possible number of *good* districts in a legal partition. Your input consists of the integer k and a boolean array GoodVote$[1..n]$ indicating which residents previously voted for Oceania (True) or Eurasia (False). You can assume that a legal partition exists. Analyze the running time of your algorithm in terms of the parameters n and k. (In particular, do **not** assume that k is a constant.)

2. The StupidScript language includes a binary operator `@` that computes the *average* of its two arguments. For example, the StupidScript code `print(3 @ 6)` would print `4.5`, because $(3+6)/2 = 4.5$.

   Expressions like $3 \mathbin{@} 7 \mathbin{@} 4$ that use the `@` operator more than once yield different results when they are evaluated in different orders:

$$(3 \mathbin{@} 7) \mathbin{@} 4 = 5 \mathbin{@} 4 = 4.5 \qquad \text{but} \qquad 3 \mathbin{@} (7 \mathbin{@} 4) = 3 \mathbin{@} 5.5 = 4.25$$

   Here is a larger example:

$$\begin{aligned}((((8 \mathbin{@} 6) \mathbin{@} 7) \mathbin{@} 5) \mathbin{@} 3) \mathbin{@} (0 \mathbin{@} 9) &= 4.5\\ ((8 \mathbin{@} 6) \mathbin{@} (7 \mathbin{@} 5)) \mathbin{@} ((3 \mathbin{@} 0) \mathbin{@} 9) &= 5.875\\ (8 \mathbin{@} (6 \mathbin{@} (7 \mathbin{@} (5 \mathbin{@} (3 \mathbin{@} 0))))) \mathbin{@} 9 &= 7.890625\end{aligned}$$

   Your goal for this problem is to describe and analyze an algorithm to compute, given a sequence of integers separated by `@` signs, the ***largest possible*** value the expression can take by adding parentheses. Your input is an array $A[1..n]$ listing the sequence of integers.

   For example, if your input sequence is $[3, 7, 4]$, your algorithm should return 4.5, and if your input sequence is $[8, 6, 7, 5, 3, 0, 9]$, your algorithm should return 7.890625. Assume all arithmetic operations (including `@`) can be performed exactly in $O(1)$ time.

   (a) Tommy Tutone suggests the following natural greedy algorithm: Merge the adjacent pair of numbers with the smallest average (breaking ties arbitrarily), replace them with their average, and recurse. For example:

$$\begin{gathered}8 \mathbin{@} 6 \mathbin{@} 7 \mathbin{@} 5 \mathbin{@} \underline{3 \mathbin{@} 0} \mathbin{@} 9\\ 8 \mathbin{@} 6 \mathbin{@} 7 \mathbin{@} \underline{5 \mathbin{@} 1.5} \mathbin{@} 9\\ 8 \mathbin{@} 6 \mathbin{@} \underline{7 \mathbin{@} 3.25} \mathbin{@} 9\\ 8 \mathbin{@} \underline{6 \mathbin{@} 5.125} \mathbin{@} 9\\ \underline{8 \mathbin{@} 5.5625} \mathbin{@} 9\\ \underline{6.78125 \mathbin{@} 9}\\ 7.890625\end{gathered}$$

   Tommy reasons that with an efficient priority queue, this algorithm will run in $O(n \log n)$ time, which is *way* faster than any dynamic programming algorithm.

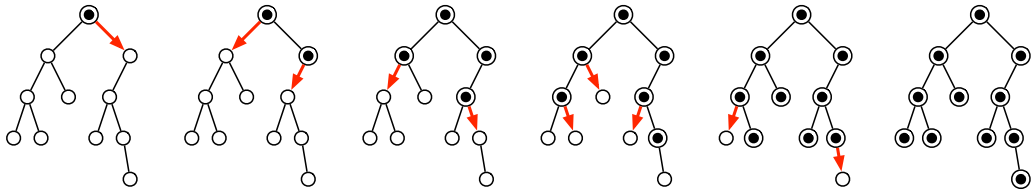
   Prove that Tommy's algorithm is incorrect, by describing a specific input array and proving that his algorithm does not yield the largest possible value for that array.

   (b) Describe and analyze a correct algorithm for this problem. Poor, poor Tommy.

3. ***Practice only. Do not submit solutions.***

Suppose we need to broadcast a message to all the nodes in a rooted binary tree. Initially, only the root node knows the message. In a single round, any node that knows the message can forward it to at most one of its children. See the figure below for an example.

Design an algorithm to compute the minimum number of rounds required to broadcast the message to every node.



A message being distributed through a binary tree in five rounds.

Solved problems

3. A string w of parentheses `(` and `)` and brackets `[` and `]` is ***balanced*** if and only if w is generated by the following context-free grammar:

$$S \to \varepsilon \mid (S) \mid [S] \mid SS$$

For example, the string $w = \texttt{([()][]())[()()]()}$ is balanced, because $w = xy$, where

$$x = \texttt{([()][]())} \qquad \text{and} \qquad y = \texttt{[()()]()}.$$

Describe and analyze an algorithm to compute the length of a longest balanced subsequence of a given string of parentheses and brackets. Your input is an array $A[1..n]$, where $A[i] \in \{(,), [,]\}$ for every index i.

Solution: Suppose $A[1..n]$ is the input string. For all indices i and k, let $\boldsymbol{LBS(i,k)}$ denote the length of the longest balanced subsequence of the substring $A[i..k]$. We need to compute $LBS(1,n)$. This function obeys the following recurrence:

$$LBS(i,k) = \begin{cases} 0 & \text{if } i \geq k \\ \max\left\{ \begin{array}{c} 2 + LBS(i+1,k-1) \\ \max\limits_{j=1}^{k-1} \big(LBS(i,j) + LBS(j+1,k)\big) \end{array} \right\} & \text{if } A[i] \sim A[k] \\ \max\limits_{j=1}^{k-1} \big(LBS(i,j) + LBS(j+1,k)\big) & \text{otherwise} \end{cases}$$

Here $\boldsymbol{A[i] \sim A[k]}$ indicates that $A[i]$ is a left delimiter and $A[k]$ is the corresponding right delimiter: Either $A[i] =$ `(` and $A[k] =$ `)`, or $A[i] =$ `[` and $A[k] =$ `]`.

We can memoize this function into a two-dimensional array $LBS[1..n, 1..n]$. Because each entry $LBS[i,k]$ depends only on entries in later rows or earlier columns (or both), we can fill this array row-by-row from bottom up (decreasing i) in the outer loop, scanning each row from left to right (increasing k) in the inner loop.

We can compute each entry $LBS[i,k]$ in $O(n)$ time, so the resulting algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ■

Solution (pseudocode): The following algorithm runs in $\boldsymbol{O(n^3)}$ ***time***:

```
LongestBalancedSubsequence(A[1..n]):
  for i ← n down to 1
      LBS[i,i] ← 0
      for k ← i+1 to n
          if (A[i] = ( and A[k] = )) or (A[i] = [ and A[k] = ])
              LBS[i,k] ← LBS[i+1,k-1] + 2
          else
              LBS[i,k] ← 0
          for j ← i to k-1
              LBS[i,k] ← max{LBS[i,k], LBS[i,j] + LBS[j+1,k]}
  return LBS[1,n]
```

Here $LBS[i, k[$ stores the length of the longest balanced subsequence of the substring $A[i..k]$. ■

Rubric: 10 points, standard dynamic programming rubric. Yes, each of these solutions is independently worth full credit.

4. Oh, no! You've just been appointed as the new organizer of Giggle, Inc.'s annual mandatory holiday party! The employees at Giggle are organized into a strict hierarchy, that is, a tree with the company president at the root. The all-knowing oracles in Human Resources have assigned a real number to each employee measuring how "fun" the employee is. In order to keep things social, there is one restriction on the guest list: An employee cannot attend the party if their immediate supervisor is also present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it's her company, after all.

Describe an algorithm that makes a guest list for the party that maximizes the sum of the "fun" ratings of the guests. The input to your algorithm is a rooted tree T describing the company hierarchy, where each node v has a field $v.fun$ storing the "fun" rating of the corresponding employee.

Solution (two functions): We define two functions over the nodes of T.

- $MaxFunYes(v)$ is the maximum total "fun" of a legal party among the descendants of v, where v is definitely invited.
- $MaxFunNo(v)$ is the maximum total "fun" of a legal party among the descendants of v, where v is definitely not invited.

We need to compute $MaxFunYes(root)$. These two functions obey the following mutual recurrences:

$$MaxFunYes(v) = v.fun + \sum_{\text{children } w \text{ of } v} MaxFunNo(w)$$

$$MaxFunNo(v) = \sum_{\text{children } w \text{ of } v} \max\{MaxFunYes(w), MaxFunNo(w)\}$$

These recurrences do not require separate base cases, because $\sum \varnothing = 0$.[a]

We can memoize these functions by adding two additional fields $v.yes$ and $v.no$ to each node v in the tree. The values at each node depend only on the vales at its children, so we can compute all $2n$ values using a postorder traversal of T.

The resulting algorithm spends $O(1)$ time at each node of T, and therefore runs in ***O(n) time***. ■

[a] A naïve recursive implementation of these recurrences would run in $O(\phi^n)$ time in the worst case, where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio. The worst case occurs when T is a single path.

Solution (two functions, pseudocode): The following algorithm runs in ***O(n) time***.

```
BestParty(T):
  ComputeMaxFun(T.root)
  return T.root.yes
```

```
ComputeMaxFun(v):
  v.yes ← v.fun
  v.no ← 0
  for all children w of v
      ComputeMaxFun(w)
      v.yes ← v.yes + w.no
      v.no ← v.no + max{w.yes, w.no}
```

We are storing two pieces of information in each node v of the tree:

- $v.yes$ is the maximum total "fun" of a legal party among the descendants of v, assuming v is invited.
- $v.no$ is the maximum total "fun" of a legal party among the descendants of v, assuming v is not invited.

(Yes, this is still dynamic programming; we're only traversing the tree recursively in ComputeMaxFun because that's the most natural way to traverse trees!) ■

Solution (one function): For each node v in the input tree T, let $MaxFun(v)$ denote the maximum total "fun" of a legal party among the descendants of v, where v may or may not be invited.

The president of the company must be invited, so none of the president's "children" in T can be invited. Thus, the value we need to compute is

$$root.fun + \sum_{\text{grandchildren } w \text{ of } root} MaxFun(w).$$

The function $MaxFun$ obeys the following recurrence:

$$MaxFun(v) = \max \left\{ \begin{array}{l} v.fun + \displaystyle\sum_{\text{grandchildren } x \text{ of } v} MaxFun(x) \\ \displaystyle\sum_{\text{children } w \text{ of } v} MaxFun(w) \end{array} \right\}$$

(This recurrence does not require a separate base case, because $\sum \varnothing = 0$.) We can memoize this function by adding an additional field $v.maxFun$ to each node v in the tree. The value at each node depends only on the values at its children and grandchildren, so we can compute all values using a postorder traversal of T.

The algorithm spends $O(1)$ time at each node (because each node has exactly one parent and one grandparent) and therefore runs in $\boldsymbol{O(n)}$ ***time*** altogether. ■

Solution (one function, pseudocode):

```
BestParty(T):
  ComputeMaxFun(T.root)
  party ← T.root.fun
  for all children w of T.root
      for all children x of w
          party ← party + x.maxFun
  return party
```

```
ComputeMaxFun(v):
  yes ← v.fun
  no ← 0
  for all children w of v
      ComputeMaxFun(w)
      no ← no + w.maxFun
      for all children x of w
          yes ← yes + x.maxFun
  v.maxFun ← max{yes, no}
```

Here $v.maxFun$ stores the maximum total "fun" of a legal party among the descendants of v, where v may or may not be invited.

Each value $v.maxFun$ is read at most three times during the algorithm's execution: Once in ComputeMaxFun($v.parent$), and once in ComputeMaxFun($v.parent.parent$), and at most once in the non-recursive part of BestParty. Thus, the entire algorithm runs in $O(n)$ *time*. ■

Rubric: 10 points: standard dynamic programming rubric. These are not the only correct solutions. Yes, each of these solutions is independently worth full credit.

CS/ECE 374 A ✧ Fall 2023

Homework 8

Due Tuesday, October 24, 2023 at 9pm Central Time

---

1. A *six-sided die* (plural *dice*) is a cube with each side marked with a different number of dots (called *pips*) from 1 to 6. On a *standard* die, numbers on opposite sides always add up to 7.

   A *rolling die maze* is a puzzle involving a standard six-sided die and a grid of squares. You should imagine the grid lying on a table; the die always rests on and exactly covers one square of the grid. In a single step, you can *roll* the die 90 degrees around one of its bottom edges, moving it to an adjacent square one step north, south, east, or west.

   Some squares in the grid may be *blocked*; the die can never rest on a blocked square. Other squares may be *labeled* with a number; whenever the die rests on a labeled square, the number on the *top* face of the die must equal the label. Squares that are neither labeled nor marked are called *free*. You may not roll the die off the edges of the grid. A rolling die maze is *solvable* if it is possible to place a die on the lower left square and roll it to the upper right square under these constraints.

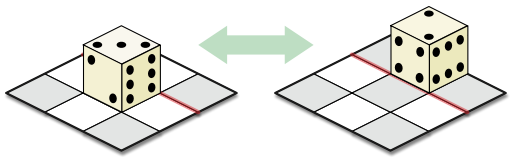


Figure 1. Rolling a (right-handed) die

   Figure 2 shows five rolling die mazes. The first two mazes are solvable using any standard die. Specifically, the first maze can be solved by placing the die on the lower left square with 1 on the top face, and then rolling the die east, north, north, east; the second maze can be solved in 12 moves. The third maze is only solvable using a *right-handed* die, where faces 1, 2, 3 appear in counterclockwise order around a common corner.[1] The last two mazes cannot be solved even with non-standard dice.

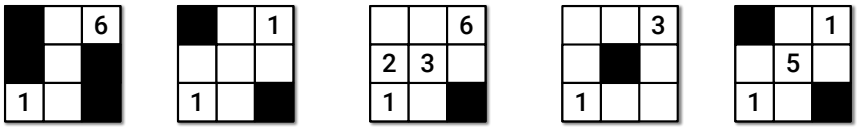


Figure 2. Five rolling die mazes.

   Describe and analyze an algorithm that determines whether a given rolling die maze can be solved with a right-handed standard die. Your input is a two-dimensional array $Label[1..n, 1..n]$, where each entry $Label[i, j]$ stores the label of the square in the ith row and jth column, where 0 means the square is free and -1 means the square is blocked.

   [Hint: You have some freedom in how to place the initial die. There are rolling die mazes that can be solved only if the initial placement is chosen correctly. Describe your solution in high-level language; don't get bogged down in grungy case analysis.]

[1] Right-handed dice are more common in the Western hemisphere; left-handed dice are more common in east Asia.

2. The Cheery Hells neighborhood of Sham-Poobanana runs a popular and well-regulated Halloween celebration, attended by thousands of costumed children from all across Poobanana County. To regulate the flood of costumed children, the Cheery Hells Neighborhood Association has designated a walking direction for each stretch of sidewalk.

   After paying the $25 entrance fee, each child receives a map of the neighborhood, in the form of a directed graph G, whose vertices represent houses. Each edge $v \to w$ indicates that one can walk directly from house v to house w following the designated sidewalk directions. (Anyone caught walking backward along a sidewalk will be ejected from Cheery Hells, without their candy. No refunds.) A special vertex s designates the entrance to Cheery Hells. Children can visit houses as many times as they like, but biometric scanners at every house ensure that each child receives candy only at their *first* visit to each house.

   The children of Cheery Hells have published a secret web site listing the amount of candy that each house in Cheery Hells will give to each visitor.

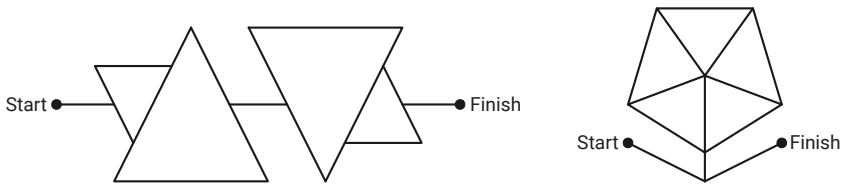
   Describe and analyze an algorithm to compute the maximum amount of candy that a single child can obtain in a walk through Cheery Hells, starting at the entrance node s. The input to your algorithm is the directed graph G, along with a non-negative integer $v.candy$ for each vertex v, describing the amount of candy the correspinding house gives to each first-time visitor.

   [Hint: Think about two special cases first: (1) Cheery Hells is strongly connected, and (2) Cheery Hells is acyclic. Solving only these two special cases is worth half credit.]

3. ***Practice only. Do not submit solutions.***

   One of my daughter's elementary-school math workbooks[2] contains several puzzles of the following type:

   > Complete each angle maze below by tracing a path from Start to Finish that has only acute angles.



   Describe and analyze an algorithm to solve arbitrary acute-angle mazes.

   Your input is a connected undirected graph G, whose vertices are points in the plane and whose edges are straight line segments. Edges do not intersect, except at their common endpoints. For example, a drawing of the letter X would have five vertices and four edges, and the first maze above has 18 vertices and 21 edges. You are also given two vertices Start and Finish.

   Your algorithm should return TRUE if G contains a walk from Start to Finish that has only acute angles, and FALSE otherwise. Formally, a walk through G is valid if, for any two

[2] Jason Batterson and Shannon Rogers, *Beast Academy Math: Practice 3A*, 2012. See https://www.beastacademy.com/resources/printables.php for several more examples.

consecutive edges $u \to v \to w$ in the walk, either $\angle uvw = \pi$ (straight) or $0 < \angle uvw < \pi/2$ (acute). Assume you have a subroutine that can determine in $O(1)$ time whether the angle between two given segments is straight, obtuse, right, or acute.

Solved problem

4. Professor McClane takes you out to a lake and hands you three empty jars. Each jar holds a positive integer number of gallons; the capacities of the three jars may or may not be different. The professor then demands that you put exactly k gallons of water into one of the jars (which one doesn't matter), for some integer k, using only the following operations:

   (a) Fill a jar with water from the lake until the jar is full.

   (b) Empty a jar of water by pouring water into the lake.

   (c) Pour water from one jar to another, until either the first jar is empty or the second jar is full, whichever happens first.

   For example, suppose your jars hold 6, 10, and 15 gallons. Then you can put 13 gallons of water into the third jar in six steps:

   - Fill the third jar from the lake.
   - Fill the first jar from the third jar. (Now the third jar holds 9 gallons.)
   - Empty the first jar into the lake.
   - Fill the second jar from the lake.
   - Fill the first jar from the second jar. (Now the second jar holds 4 gallons.)
   - Empty the second jar into the third jar.

   Describe and analyze an efficient algorithm that either finds the smallest number of operations that leave exactly k gallons in any jar, or reports correctly that obtaining exactly k gallons of water is impossible. Your input consists of the capacities of the three jars and the positive integer k. For example, given the four numbers 6, 10, 15, and 13 as input, your algorithm should return the number 6 (the length of the sequence of operations listed above).

   Solution: Let A, B, C denote the capacities of the three jars. We reduce the problem to breadth-first search in a directed graph $G = (V, E)$ defined as follows:

   - $V = \{(a, b, c) \mid 0 \le a \le A \text{ and } 0 \le b \le B \text{ and } 0 \le c \le C\}$. Each vertex corresponds to a possible ***configuration*** of water in the three jars. There are $(A+1)(B+1)(C+1) = O(ABC)$ vertices altogether.
   - G contains a directed edge $(a, b, c) \to (a', b'c')$ whenever it is possible to change the first configuration into the second in one step. Specifically, G contains an edge from (a, b, c) to each of the following vertices (except those already equal to (a, b, c)):
     - $(0, b, c)$ and $(a, 0, c)$ and $(a, b, 0)$ — dumping a jar into the lake
     - (A, b, c) and (a, B, c) and (a, b, C) — filling a jar from the lake

- $\begin{cases} (0, a+b, c) & \text{if } a+b \le B \\ (a+b-B, B, c) & \text{if } a+b \ge B \end{cases}$ — pouring from jar 1 into jar 2
- $\begin{cases} (0, b, a+c) & \text{if } a+c \le C \\ (a+c-C, b, C) & \text{if } a+c \ge C \end{cases}$ — pouring from jar 1 into jar 3
- $\begin{cases} (a+b, 0, c) & \text{if } a+b \le A \\ (A, a+b-A, c) & \text{if } a+b \ge A \end{cases}$ — pouring from jar 2 into jar 1
- $\begin{cases} (a, 0, b+c) & \text{if } b+c \le C \\ (a, b+c-C, C) & \text{if } b+c \ge C \end{cases}$ — pouring from jar 2 into jar 3
- $\begin{cases} (a+c, b, 0) & \text{if } a+c \le A \\ (A, b, a+c-A) & \text{if } a+c \ge A \end{cases}$ — pouring from jar 3 into jar 1
- $\begin{cases} (a, b+c, 0) & \text{if } b+c \le B \\ (a, B, b+c-B) & \text{if } b+c \ge B \end{cases}$ — pouring from jar 3 into jar 2

Because each vertex has at most 12 outgoing edges, there are at most $12(A+1) \times (B+1)(C+1) = O(ABC)$ edges altogether.

To solve the jars problem, we need to find the ***shortest path*** in G from the start vertex $(0,0,0)$ to any target vertex of the form $(k,\cdot,\cdot)$ or $(\cdot,k,\cdot)$ or $(\cdot,\cdot,k)$.

We can compute this shortest path by calling ***breadth-first search*** starting at $(0,0,0)$, and then examining every target vertex by brute force. If BFS does not visit any target vertex, we report that no legal sequence of moves exists. Otherwise, we find the target vertex closest to $(0,0,0)$ and trace its parent pointers back to $(0,0,0)$ to determine the shortest sequence of moves. The resulting algorithm runs in $O(V+E) = \boldsymbol{O(ABC)}$ ***time***.

We can speed up this algorithm by observing that every move leaves at least one jar either completely empty or completely full. Thus, we only need vertices (a,b,c) where either $a=0$ or $b=0$ or $c=0$ or $a=A$ or $b=B$ or $c=C$; no other vertices are reachable from $(0,0,0)$. The number of non-redundant vertices and edges is $O(AB+BC+AC)$. Thus, if we only construct and search the relevant portion of G, the algorithm runs in $\boldsymbol{O(AB+BC+AC)}$ ***time***. ■

Rubric: 10 points: standard graph reduction rubric

- Brute force construction is fine.

— 1 for calling Dijkstra instead of BFS

- max 8 points for $O(ABC)$ time; scale partial credit.

CS/ECE 374 A Fall 2023

Homework 9

Due Tuesday, October 31, 2023 at 9pm

This is the last homework before Midterm 2.

---

1. You are planning a hiking trip in Jasper National Park in British Columbia over winter break. You have a complete map of the park's trails, which indicates that hikers on certain trails have a higher chance of encountering a sasquatch. All visitors to the park are required to purchase a canister of sasquatch repellent. You can safely traverse a high-risk trail segment only by *completely* using up a *full* canister. The park rangers have helpfully installed several refilling stations around the park, where you can refill empty canisters at no cost. The canisters themselves are expensive and heavy, so you can only carry one. The trails are narrow, so each trail segment allows traffic in only one direction.

   You have converted the trail map into a directed graph $G = (V, E)$, whose vertices represent trail intersections, and whose edges represent trail segments. A subset $R \subseteq V$ of the vertices indicate the locations of the *R*epellent *R*efilling stations, and a subset $H \subseteq E$ of the edges are marked as *H*igh-risk. Each edge e is labeled with the length $\ell(e)$ of the corresponding trail segment. Your campsite appears on the map as a vertex $s \in V$, and the visitor center is another vertex $t \in V$.

   (a) Describe and analyze an algorithm that finds the shortest *safe* hike from your campsite s to the visitor center t. Assume there is a refill station at your campsite, and another refill station at the visitor center.

   (b) Describe and analyze an algorithm to decide if you can safely hike from *any* refill station to *any* other refill station. In other words, for *every* pair of vertices u and v in R, is there a safe hike from u to v?

2. You are driving through the back-country roads of Tenkucky, desperately trying to leave the state before the state's annual Halloween Purge begins. Every road in the state is patrolled by a Driving Posse who will let you exercise your god-given right to drive as fast as you damn well please, provided you pay the appropriate speed tax. The faster you traverse any road, the more you have to pay. What's the fastest way to escape the state?

   You have an accurate map of the state, in the form of a directed graph $G = (V, E)$, whose vertices V represent small towns and whose edges E represent one-lane dirt roads between towns.[1] One vertex s is marked as your starting location; a subset $X \subset V$ of vertices are marked as exits. Each edge e has an associated value $\$(e)$ with the following interpretation.

   - If you drive from one end of road e to the other in m minutes, for any positive real number m, then you must pay road e's Driving Posse a speed tax of $\lceil \$(e)/m \rceil$ dollars.

[1] Paved roads are far too expensive!

- Equivalently, if you pay road e's Driving Posse a speed tax of d dollars, for any positive integer d, you are allowed to drive the entire length of road e in $\$(e)/d$ minutes, but no less.

In particular, any road you drive on *at all* will cost you *at least* one dollar. Anyone who violates this rule (for example, by running out of money) will be thrown in jail, which means almost certain death in the Purge.

The Driving Posses do not accept coins, credit cards, Venmo, Zelle, or any other mobile payment app—only cold hard American paper currency—and they do not give change. Fortunately, you are starting your journey with a pile of D crisp new $1 bills.

Describe and analyze and algorithm to compute the fastest possible driving route from s to any exit node in X. The input to your algorithm consists of the map $G = (V, E)$, the start vertex s, the exit vertices X, and the positive integer D. Report the running time of your algorithm as a function of the parameters V, E, and D.

3. ***Practice only. Do not submit solutions.***

After a grueling midterm at the See-Bull Center for Commuter Silence, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Sham-Poobanana. Unfortunately, no single bus visits both the See-Bull Center and your home; you must change buses at least once. There are exactly b different buses. Each bus starts at 12:00:01AM, makes exactly n stops, and finally stops running at 11:59:59PM. Buses always run exactly on schedule, and you have an accurate watch. Finally, you are far too tired to walk between bus stops.

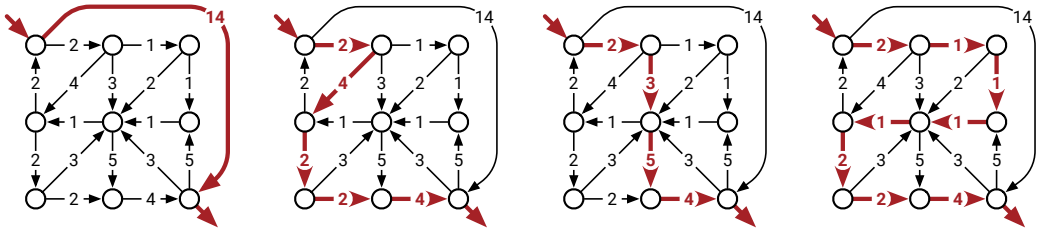
(a) Describe and analyze an algorithm to determine a sequence of bus rides that gets you home as early as possible. Your goal is to minimize your *arrival time*, not the time you spend traveling.

(b) Oh, no! The midterm was held on Halloween, and the streets are infested with zombies! Describe how to modify your algorithm from part (a) to minimize *the total time you spend waiting at bus stops*; you don't care how late you get home or how much time you spend on buses. (Assume you can wait inside the See-Bull Center until your first bus is just about to leave.)

For both questions, your input consists of the exact time when the midterm ends See-Bull and two arrays $Time[1..b, 1..n]$ and $Stop[1..b, 1..n]$, where $Time[i, j]$ is the scheduled time of the ith bus's jth stop, and $Stop[i, j]$ is the location of that stop. Report the running times of your algorithms as functions of the parameters n and b.

Solved Problems

4. Although we typically speak of "the" shortest path from one vertex to another, a single graph could contain several minimum-length paths with the same endpoints.



Four (of many) equal-length shortest paths.

Describe and analyze an algorithm to compute the *number* of shortest paths from a source vertex s to a target vertex t in an arbitrary directed graph G with weighted edges. Assume that all edge weights are positive and that any necessary arithmetic operations can be performed in $O(1)$ time each.

[Hint: Compute shortest path distances from s to every other vertex. Throw away all edges that cannot be part of a shortest path from s to another vertex. What's left?]

Solution: We start by computing shortest-path distances $dist(v)$ from s to v, for every vertex v, using Dijkstra's algorithm. Call an edge $u \to v$ ***tight*** if $dist(u) + w(u \to v) = dist(v)$. Every edge in a shortest path from s to t must be tight. Conversely, every path from s to t that uses only tight edges has total length $dist(t)$ and is therefore a shortest path!

Let H be the subgraph of all tight edges in G. We can easily construct H in $O(V+E)$ time. Because all edge weights are positive, H is a directed acyclic graph. It remains only to count the number of paths from s to t in H.

For any vertex v, let $NumPaths(v)$ denote the number of paths in H from v to t; we need to compute $NumPaths(s)$. This function satisfies the following simple recurrence:

$$NumPaths(v) = \begin{cases} 1 & \text{if } v = t \\ \displaystyle\sum_{v \to w} NumPaths(w) & \text{otherwise} \end{cases}$$

In particular, if v is a sink but $v \neq t$ (and thus there are no paths from v to t), this recurrence correctly gives us $NumPaths(v) = \sum \varnothing = 0$.

We can memoize this function into the graph itself, storing each value $NumPaths(v)$ at the corresponding vertex v. Since each subproblem depends only on its successors in H, we can compute $NumPaths(v)$ for all vertices v by considering the vertices in reverse topological order, or equivalently, by performing a depth-first search of H starting at s. The resulting algorithm runs in $O(V+E)$ time.

The overall running time of the algorithm is dominated by Dijkstra's algorithm in the preprocessing phase, which runs in $\boldsymbol{O(E \log V)}$ ***time***. ■

Rubric: 10 points = 5 points for reduction to counting paths in a dag (standard graph reduction rubric) + 5 points for the path-counting algorithm (standard dynamic programming rubric)

5. After moving to a new city, you decide to choose a walking route from your home to your new office. Your route must consist of an uphill path (for exercise) followed by a downhill path (to cool down), or just an uphill path, or just a downhill path. But you also want the *shortest* path that satisfies these conditions, so that you actually get to work on time.

   Your input consists of an undirected graph G, whose vertices represent intersections and whose edges represent road segments, along with a start vertex s and a target vertex t. Every vertex v has a value $h(v)$, which is the height of that intersection above sea level, and each edge uv has a value $\ell(uv)$, which is the length of that road segment.

   (a) Describe and analyze an algorithm to find the shortest uphill–downhill walk from s to t. Assume all vertex heights are distinct.

Solution: We define a new directed graph $G' = (V', E')$ as follows:

- $V' = \{v^\uparrow, v^\downarrow \mid V \in V\}$. Vertex $v^\uparrow$ indicates that we are at intersection v moving uphill, and vertex $v^\downarrow$ indicates that we are at intersection v moving downhill.
- E' is the union of three sets:
  - Uphill edges: $\{u^\uparrow \to v^\uparrow \mid uv \in E \text{ and } h(u) < h(v)\}$. Each uphill edge $u^\uparrow \to v^\uparrow$ has weight $\ell(uv)$.
  - Downhill edges: $\{u^\downarrow \to v^\downarrow \mid uv \in E \text{ and } h(u) > h(v)\}$. Each downhill edge $u^\downarrow \to v^\downarrow$ has weight $\ell(uv)$.
  - Switch edges: $\{v^\uparrow \to v^\downarrow \mid v \in V\}$; each switch edge has weight 0.

We need to compute three shortest paths in this graph:

- The shortest path from $s^\uparrow$ to $t^\downarrow$ gives us the best uphill-then-downhill route.
- The shortest path from $s^\uparrow$ to $t^\uparrow$ gives us the best uphill-only route.
- The shortest path from $s^\downarrow$ to $t^\downarrow$ gives us the best downhill-only route.

G' is a directed ***acyclic*** graph; we can get a topological ordering by listing the up vertices $v^\uparrow$, sorted by increasing height, followed by the down vertices $v^\downarrow$, sorted by decreasing height. Thus, we can compute the shortest path in G' from any vertex to any other in $O(V' + E') = O(V + E)$ time by dynamic programming. (The algorithm is the same as the longest-path algorithm in the notes, except we use "min" instead of "max" in the recurrence, and define $\min \emptyset = \infty$.)

Our overall algorithm runs in $\boldsymbol{O(V + E)}$ ***time***. ■

Rubric: 5 points = 1 for vertices + 1 for edges + 1 for arguing G' is a dag + 1 for algorithm + 1 for running time. This is not the only correct solution. Max 4 points for a correct reduction to Dijkstra's algorithm that runs in $O(E \log V)$ time.

(b) Suppose you discover that there is no path from s to t with the structure you want. Describe an algorithm to find a path from s to t that alternates between "uphill" and "downhill" subpaths as few times as possible, and has minimum length among all such paths. (There may be even shorter paths with more alternations, but you don't care about them.) Again, assume all vertex heights are distinct.

Solution (Dijkstra, 5/5): Let $L = 1 + \sum_{u \to v} \ell(u \to v)$. Define a new graph $G' = (V', E')$ as follows:

- $V' = \{v^\uparrow, v^\downarrow \mid V \in V\} \cup \{s, t\}$. Vertex $v^\uparrow$ indicates that we are at intersection v moving uphill, and vertex $v^\downarrow$ indicates that we are at intersection v moving downhill.
- E' contains four types of edges:
  - Uphill edges: $\{u^\uparrow \to v^\uparrow \mid uv \in E \text{ and } h(u) < h(v)\}$. Each uphill edge $u^\uparrow \to v^\uparrow$ has weight $\ell(uv)$.
  - Downhill edges: $\{u^\downarrow \to v^\downarrow \mid uv \in E \text{ and } h(u) > h(v)\}$. Each downhill edge $u^\downarrow \to v^\downarrow$ has weight $\ell(uv)$.
  - Switch edges: $\{v^\uparrow \to v^\downarrow \mid v \in V\} \cup \{v^\downarrow \to v^\uparrow \mid v \in V\}$. Each switch edge has weight L.
  - Start and end edges $s \to s^\uparrow$, $s \to s^\downarrow$, $t^\uparrow \to t$, and $t^\downarrow \to t$, each with weight 0,

We need to compute the shortest path from s to t in G'; the large weight L on the switch edges guarantees that this path with have the minimum number of switches, and the minimum length among all paths with that number of switches. Dijkstra's algorithm finds this shortest path in $O(E' \log V') = \boldsymbol{O(E \log V)}$ ***time***.

(Because G' includes switch edges in both directions, G' is not a dag, so we can't use dynamic programming directly.) ■

Rubric: 5 points, standard graph-reduction rubric. This is not the only correct solution with running time $O(E \log V)$.

Solution (clever, extra credit): Our algorithm works in two phases: First we determine the minimum number of switches required to reach every vertex, and then we compute the shortest path from s to t with the minimum number of switches. The first phase is can be solved in $O(V + E)$ time by a modification of breadth-first search; the second by computing shortest paths in a dag.

For the first phase, we define a new graph $G' = (V', E')$ as follows:

- $V' = \{v^\uparrow, v^\downarrow \mid V \in V\} \cup \{s, t\}$. Vertex $v^\uparrow$ indicates that we are at intersection v moving uphill, and vertex $v^\downarrow$ indicates that we are at intersection v moving downhill.
- E' contains four types of edges:
  - Uphill edges: $\{u^\uparrow \to v^\uparrow \mid uv \in E \text{ and } h(u) < h(v)\}$. Each uphill edge has weight 0.
  - Downhill edges: $\{u^\downarrow \to v^\downarrow \mid uv \in E \text{ and } h(u) > h(v)\}$. Each downhill

edge has weight 0.

- Switch edges: $\{v^{\uparrow}\to v^{\downarrow} \mid v \in V\} \cup \{v^{\downarrow}\to v^{\uparrow} \mid v \in V\}$. Each switch edge has weight 1.
- Start and end edges $s\to s^{\uparrow}$, $s\to s^{\downarrow}$, $t^{\uparrow}\to t$, and $t^{\downarrow}\to t$, each with weight 0.

Now we compute the shortest path distance from s to every other vertex in G'. We could use Dijkstra's algorithm in $O(E \log V)$ time, but the structure of the graph supports a faster algorithm.

Intuitively, we break the shortest-path computation into phases, where in the kth phase, we mark all vertices at distance k from the source vertex s. During the kth phase, we may also discover vertices at distance $k+1$, but no further. So instead of using a binary heap for the priority queue, it suffices to use two bags: one for vertices at distance k, and one for vertices at distance $k+1$.

```
ZeroOneDijkstra(G, ℓ, s):
  s.dist ← 0
  for all vertices v ≠ s
      v.dist ← ∞
  curr ← new empty bag
  add s to curr
  for k ← 0 to V
      next ← new empty bag
      while curr is not empty
          take v from curr            ⟨⟨v.dist = k⟩⟩
          for all edges v→w
              if w.dist > v.dist + ℓ(v→w)
                  w.dist ← v.dist + ℓ(v→w)
                  if ℓ(v→w) = 0
                      add w to curr
                  else ⟨⟨if ℓ(v→w) = 1⟩⟩
                      add w to next
      curr ← next
```

This phase of the algorithm runs in $O(V' + E') = O(V + E)$ time.

Once we have computed distances in G', we construct a second graph $G'' = (V', E'')$ with the same vertices as G', but only a subset of the edges:

$$E'' = \left\{u'\to v' \in E' \;\middle|\; u'.dist + \ell(u'\to v') = v'.dist\right\}$$

Equivalently, an edge $u'\to v'$ belongs to E'' if and only if that edge is part of at least one shortest path in G' from s to another vertex. It follows (by induction, of course), that every path in G'' from s to another vertex v' is a shortest path in G', and therefore a minimum-switch path in G.

We also reassign the edge weights in G''. Specifically, we assign each uphill edge $u^{\uparrow}\to v^{\uparrow}$ and downhill edge $u^{\downarrow}\to v^{\downarrow}$ in G'' weight $\ell(uv)$, and we assign every switch edge, start edge, and end edge weight 0. **Now we need to compute the shortest path from s to t in G'', with respect to these new edge weights.**

We can expand the definition of E'' in terms of the original input graph as follows:

$$\begin{aligned} E'' = &\left\{u^{\uparrow} \to v^{\uparrow} \;\middle|\; uv \in E \text{ and } h(u) < h(v) \text{ and } u^{\uparrow}.dist = v^{\uparrow}.dist\right\} \\ &\cup \left\{u^{\downarrow} \to v^{\downarrow} \;\middle|\; uv \in E \text{ and } h(u) > h(v) \text{ and } u^{\downarrow}.dist = v^{\downarrow}.dist\right\} \\ &\cup \left\{v^{\uparrow} \to v^{\downarrow} \;\middle|\; v \in V \text{ and } v^{\uparrow}.dist < v^{\downarrow}.dist\right\} \\ &\cup \left\{v^{\downarrow} \to v^{\uparrow} \;\middle|\; v \in V \text{ and } v^{\downarrow}.dist < v^{\uparrow}.dist\right\} \end{aligned}$$

We can topologically sort G'' by first sorting the vertices by increasing $v'.dist$, and then within each subset of vertices with equal $v'.dist$, listing the up-vertices by increasing height, followed by the down vertices by decreasing height. It follows that G'' is a dag! Thus, we can compute shortest paths in G'' in $O(V'' + E'') = O(V + E)$ time, using the same dynamic programming algorithm that we used in part (a).

The overall algorithm runs in $\boldsymbol{O(V+E)}$ ***time***. ■

Rubric: max 10 points =

- 5 for computing minimum-switch paths = 1 for vertices + 1 for edges (including weights) + 2 for 0/1 shortest path algorithm + 1 for running time.
- 5 for computing shortest minimum-switch paths = 1 for vertices + 1 for edges (including weights) + 1 for proving dag + 1 for dynamic programming algorithm + 1 for running time

CS/ECE 374 A ✧ Fall 2023

Homework 10

Due Tuesday, November 14, 2019 at 9pm

---

1. This problem asks you to describe polynomial-time reductions between two closely related problems:

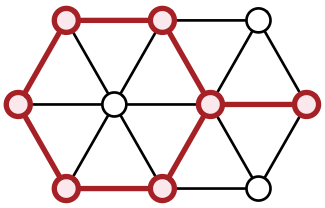
   - SUBSETSUM: Given a set S of positive integers and a target integer T, is there a subset of S whose sum is T?
   - PARTITION: Given a set S of positive integers, is there a way to partition S into two subsets S_1 and S_2 that have the same sum?

   (a) Describe a polynomial-time reduction from SUBSETSUM to PARTITION.

   (b) Describe a polynomial-time reduction from PARTITION to SUBSETSUM.

   Don't forget to to prove that your reductions are correct.

2. A subset S of vertices in an undirected graph G is called *triangle-free* if, for every triple of vertices $u, v, w \in S$, at least one of the three edges uv, uw, vw is *absent* from G. Prove that finding the size of the largest triangle-free subset of vertices in a given undirected graph is NP-hard.



A triangle-free subset of 7 vertices and its induced edges.
This is ***not*** the largest triangle-free subset in this graph.

Solved Problem

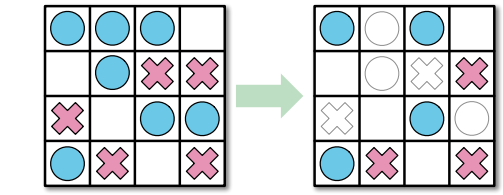
4. ***RedBlue*** is a puzzle that consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions:

   (1) Every row contains at least one stone.

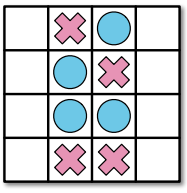
   (2) No column contains stones of both colors.

   For some RedBlue puzzles, reaching this goal is impossible; see the example below.

   Prove that it is NP-hard to determine whether a given RedBlue puzzle has a solution.



A solvable RedBlue puzzle and one of its many solutions.



An unsolvable RedBlue puzzle.

Solution: We show that RedBlue is NP-hard by describing a reduction from 3SAT.

Let Φ be a 3CNF boolean formula with m variables and n clauses. We transform this formula into a RedBlue instance X in polynomial time as follows. The size of the board is $n \times m$. The stones are placed as follows, for all indices i and j:

- If the variable x_j appears in the ith clause of Φ, we place a blue stone at (i, j).
- If the negated variable $\overline{x_j}$ appears in the ith clause of Φ, we place a red stone at (i, j).
- Otherwise, we leave cell (i, j) blank.

To prove that RedBlue is NP-hard, it suffices to prove the following claim:

Φ is satisfiable if and only if RedBlue puzzle X is solvable.

$\Longrightarrow$ First, suppose Φ is satisfiable; consider an arbitrary satisfying assignment. For each index j, remove stones from column j according to the value assigned to x_j:

- If $x_j = \text{TRUE}$, remove all red stones from column j.
- If $x_j = \text{FALSE}$, remove all blue stones from column j.

In other words, remove precisely the stones that correspond to FALSE literals. Because every variable appears in at least one clause, each column now contains stones of only one color (if any). On the other hand, each clause of Φ must contain at least one TRUE literal, and thus each row still contains at least one stone. We conclude that RedBlue puzzle X is solvable.

⟸ On the other hand, suppose RedBlue puzzle X is solvable; consider an arbitrary solution. For each index j, assign a value to x_j depending on the colors of stones left in column j:

- If column j contains blue stones, set $x_j = \text{TRUE}$.
- If column j contains red stones, set $x_j = \text{FALSE}$.
- If column j is empty, set x_j arbitrarily.

In other words, assign values to the variables so that the literals corresponding to the remaining stones are all TRUE. Each row still has at least one stone, so each clause of Φ contains at least one TRUE literal, so this assignment makes $\Phi = \text{TRUE}$. We conclude that Φ is satisfiable.

This reduction clearly requires only polynomial time. ■

Standard NP-hardness rubric. 10 points =

+ 1 point for choosing a reasonable NP-hard problem X to reduce from.
  - The Cook-Levin theorem implies that *in principle* one can prove NP-hardness by reduction from *any* NP-complete problem. What we're looking for here is a problem where a simple and direct NP-hardness proof seems likely.
  - You can use any of the NP-hard problems listed on the next page or in the textbook (except the one you are trying to prove NP-hard, of course).
+ 2 points for a *structurally sound* polynomial-time reduction. Specifically, the reduction must:
  - take an *arbitrary* instance of the declared problem X ***and nothing else*** as input,
  - transform that input into a corresponding instance of Y (the problem we're trying to prove NP-hard),
  - transform the output of the magic algorithm for Y into a reasonable output for X, and
  - run in polynomial time.

  (The output transformation is usually trivial.) This is strictly about the structure of the reduction algorithm, not about its correctness. **No credit for the rest of the problem if this is wrong.**
+ 2 points for a *correct* polynomial-time reduction. That is, assuming a black-box algorithm that solves Y in polynomial time, the proposed reduction actually solves problem X in polynomial time.
+ 2 points for the "if" proof of correctness. (Every good instance of X is transformed into a good instance of Y.)
+ 2 points for the "only if" proof of correctness. (Every bad instance of X is transformed into a bad instance of Y.)
+ 1 point for writing "polynomial time"

- An incorrect but structurally sound polynomial-time reduction that still satisfies half of the correctness proof is worth at most 6/10.
- A reduction in the wrong direction is worth at most 1/10.

Some useful NP-hard problems. You are welcome to use any of these in your own NP-hardness proofs, except of course for the specific problem you are trying to prove NP-hard.

CircuitSat: Given a boolean circuit, are there any input values that make the circuit output True?

3Sat: Given a boolean formula in conjunctive normal form, with exactly three distinct literals per clause, does the formula have a satisfying assignment?

MaxIndependentSet: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MaxClique: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MinVertexCover: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

MinSetCover: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subcollection whose union is S?

MinHittingSet: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subset of S that intersects every subset S_i?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

ChromaticNumber: Given an undirected graph G, what is the minimum number of colors required to color its vertices, so that every edge touches vertices with two different colors?

HamiltonianPath: Given graph G (either directed or undirected), is there a path in G that visits every vertex exactly once?

HamiltonianCycle: Given a graph G (either directed or undirected), is there a cycle in G that visits every vertex exactly once?

TravelingSalesman: Given a graph G (either directed or undirected) with weighted edges, what is the minimum total weight of any Hamiltonian path/cycle in G?

LongestPath: Given a graph G (either directed or undirected, possibly with weighted edges), what is the length of the longest simple path in G?

SteinerTree: Given an undirected graph G with some of the vertices marked, what is the minimum number of edges in a subtree of G that contains every marked vertex?

SubsetSum: Given a set X of positive integers and an integer k, does X have a subset whose elements sum to k?

Partition: Given a set X of positive integers, can X be partitioned into two subsets with the same sum?

3Partition: Given a set X of $3n$ positive integers, can X be partitioned into n three-element subsets, all with the same sum?

IntegerLinearProgramming: Given a matrix $A \in \mathbb{Z}^{n \times d}$ and two vectors $b \in \mathbb{Z}^n$ and $c \in Z^d$, compute $\max\{c \cdot x \mid Ax \leq b, x \geq 0, x \in \mathbb{Z}^d\}$.

FeasibleILP: Given a matrix $A \in \mathbb{Z}^{n \times d}$ and a vector $b \in \mathbb{Z}^n$, determine whether the set of feasible integer points $\max\{x \in \mathbb{Z}^d \mid Ax \leq b, x \geq 0\}$ is empty.

Draughts: Given an $n \times n$ international draughts configuration, what is the largest number of pieces that can (and therefore must) be captured in a single move?

SuperMarioBrothers: Given an $n \times n$ Super Mario Brothers level, can Mario reach the castle?

CS/ECE 374 A ✧ Fall 2023

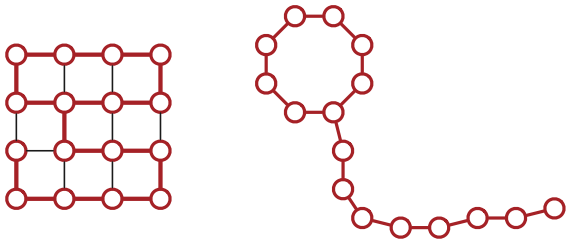
Homework 11

Due Tuesday, November 28, 2023 at 9pm

This is the last graded homework before the final exam.

---

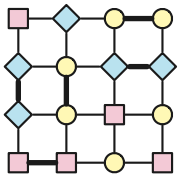
1. A *balloon* of size ℓ is an undirected graph consisting of a (simple) cycle of length ℓ and a (simple) path of length ℓ, where one endpoint of the path lies on the cycle, and otherwise the cycle and the path are disjoint. Every balloon of size ℓ has exactly 2ℓ vertices and 2ℓ edges. For example, the 4×4 grid graph shown below contains a balloon subgraph of size 8.



Prove that it is NP-hard to find the size of the the largest balloon subgraph of a given undirected graph.

2. Recall that a *3-coloring* of a graph assigns each vertex one of three colors, say red, yellow, and blue. A 3-coloring is *proper* if every edge has endpoints with different colors. The 3Color problem asks, given an arbitrary undirected graph G, whether G has a proper 3-coloring.

   Call a 3-coloring of a graph G *slightly improper* if each vertex has *at most one neighbor* with the same color. The SlightlyImproper3Color problem asks, given an arbitrary undirected graph G, whether G has a slightly improper 3-coloring.



   (a) Consider the following attempt to prove that SlightlyImproper3Color is NP-hard, using a reduction from 3Color.

   Non-solution: We reduce from 3Color. Given an arbitrary input graph G, we construct a new graph H by attaching a clique of 4 vertices to every vertex of G. Specifically, for each vertex v in G, the graph H contains three new vertices v_1, v_2, v_3, along with edges $vv_1, vv_2, vv_3, v_1v_2, v_1v_3, v_2v_3$. I claim that

G has a proper 3-coloring
if and only if
H has a slightly improper 3-coloring.

$\Longrightarrow$ Suppose G has a proper 3-coloring, using the colors red, yellow, and blue. Extend this color assignment to the vertices of H by coloring each vertex v_1 red, each vertex v_2 yellow, and each vertex v_3 blue. With this assignment, each vertex of H has at most one neighbor with the same color. Specifically, each vertex of G has the same color as one of the vertices in its gadget, and the other two vertices in v's gadget have no neighbors with the same color.

$\Longleftarrow\!\!\!\!\!\!\times$ Now suppose H has a slightly improper 3-coloring. Then G must have a proper 3-coloring because. . . um. . .

♣

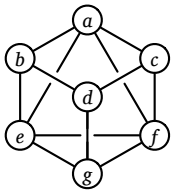
Describe a graph G that does not have a proper 3-coloring, such that the graph H constructed by this reduction does have a slightly improper 3-coloring.

(b) Describe a small graph X with the following property: In every slightly improper 3-coloring of X, every vertex of X has *exactly* one neighbor with the same color.

(c) Describe a correct polynomial-time reduction from 3Color to SlightlyImproper-3Color. *[Hint: Use your graph from part (b) as a gadget.]* This reduction will prove that SlightlyImproper3Color is indeed NP-hard.

Solved Problem

3. A *double-Hamiltonian tour* in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Prove that it is NP-hard to decide whether a given graph G has a double-Hamiltonian tour.



This graph contains the double-Hamiltonian tour $a \to b \to d \to g \to e \to b \to d \to c \to f \to a \to c \to f \to g \to e \to a$.

Solution: We prove the problem is NP-hard with a reduction from the standard Hamiltonian cycle problem. Let G be an arbitrary undirected graph. We construct a new graph H by attaching a small gadget to every vertex of G. Specifically, for each vertex v, we add two vertices $v^\sharp$ and $v^\flat$, along with three edges $vv^\flat$, $vv^\sharp$, and $v^\flat v^\sharp$.

A vertex in G and the corresponding vertex gadget in H.

Now I claim that

G has a Hamiltonian cycle if and only if H has a double-Hamiltonian tour.

$\Longrightarrow$ Suppose G contains a Hamiltonian cycle $C = v_1 \to v_2 \to \cdots \to v_n \to v_1$. We can construct a double-Hamiltonian tour of H by replacing each vertex v_i in C with the following walk:

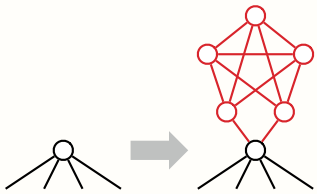
$$\cdots \to v_i \to v_i^\flat \to v_i^\sharp \to v_i^\flat \to v_i^\sharp \to v_i \to \cdots$$

$\Longleftarrow$ Conversely, suppose H has a double-Hamiltonian tour D. Consider any vertex v in the original graph G; the tour D must visit v exactly twice. Those two visits split D into two closed walks, each of which visits v exactly once. Any walk from $v^\flat$ or $v^\sharp$ to any other vertex in H must pass through v. Thus, one of the two closed walks visits only the vertices v, $v^\flat$, and $v^\sharp$. Thus, if we remove the vertices and edges in $H \setminus G$ from D, we obtain a closed walk in G that visits every vertex in G exactly once.

Given any graph G, we can clearly construct the corresponding graph H in polynomial time by brute force.

With more effort, we can construct a graph H that contains a double-Hamiltonian tour ***that traverses each edge of H at most once*** if and only if G contains a Hamiltonian

cycle. For each vertex v in G we attach a more complex gadget containing five vertices and eleven edges, as shown on the next page.

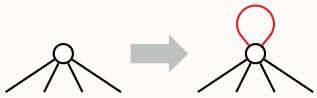


A vertex in G, and the corresponding modified vertex gadget in H.

■

Rubric: 10 points, standard polynomial-time reduction rubric. This is not the only correct solution.

Non-solution (self-loops): We attempt to prove the problem is NP-hard with a reduction from the Hamiltonian cycle problem. Let G be an arbitrary undirected graph. We construct a new graph H by attaching a self-loop every vertex of G. Given any graph G, we can clearly construct the corresponding graph H in polynomial time.



An incorrect vertex gadget.

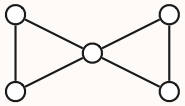
Now I claim that

G has a Hamiltonian cycle
if and only if
H has a double-Hamiltonian tour.

$\Longrightarrow$ Suppose G has a Hamiltonian cycle $v_1 \to v_2 \to \cdots \to v_n \to v_1$. We can construct a double-Hamiltonian tour of H by alternating between edges of the Hamiltonian cycle and self-loops: $v_1 \to v_1 \to v_2 \to v_2 \to v_3 \to \cdots \to v_n \to v_n \to v_1$.

$\not\Longleftarrow$ Um. . .

Unfortunately, if H has a double-Hamiltonian tour, we *cannot* conclude that G has a Hamiltonian cycle, because we cannot guarantee that a double-Hamiltonian tour in H uses *any* self-loops. The graph G shown below is a counterexample; it has a double-Hamiltonian tour (even before adding self-loops!) but no Hamiltonian cycle.



This graph has a double-Hamiltonian tour.

♣

Some useful NP-hard problems. You are welcome to use any of these in your own NP-hardness proofs, except of course for the specific problem you are trying to prove NP-hard.

CircuitSat: Given a boolean circuit, are there any input values that make the circuit output True?

3Sat: Given a boolean formula in conjunctive normal form, with exactly three distinct literals per clause, does the formula have a satisfying assignment?

MaxIndependentSet: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MaxClique: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MinVertexCover: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

MinSetCover: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subcollection whose union is S?

MinHittingSet: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subset of S that intersects every subset S_i?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

ChromaticNumber: Given an undirected graph G, what is the minimum number of colors required to color its vertices, so that every edge touches vertices with two different colors?

HamiltonianPath: Given graph G (either directed or undirected), is there a path in G that visits every vertex exactly once?

HamiltonianCycle: Given a graph G (either directed or undirected), is there a cycle in G that visits every vertex exactly once?

TravelingSalesman: Given a graph G (either directed or undirected) with weighted edges, what is the minimum total weight of any Hamiltonian path/cycle in G?

LongestPath: Given a graph G (either directed or undirected, possibly with weighted edges), what is the length of the longest simple path in G?

SteinerTree: Given an undirected graph G with some of the vertices marked, what is the minimum number of edges in a subtree of G that contains every marked vertex?

SubsetSum: Given a set X of positive integers and an integer k, does X have a subset whose elements sum to k?

Partition: Given a set X of positive integers, can X be partitioned into two subsets with the same sum?

3Partition: Given a set X of $3n$ positive integers, can X be partitioned into n three-element subsets, all with the same sum?

IntegerLinearProgramming: Given a matrix $A \in \mathbb{Z}^{n \times d}$ and two vectors $b \in \mathbb{Z}^n$ and $c \in Z^d$, compute $\max\{c \cdot x \mid Ax \leq b, x \geq 0, x \in \mathbb{Z}^d\}$.

FeasibleILP: Given a matrix $A \in \mathbb{Z}^{n \times d}$ and a vector $b \in \mathbb{Z}^n$, determine whether the set of feasible integer points $\max\{x \in \mathbb{Z}^d \mid Ax \leq b, x \geq 0\}$ is empty.

Draughts: Given an $n \times n$ international draughts configuration, what is the largest number of pieces that can (and therefore must) be captured in a single move?

SuperMarioBrothers: Given an $n \times n$ Super Mario Brothers level, can Mario reach the castle?

CS/ECE 374 A ✧ Fall 2023

☙ "Homework" 12 ❧

"Due" Monday, December 4, 2023

This homework is *not* for submission. However, we are planning to ask a few (true/false, multiple-choice, or short-answer) questions about undecidability on the final exam, so we still strongly recommend treating these questions as regular homework. Solutions will be released next Monday.

---

1. Let $\langle M \rangle$ denote the encoding of a Turing machine M (or if you prefer, the Python source code for the executable code M). Recall that w^R denotes the reversal of string w. Prove that the following language is undecidable.

$$\textsc{SelfRevAccept} := \left\{ \langle M \rangle \;\middle|\; M \text{ accepts the string } \langle M \rangle^R \right\}$$

   Note that Rice's theorem does *not* apply to this language.

2. Let M be a Turing machine, let w be a string, and let s be an integer. We say that ***M accepts w in space s*** if, given w as input, M accesses ***at most*** the first s cells on its tape and eventually accepts. (If you prefer to think in terms of programs instead of Turing machines, "space" is how much memory your program needs to run correctly.)

   Prove that the following language is undecidable:

$$\textsc{SomeSquareSpace} = \left\{ \langle M \rangle \;\middle|\; M \text{ accepts at least one string } w \text{ in space } |w|^2 \right\}$$

   Note that Rice's theorem does *not* apply to this language.

   [Hint: The only thing you actually need to know about Turing machines for this problem is that they consume a resource called "space".]

3. Prove that the following language is undecidable:

$$\textsc{Picky} = \left\{ \langle M \rangle \;\middle|\; \begin{array}{c} M \text{ accepts at least one input string} \\ \text{and } M \text{ rejects at least one input string} \end{array} \right\}$$

   Note that Rice's theorem does *not* apply to this language.

Solved Problem

4. Consider the language SOMETIMESHALT $= \{\langle M \rangle \mid M \text{ halts on at least one input string}\}$. Note that $\langle M \rangle \in$ SOMETIMESHALT does not imply that M *accepts* any strings; it is enough that M *halts* on (and possibly rejects) some string.

   (a) Prove that SOMETIMESHALT is undecidable.

Solution (Rice): Let $\mathscr{L}$ be the family of all non-empty languages. Let N be any Turing machine that never halts, so $\text{HALT}(N) = \emptyset \notin \mathscr{L}$. Let Y be any Turing machine that always halts, so $\text{HALT}(Y) = \Sigma^* \in \mathscr{L}$. Rice's Halting Theorem immediately implies that SOMETIMESHALT $=$ HALTIN$(\mathscr{L})$ is undecidable. ■

Solution (closure): Let ENCODINGS be the language of all Turing machine encodings (for some fixed universal Turing machine); this language is decidable. We immediately have ENCODINGS $=$ NEVERHALT $\cup$ SOMETIMESHALT, or equivalently, NEVERHALT $=$ ENCODINGS $\setminus$ SOMETIMESHALT.

The lectures notes include a proof that NEVERHALT is undecidable. On the other hand, the existence of a universal Turing machine implies that ENCODINGS is decidable. So Corollary 3(d) in the undecidability notes implies that SOMETIMESHALT is undecidable. ■

Solution (reduction from HALT): We can reduce the standard halting problem to SOMETIMESHALT as follows:

```
DECIDEHALT(⟨M, w⟩):
  Encode the following Turing machine M′:
    M′(x):
      (ignore x)
      run M on input w
  return DECIDESOMETIMESHALT(⟨M′⟩)
```

We prove this reduction correct as follows:

$\Longrightarrow$ Suppose M halts on input w.
Then M' halts on *every* input string x.
So DECIDESOMETIMESHALT must accept the encoding $\langle M' \rangle$.
We conclude that 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.
So DECIDESOMETIMESHALT must reject the encoding $\langle M' \rangle$.
We conclude that DECIDEHALT correctly rejects the encoding $\langle M, w \rangle$.

■

CS/ECE 374 A ✧ Fall 2023

Midterm 1 Study Questions

This is a "core dump" of potential questions for Midterm 1. This should give you a good idea of the *types* of questions that we will ask on the exam—in particular, there *will* be a series of True/False/explain questions—but the actual exam questions may or may not appear in this handout. This list intentionally includes a few questions that are too long or difficult for exam conditions; most of these are indicated with a *star.

Questions from Jeff's past exams are labeled with the semester they were used—⟨⟨*S14*⟩⟩ or ⟨⟨*F19*⟩⟩, for example. Questions from this semester's homework (either written or on PrairieLearn) are labeled ⟨⟨*HW*⟩⟩. Questions from this semester's labs are labeled ⟨⟨*Lab*⟩⟩. Some unflagged questions may have been used in exams by other instructors. Many of these problems appear as auto-graded practice exercises on PrairieLearn.

How to Use These Problems

Solving every problem in this handout is ***not*** the best way to study for the exam. Memorizing the solutions to every problem in this handout is the ***absolute worst*** way to study for the exam.

Instead we recommend *sampling* the problems. Choose one or two problems at random from each section and try to solve them from scratch under exam conditions—by yourself, in a quiet room, with a 30-minute timer, *without* your notes, *without* the internet, and if possible, even without your cheat sheet. If you can comfortably solve a few problems in some section under exam conditions, you're ready for that type of problem on the exam. Move on to the next section.

Discussing problems with other people (in your study groups, in the review sessions, in office hours, on Discord, or on Ed Discussion) and/or looking up old solutions can be *extremely* helpful, but ***only after*** you have (1) made a good-faith effort to solve the problem on your own, and (2) you have either a candidate solution or some idea about where you're getting stuck.

If you find yourself getting stuck on a particular type of problem, try to figure out *why* you're stuck. Do you understand the problem statement? Are you stuck on choosing the right high-level approach? Are you stuck on the technical details? Are you struggling to express your ideas clearly? Are you confused about notation?

Similarly, if past feedback suggests that your solutions to some problem types are incorrect or incomplete, try to figure out what you missed. The grading rubrics can be incredibly useful here. For induction proofs: Are you sure you have the right induction hypothesis? Are your cases obviously exhaustive? For regular expressions, DFAs, NFAs, and context-free grammars: Is your solution both exclusive and exhaustive? Did you try both positive and negative examples? Both short and long examples? For fooling sets: Are you imposing enough structure? Are x and y really *arbitrary* strings from F? For language transformations: Are you transforming in the right direction? Are you using non-determinism correctly? Do you understand the formal notation?

Remember that your goal is *not* merely to "understand"—or worse, to *remember*—the solution to any particular problem, but to become more comfortable with solving each *type* of problem on your own. **"Understanding" is a seductive trap; aim for mastery.** If you can identify specific steps that you find problematic, read more *about those steps*, focus your practice *on those steps*, and look for helpful information *about those steps* to write on your cheat sheet. Then work on the next problem!

Induction on Strings

Give complete, formal inductive proofs for the following claims. Your proofs must reply on the formal recursive definitions of the relevant string functions, not on intuition. Recall that the concatenation $\bullet$ and length $|\cdot|$ functions are formally defined as follows:

$$w \bullet y := \begin{cases} y & \text{if } w = \varepsilon \\ a \cdot (x \bullet y) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

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

1.1 The ***reversal*** w^R of a string w is defined recursively as follows:

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

(a) Prove that $(w \bullet x)^R = x^R \bullet w^R$ for all strings w and x. ⟨⟨***Lab***⟩⟩

(b) Prove that $(w^R)^R = w$ for every string w. ⟨⟨***Lab***⟩⟩

(c) Prove that $|w| = |w^R|$ for every string w. ⟨⟨***Lab***⟩⟩

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

(a) Give a formal recursive definition of $\#(a, w)$.⟨⟨***Lab***⟩⟩

(b) Prove that $\#(a, w \bullet z) = \#(a, w) + \#(a, z)$ for all symbols a and all strings w and z. ⟨⟨***Lab***⟩⟩

(c) Prove that $\#(a, w^R) = \#(a, w)$ for all symbols a and all strings w, where w^r denotes the reversal of w. ⟨⟨***Lab***⟩⟩

1.3 For any string w and any non-negative integer n, let $\boldsymbol{w^n}$ denote the string obtained by concatenating n copies of w; more formally, define

$$w^n := \begin{cases} \varepsilon & \text{if } n = 0 \\ w \bullet w^{n-1} & \text{otherwise} \end{cases}$$

For example, $(\texttt{BLAH})^5 = \texttt{BLAHBLAHBLAHBLAHBLAH}$ and $\varepsilon^{374} = \varepsilon$.

(a) Prove that $w^m \bullet w^n = w^{m+n}$ for every string w and all non-negative integers n and m.

(b) Prove that $(w^m)^n = w^{mn}$ for every string w and all non-negative integers n and m.

(c) Prove that $|w^n| = n|w|$ for every string w and every integer $n \geq 0$.

(d) Prove that $(w^n)^R = (w^R)^n$ for every string w and every integer $n \geq 0$.

1.4 Consider the following pair of mutually recursive functions:

$$evens(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ odds(x) & \text{if } w = ax \end{cases} \qquad odds(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ a \cdot evens(x) & \text{if } w = ax \end{cases}$$

For example, $evens(\texttt{0001101}) = \texttt{010}$ and $odds(\texttt{0001101}) = \texttt{0011}$.

(a) Prove the following identity for all strings w and x:

$$evens(w \bullet x) = \begin{cases} evens(w) \bullet evens(x) & \text{if } |w| \text{ is even,} \\ evens(w) \bullet odds(x) & \text{if } |w| \text{ is odd.} \end{cases}$$

(b) State and prove a similar identity for $odds(w \bullet x)$.

(c) Prove the following identity for all strings w:

$$evens(w^R) = \begin{cases} (evens(w))^R & \text{if } |w| \text{ is odd,} \\ (odds(w))^R & \text{if } |w| \text{ is even.} \end{cases}$$

(d) Prove that $|w| = |evens(w)| + |odds(w)|$ for every string w.

1.5 The ***complement*** w^c of a string $w \in \{\texttt{0}, \texttt{1}\}^*$ is obtained from w by replacing every $\texttt{0}$ in w with a $\texttt{1}$ and vice versa. The complement function can be defined recursively as follows:

$$w^c := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \texttt{1} \cdot x^c & \text{if } w = \texttt{0}x \\ \texttt{0} \cdot x^c & \text{if } w = \texttt{1}x \end{cases}$$

(a) Prove that $|w| = |w^c|$ for every string w.

(b) Prove that $(x \bullet y)^c = x^c \bullet y^c$ for all strings x and y.

(c) Prove that $\#(\texttt{1}, w) = \#(\texttt{0}, w^c)$ for every string w.

1.6 Consider the following recursively defined function:

$$stutter(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ aa \bullet stutter(x) & \text{if } w = ax \end{cases}$$

For example, $stutter(\texttt{MISSISSIPPI}) = \texttt{MMIISSSSIISSSSIIPPPPII}$.

(a) Prove that $|stutter(w)| = 2|w|$ for every string w. ⟨⟨***HW***⟩⟩

(b) Prove that $evens(stutter(w)) = w$ for every string w.

(c) Prove that $odds(stutter(w)) = w$ for every string w.

(d) Prove that w is a palindrome if and only if $stutter(w)$ is a palindrome, for every string w.

1.7 Consider the following recursive function:

$$\mathit{shuffle}(w,z) := \begin{cases} z & \text{if } w = \varepsilon \\ a \cdot \mathit{shuffle}(z,x) & \text{if } w = ax \end{cases}$$

For example, $\mathit{shuffle}(\texttt{0011},\texttt{0101}) = \texttt{00011011}$.

(a) Prove that $|\mathit{shuffle}(x,y)| = |x| + |y|$ for all strings x and y.

(b) Prove that $\mathit{shuffle}(w,w) = \mathit{stutter}(w)$ for every string w.

(c) Prove that $\mathit{shuffle}(\mathit{odds}(w), \mathit{evens}(w)) = w$ for every string w.

(d) Prove that $\mathit{evens}(\mathit{shuffle}(w,z)) = z$ for all strings w and z such that $|w| = |z|$.

Regular expressions

For each of the following languages over the alphabet $\Sigma = \{0, 1\}$, give an equivalent regular expression, and *briefly* argue why your expression is correct. (On exams, we will not ask for justifications, but you should still justify your expressions in your head.)

2.1 Every string of length at most 3. *[Hint: Don't try to be clever.]*

2.2 All strings except `010`. ⟨⟨***Lab***⟩⟩

2.3 All strings that end with the suffix `010`.

2.4 All strings that do not start with the prefix `010`.

2.5 All strings that contain the substring `010`. ⟨⟨***Lab***⟩⟩

2.6 All strings that do not contain the substring `010`.

2.7 All strings that contain the subsequence `010`.

2.8 All strings that do not contain the subsequence `010`.

2.9 All strings containing the substring `10` or the substring `01`.

2.10 All strings containing either the substring `10` or the substring `01`, but not both. ⟨⟨***F16***⟩⟩

2.11 All strings that do not contain either `001` or `110` as a substring. ⟨⟨***F19***⟩⟩

2.12 All strings containing the subsequence `10` or the subsequence `01` (or possibly both).

2.13 All strings containing the subsequence `10` or the subsequence `01`, but not both.

2.14 All strings containing at least two `1`s and at least one `0`. ⟨⟨***Lab***⟩⟩

2.15 All strings containing at least two `1`s or at least one `0` (or possibly both).

2.16 All strings containing at least two `1`s or at least one `0`, but not both.

2.17 All strings in which every run of consecutive `0`s has even length. ⟨⟨***S21***⟩⟩

2.18 All strings in which every run of consecutive `0`s has even length and every run of consecutive `1`s has odd length. ⟨⟨***F14***⟩⟩

2.19 All strings whose length is divisible by 3.

2.20 All strings in which the number of `1`s is divisible by 3.

2.21 All strings in 0^*1^* whose length is divisible by 3. ⟨⟨***S14***⟩⟩

2.22 All strings in 0^*10^* whose length is divisible by 3. ⟨⟨***S18***⟩⟩

2.23 All strings in $0^*1^*0^*$ whose length is even. ⟨⟨***S18***⟩⟩

2.24 $\{0^n w 1^n \mid n > 1 \text{ and } q \in \Sigma^*\}$ ⟨⟨***S18***⟩⟩

Direct DFA construction

Draw or formally describe a DFA that recognizes each of the following languages. Don't forget to describe the states of your DFA in English. Unless otherwise specified, all languages are over the alphabet $\Sigma = \{0, 1\}$.

2.1 The language $\{\texttt{LONG}, \texttt{LUG}, \texttt{LEGO}, \texttt{LEG}, \texttt{LUG}, \texttt{LOG}, \texttt{LINGO}\}$.

2.2 The language $\texttt{MOO}^* + \texttt{MEOO}^*\texttt{W}$

2.3 Every string of length at most 3.

2.4 All strings except 010. ⟨⟨*Lab*⟩⟩

2.5 All strings that end with the suffix 010.

2.6 All strings that do not start with the prefix 010.

2.7 All strings that contain the substring 010. ⟨⟨*Lab*⟩⟩

2.8 All strings that do not contain the substring 010. ⟨⟨*Lab*⟩⟩

2.9 All strings that contain the subsequence 010.

2.10 All strings containing the substring 10 or the substring 01.

2.11 All strings containing either the substring 10 or the substring 01, but not both. ⟨⟨*F16*⟩⟩

2.12 All strings that do not contain either 001 or 110 as a substring. ⟨⟨*F19*⟩⟩

2.13 All strings containing the subsequence 10 or the subsequence 01 (or possibly both).

2.14 All strings containing at least two 1s and at least one 0. ⟨⟨*Lab*⟩⟩

2.15 All strings containing at least two 1s or at least one 0, but not both.

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

2.17 All strings in which every run of consecutive 0s has even length. ⟨⟨*S21*⟩⟩

2.18 All strings in which every run of consecutive 0s has even length and every run of consecutive 1s has odd length. ⟨⟨*F14*⟩⟩

2.19 All strings that end with 01 and that have odd length ⟨⟨*S21*⟩⟩

2.20 All strings in which the number of 1s is divisible by 3.

2.21 All strings that represent an integer divisible by 3 in binary.

2.22 All strings that represent an integer divisible by 5 in base 7.

2.23 All strings in 0^*1^* whose length is divisible by 3. ⟨⟨*S14*⟩⟩

2.24 All strings in 0^*10^* whose length is divisible by 3. ⟨⟨*S18*⟩⟩

2.25 All strings in $0^*1^*0^*$ whose length is even. ⟨⟨*S18*⟩⟩

2.26 $\{0^n w 1^n \mid n > 1 \text{ and } q \in \Sigma^*\}$ ⟨⟨*S18*⟩⟩

Fooling sets

Prove that each of the following languages is *not* regular. Unless specified otherwise, all languages are over the alphabet $\Sigma = \{0, 1\}$.

4.1 All strings with more 0s than 1s. ⟨⟨*S14*⟩⟩

4.2 All strings with exactly twice as many 0s as 1s.

4.3 All strings with at least twice as many 0s as 1s.

4.4 $\{0^{2^n} \mid n \geq 0\}$ ⟨⟨*Lab*⟩⟩

4.5 $\{0^{3^n} \mid n \geq 0\}$ ⟨⟨*S21*⟩⟩

4.6 $\{0^{F_n} \mid n \geq 0\}$, where F_n is the nth Fibonacci number, defined recursively as follows:

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

[Hint: If $F_i + F_j$ is a Fibonacci number, then either $i = j \pm 1$ or $\min\{i, j\} \leq 2$.]

4.7 $\{0^{n^2} \mid n \geq 0\}$ ⟨⟨*Lab*⟩⟩

4.8 $\{0^{n^3} \mid n \geq 0\}$

4.9 $\{0^m 1^n \mid m \neq 2n\}$ ⟨⟨*Lab*⟩⟩

4.10 $\{0^i 1^j 0^k \mid 2i = k \text{ or } i = 2k\}$ ⟨⟨*S18*⟩⟩

4.11 $\{0^i 1^j 0^k \mid i + j = 2k\}$ ⟨⟨*F19*⟩⟩

4.12 $\{0^i 1^j 0^k \mid k > 0 \text{ and } j \bmod k = 0 \text{ and } i \bmod k = 0\}$ ⟨⟨*F21*⟩⟩

4.13 $\{0^m 1^n \mid n > 0 \text{ and } m = n^n\}$ ⟨⟨*F21*⟩⟩

4.14 $\{(01)^n (10)^n \mid n \geq 0\}$

4.15 $\{(01)^m (10)^n \mid n \geq m \geq 0\}$

4.16 $\{x\#y \mid x, y \in \{0, 1\}^* \text{ and } \#(0, x) = \#(1, y)\}$

4.17 $\{xx^c \mid x \in \{0, 1\}^*\}$, where x^c is the *bitwise complement* of x, obtained by replacing every 0 in x with a 1 and vice versa. For example, $0001101^c = 1110010$.

4.18 Properly balanced strings of parentheses, described by the context-free grammar $S \to \varepsilon \mid SS \mid (S)$. ⟨⟨*Lab*⟩⟩

4.19 Palindromes whose length is divisible by 3. ⟨⟨*Lab*⟩⟩

4.20 Strings in which at least two runs of consecutive 0s have the same length.

4.21 $\{w\#x\#y \mid w, x, y \in \Sigma^* \text{ and } w, x, y \text{ are not all equal}\}$

Regular or Not?

For each of the following languages, either ***prove*** that the language is regular (for example, by describing a DFA, NFA, or regular expression), or ***prove*** that the language is not regular (for example, using a fooling set argument). Unless otherwise specified, all languages are over the alphabet $\Sigma = \{0, 1\}$. Read the language descriptions ***very*** carefully.

5.1 The set of all strings in $\{0,1\}^*$ in which the substrings 01 and 10 appear the same number of times. (For example, the substrings 01 and 01 each appear three times in the string 1100001101101.) ⟨⟨***F14,HW***⟩⟩

5.2 The set of all strings in $\{0,1\}^*$ in which the substrings 00 and 11 appear the same number of times. (For example, the substrings 00 and 11 each appear three times in the string 1100001101101.) ⟨⟨***F14,HW***⟩⟩

5.3 $\{www \mid w \in \Sigma^*\}$ ⟨⟨***F14***⟩⟩

5.4 $\{wxw \mid w, x \in \Sigma^*\}$ ⟨⟨***F14***⟩⟩

5.5 All strings such that in every prefix, the number of 0s is greater than the number of 1s.

5.6 All strings such that in every *non-empty* prefix, the number of 0s is greater than the number of 1s.

5.7 $\{0^m 1^n \mid 0 \le m - n \le 374\}$

5.8 $\{0^m 1^n \mid 0 \le m + n \le 374\}$

5.9 The language generated by the following context-free grammar:

$$S \to 0A1 \mid \varepsilon$$
$$A \to 1S0 \mid \varepsilon$$

5.10 The language generated by the context-free grammar $S \to 0S1 \mid 1S0 \mid \varepsilon$

5.11 $\{0^i 1^j 0^k \mid k = i + j\}$ ⟨⟨***F21***⟩⟩

5.12 $\{0^i 1^j 0^k \mid k \equiv i + j \pmod 2\}$ ⟨⟨***F21,HW***⟩⟩

5.13 $\{w\#x \mid w, x \in \{0,1\}^*$ and w is a substring of $x\}$

5.14 $\{w\#x \mid w, x \in \{0,1\}^*$ and w is a proper substring of $x\}$

5.15 $\{xy \mid x$ is a palindrome and y is a palindrome$\}$ ⟨⟨***F19***⟩⟩

5.16 $\{xy \mid x$ is not a palindrome$\}$ ⟨⟨***F19***⟩⟩

5.17 $\{xy \mid x$ is a palindrome and $|x| > 1\}$ ⟨⟨***F19***⟩⟩

5.18 $\{xy \mid \#(0,x) = \#(1,y)$ and $\#(1,x) = \#(0,y)\}$

5.19 $\{xy \mid \#(0,x) = \#(1,y)$ or $\#(1,x) = \#(0,y)\}$

Product/Subset Constructions

For each of the following languages L over the alphabet $\{0,1\}$, formally describe a DFA $M = (Q, s, A, \delta)$ that recognizes L. ***Do not attempt to draw the DFA. Do not use the phrase "product construction".*** Instead, give a complete, precise, and self-contained description of the state set Q, the start state s, the accepting state A, and the transition function δ.

6.1 ⟨⟨*S14*⟩⟩ All strings that satisfy *all* of the following conditions:

(a) the number of 0s is even

(b) the number of 1s is divisible by 3

(c) the total length is divisible by 5

6.2 All strings that satisfy *at least one* of the following conditions: ...

6.3 All strings that satisfy *exactly one* of the following conditions: ...

6.4 All strings that satisfy *exactly two* of the following conditions: ...

6.5 All strings that satisfy *an odd number of* of the following conditions: ...

6.6 All strings w such that $(\#(0,w) \bmod 3) + (\#(1,w) \bmod 3)$ is odd.

- Other possible conditions:

(a) The number of 0s in w is odd.

(b) The number of 1s in w is not divisible by 5.

(c) The length $|w|$ is divisible by 7.

(d) The binary value of w is divisible by 7.

(e) w represents a number divisible by 5 in base 7.

(f) w contains the substring 00

(g) w does not contain the substring 11

(h) ww does not contain the substring 101

Regular Language Transformations

Let L be an arbitrary regular language over the alphabet $\Sigma = \{0, 1\}$. ***Prove*** that each of the following languages is regular.

7.0 L^*

7.1 All strings in L whose length is divisible by 3.

7.2 $\textsc{OneInFront}(L) := \{1x \mid x \in L\}$

7.3 $\textsc{MissingFirstOne}(L) := \{w \in \Sigma^* \mid 1w \in L\}$

7.4 $\textsc{MissingOneOne}(L) := \{xy \mid x1y \in L\}$ ⟨⟨***Lab***⟩⟩

7.5 $\textsc{Prefixes}(L) := \{x \mid xy \in L \text{ for some } y \in \Sigma^*\}$

7.6 $\textsc{Suffixes}(L) := \{y \mid xy \in L \text{ for some } x \in \Sigma^*\}$ ⟨⟨***F16***⟩⟩

7.7 $\textsc{Evens}(L) := \{evens(w) \mid w \in L\}$, where the functions *evens* and *odds* are recursively defined as follows:

$$evens(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ odds(x) & \text{if } w = ax \end{cases} \qquad odds(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ a \cdot evens(x) & \text{if } w = ax \end{cases}$$

For example, $evens(0001101) = 010$ and $odds(0001101) = 0011$. ⟨⟨***F14, Lab***⟩⟩

7.8 $\textsc{Unevens}(L) := \{w \mid evens(w) \in L\}$, where the functions *evens* and *odds* are recursively defined as above. ⟨⟨***F14, Lab***⟩⟩

7.9 $\textsc{AddParity}(L) = \{addparity(w) \mid w \in L\}$, where ⟨⟨***S18***⟩⟩

$$addparity(w) = \begin{cases} 0w & \text{if } \#(1, w) \text{ is even} \\ 1w & \text{if } \#(1, w) \text{ is odd} \end{cases}$$

7.10 $\textsc{StripFinal0s}(L) = \{w \mid w0^n \in L \text{ for some } n \geq 0\}$. Less formally, $\textsc{StripFinal0s}(L)$ is the set of all strings obtained by removing any number of final 0s from strings in L. ⟨⟨***S18***⟩⟩

7.11 $\textsc{Obliviate}(L) := \{obliviate(w) \mid w \in L\}$, where $obliviate(w) = 0^{\#(0,w)}$ is the string obtained from w by deleting every 1. ⟨⟨***F19***⟩⟩

7.12 $\textsc{UnObliviate}(L) := \{w \in \Sigma^* \mid obliviate(w) \in L\}$, where $obliviate(w) = 0^{\#(0,w)}$ is the string obtained from w by deleting every 1. ⟨⟨***F19***⟩⟩

7.13 $\textsc{SameSlash}(w) = \{sameslash(w) \mid w \in L\}$, where $sameslash(w)$ is the string in $\{0, 1, /\}$ obtained from w by inserting a new symbol / between any two consecutive appearances of the same symbol. ⟨⟨***F19***⟩⟩

7.14 $\textsc{DiffSlash}(w) = \{diffslash(w) \mid w \in L\}$, where $diffslash(w)$ is the string in $\{0, 1, /\}$ obtained from w by inserting a new symbol / between any two consecutive symbols that are ***not*** equal. ⟨⟨***F19***⟩⟩

Context-Free Grammars

Construct context-free grammars for each of the following languages, and give a *brief* explanation of how your grammar works, including the language of each non-terminal. Unless specified otherwise, all languages are over the alphabet $\{0, 1\}$. We explicitly do ***not*** want a formal proof of correctness.

8.1 All strings in $\{0, 1\}^*$ whose length is divisible by 5.

8.2 All strings in which the substrings 01 and 10 appear the same number of times.

8.3 $\{0^n 1^{2n} \mid n \geq 0\}$ ⟨⟨***Lab***⟩⟩

8.4 $\{0^m 1^n \mid n \neq 2m\}$ ⟨⟨***Lab***⟩⟩

8.5 $\{0^i 1^j 0^{i+j} \mid i, j \geq 0\}$

8.6 $\{0^{i+j} \# 0^j \# 0^i \mid i, j \geq 0\}$

8.7 $\{0^i 1^j 2^k \mid j \neq i + k\}$

8.8 $\{0^i 1^j 2^k \mid i = 2k \text{ or } 2i = k\}$ ⟨⟨***S18***⟩⟩

8.9 $\{0^i 1^j 2^k \mid i + j = 2k\}$ ⟨⟨***F19***⟩⟩

8.10 $\left\{w \# 0^{\#(0,w)} \;\middle|\; w \in \{0, 1\}^*\right\}$

8.11 $\{0^i 1^j 2^k \mid i = j \text{ or } j = k \text{ or } i = k\}$

8.12 $\{0^i 1^j 2^k \mid i \neq j \text{ or } j \neq k\}$

8.13 $\{0^{2i} 1^{i+j} 2^{2j} \mid i, j \geq 0\}$

8.14 $\left\{x \# y^R \;\middle|\; x, y \in \{0, 1\}^* \text{ and } x \neq y\right\}$

8.15 All strings in $\{0, 1\}^*$ that are *not* palindromes. ⟨⟨***HW***⟩⟩

8.16 $\left\{0^n 1^{an+b} \;\middle|\; n \geq 0\right\}$, where a and b are arbitrary fixed natural numbers.

8.17 $\left\{0^n 1^{an-b} \;\middle|\; n \geq b/a\right\}$, where a and b are arbitrary fixed natural numbers.

True or False (sanity check)

For each statement below, check "Yes" if the statement is ***ALWAYS*** true and "No" otherwise, and give a ***brief*** explanation of your answer. For example:

[X] Yes [ ] No If $2+2=5$ then Jeff is the Queen of England.
The hypothesis is false, so the implication is true.

[ ] Yes [X] No $x+y$ is even.
Suppose $x=1$ and $y=0$.

[X] Yes [ ] No The set of all binary strings with an even number of 1s is regular.
$0^*(10^*10^*)^*$
—*or*—
Accepted by 2-state DFA, where current state = #1*s* mod 2.

Read each statement *very* carefully. Some of these are deliberately subtle. On the other hand, you should not spend more than two minutes on any single statement.

Definitions

A.1 Every language is regular.

A.2 Every finite language is regular.

A.3 Every infinite language is regular.

A.4 For every language L, if L is regular then L can be represented by a regular expression.

A.5 For every language L, if L is not regular then L cannot be represented by a regular expression.

A.6 For every language L, if L can be represented by a regular expression, then L is regular.

A.7 For every language L, if L cannot be represented by a regular expression, then L is not regular.

A.8 For every language L, if there is a DFA that accepts every string in L, then L is regular.

A.9 For every language L, if there is a DFA that accepts every string not in L, then L is not regular.

A.10 For every language L, if there is a DFA that rejects every string not in L, then L is regular.

A.11 For every language L, if for every string $w \in L$ there is a DFA that accepts w, then L is regular.
⟨⟨*S14*⟩⟩

A.12 For every language L, if for every string $w \notin L$ there is a DFA that rejects w, then L is regular.

A.13 For every language L, if some DFA recognizes L, then some NFA also recognizes L.

A.14 For every language L, if some NFA recognizes L, then some DFA also recognizes L.

A.15 For every language L, if some NFA with ε-transitions recognizes L, then some NFA without ε-transitions also recognizes L.

A.16 For every language L, and for every string $w \in L$, there is a DFA that accepts w. ⟨⟨***F19***⟩⟩

A.17 Every regular language is recognized by a DFA with exactly 374 accepting states. ⟨⟨***F19***⟩⟩

A.18 Every regular language is recognized by an NFA with exactly 374 accepting states. ⟨⟨***F19***⟩⟩

Closure Properties of Regular Languages

B.1 For all regular languages L and L', the language $L \cap L'$ is regular.

B.2 For all regular languages L and L', the language $L \cup L'$ is regular.

B.3 For all regular languages L, the language L^* is regular.

B.4 For all regular languages A, B, and C, the language $(A \cup B) \setminus C$ is regular.

B.5 For all languages $L \subseteq \Sigma^*$, if L is regular, then $\Sigma^* \setminus L$ is regular.

B.6 For all languages $L \subseteq \Sigma^*$, if L is regular, then $\Sigma^* \setminus L$ is not regular.

B.7 For all languages $L \subseteq \Sigma^*$, if L is not regular, then $\Sigma^* \setminus L$ is regular.

B.8 For all languages $L \subseteq \Sigma^*$, if L is not regular, then $\Sigma^* \setminus L$ is not regular.

B.9 For all languages L and L', the language $L \cap L'$ is regular. ⟨⟨***S14***⟩⟩

B.10 For all languages L and L', the language $L \cup L'$ is regular. ⟨⟨***F14***⟩⟩

B.11 For every language L, the language L^* is regular. ⟨⟨***F14, F16***⟩⟩

B.12 For every language L, if L^* is regular, then L is regular.

B.13 For all languages A, B, and C, the language $(A \cup B) \setminus C$ is regular.

B.14 For every language L, if L is finite, then L is regular.

B.15 For all languages L and L', if L and L' are finite, then $L \cup L'$ is regular.

B.16 For all languages L and L', if L and L' are finite, then $L \cap L'$ is regular.

B.17 For all languages L and L', if L is finite, then $L \cup L'$ is regular. ⟨⟨***F21***⟩⟩

B.18 For all languages L and L', if L is finite, then $L \cap L'$ is regular. ⟨⟨***F21***⟩⟩

B.19 For all languages $L \subseteq \Sigma^*$, if L contains infinitely many strings in Σ^*, then L is not regular.

B.20 For all languages $L \subseteq \Sigma^*$, if L contains all but a finite number of strings of Σ^*, then L is regular. ⟨⟨***S14***⟩⟩

B.21 For all languages $L \subseteq \{0, 1\}^*$, if L contains a finite number of strings in 0^*, then L is regular.

B.22 For all languages $L \subseteq \{0,1\}^*$, if L contains all but a finite number of strings in 0^*, then L is regular.

B.23 If L and L' are not regular, then $L \cap L'$ is not regular.

B.24 If L and L' are not regular, then $L \cup L'$ is not regular.

B.25 If L is regular and $L \cup L'$ is regular, then L' is regular. ⟨⟨*S14*⟩⟩

B.26 If L is regular and $L \cup L'$ is not regular, then L' is not regular. ⟨⟨*S14*⟩⟩

B.27 If L is not regular and $L \cup L'$ is regular, then L' is regular.

B.28 If L is regular and $L \cap L'$ is regular, then L' is regular.

B.29 If L is regular and $L \cap L'$ is not regular, then L' is not regular.

B.30 If L is regular and L' is finite, then $L \cup L'$ is regular. ⟨⟨*S14*⟩⟩

B.31 If L is regular and L' is finite, then $L \cap L'$ is regular.

B.32 If L is regular and $L \cap L'$ is finite, then L' is regular.

B.33 If L is regular and $L \cap L' = \emptyset$, then L' is not regular.

B.34 If L is not regular and $L \cap L' = \emptyset$, then L' is regular. ⟨⟨*F16*⟩⟩

B.35 If L is regular and L' is not regular, then $L \cap L' = \emptyset$.

B.36 If $L \subseteq L'$ and L is regular, then L' is regular.

B.37 If $L \subseteq L'$ and L' is regular, then L is regular. ⟨⟨*F14*⟩⟩

B.38 If $L \subseteq L'$ and L is not regular, then L' is not regular.

B.39 If $L \subseteq L'$ and L' is not regular, then L is not regular. ⟨⟨*F14*⟩⟩

B.40 Two languages L and L' are regular if and only if $L \cap L'$ is regular. ⟨⟨*F19*⟩⟩

B.41 For all languages $L \subseteq \Sigma^*$, if L cannot be described by a regular expression, then some DFA accepts $\Sigma^* \setminus L$.

B.42 For all languages $L \subseteq \Sigma^*$, if no DFA accepts L, then the complement $\Sigma^* \setminus L$ can be described by a regular expression.

B.43 For all languages $L \subseteq \Sigma^*$, if no DFA accepts L, then the complement $\Sigma^* \setminus L$ cannot be described by a regular expression.

B.44 For all languages $L \subseteq \Sigma^*$, if L is recognized by a DFA, then $\Sigma^* \setminus L$ can be described by a regular expression. ⟨⟨*F16*⟩⟩

Properties of Context-free Languages

C.1 For all languages $L \subseteq \Sigma^*$, if L cannot be recognized by a DFA, then L is context-free.

C.2 For all languages $L \subseteq \Sigma^*$, if L cannot be recognized by a DFA, then L is not context-free.

C.3 For all languages $L \subseteq \Sigma^*$, if L can be recognized by a DFA, then L is context-free.

C.4 For all languages $L \subseteq \Sigma^*$, if L can be recognized by a DFA, then L is not context-free.

C.5 For all languages $L \subseteq \Sigma^*$, if L is not context-free, then L is regular.

C.6 For all languages $L \subseteq \Sigma^*$, if L is not context-free, then $\Sigma^* \setminus L$ is regular.

C.7 For all languages $L \subseteq \Sigma^*$, if L is not context-free, then L is not regular.

C.8 For all languages $L \subseteq \Sigma^*$, if L is not context-free, then $\Sigma^* \setminus L$ is not regular.

C.9 The empty language is context-free. ⟨⟨*F19*⟩⟩

C.10 Every finite language is context-free.

C.11 Every context-free language is regular. ⟨⟨*F14*⟩⟩

C.12 Every regular language is context-free.

C.13 Every non-context-free language is non-regular. ⟨⟨*F16*⟩⟩

C.14 Every language is either regular or context-free. ⟨⟨*F19*⟩⟩

C.15 For all context-free languages L and L', the language $L \bullet L'$ is also context-free. ⟨⟨*F16*⟩⟩

C.16 For every context-free language L, the language L^* is also context-free.

C.17 For all context-free languages A, B, and C, the language $(A \cup B)^* \bullet C$ is also context-free.

C.18 For every language L, the language L^* is context-free.

C.19 For every language L, if L^* is context-free then L is context-free.

Equivalence Classes. Recall that for any language $L \subset \Sigma^*$, two strings $x, y \in \Sigma^*$ are equivalent with respect to L if and only if, for every string $z \in \Sigma^*$, either both xz and yz are in L, or neither xz nor yz is in L—or more concisely, if x and y have no distinguishing suffix with respect to L. We denote this equivalence by $x \equiv_L y$.

D.1 For every language L, if L is regular, then $\equiv_L$ has finitely many equivalence classes.

D.2 For every language L, if L is not regular, then $\equiv_L$ has infinitely many equivalence classes. ⟨⟨*S14*⟩⟩

D.3 For every language L, if $\equiv_L$ has finitely many equivalence classes, then L is regular.

D.4 For every language L, if $\equiv_L$ has infinitely many equivalence classes, then L is not regular.

D.5 For all regular languages L, each equivalence class of $\equiv_L$ is a regular language.

D.6 For every language L, each equivalence class of $\equiv_L$ is a regular language.

Fooling Sets

E.1 If a language L has an infinite fooling set, then L is not regular.

E.2 If a language L has an finite fooling set, then L is regular.

E.3 If a language L does not have an infinite fooling set, then L is regular.

E.4 If a language L is not regular, then L has an infinite fooling set.

E.5 If a language L is regular, then L has no infinite fooling set.

E.6 If a language L is not regular, then L has no finite fooling set. ⟨⟨***F14, F16***⟩⟩

E.7 If a language L has a fooling set of size 374, then L is not regular. ⟨⟨***F19***⟩⟩

E.8 If a language L does not have a fooling set of size 374, then L is regular. ⟨⟨***F19***⟩⟩

Specific Languages (Gut Check). Do *not* construct complete DFAs, NFAs, regular expressions, or fooling-set arguments for these languages. You don't have time.

F.1 $\{0^i 1^j 2^k \mid i + j - k = 374\}$ is regular. ⟨⟨***S14***⟩⟩

F.2 $\{0^i 1^j 2^k \mid i + j - k \le 374\}$ is regular.

F.3 $\{0^i 1^j 2^k \mid i + j + k = 374\}$ is regular.

F.4 $\{0^i 1^j 2^k \mid i + j + k > 374\}$ is regular.

F.5 $\{0^i 1^j \mid i < 374 < j\}$ is regular. ⟨⟨***S14***⟩⟩

F.6 $\left\{0^m 1^n \;\middle|\; 0 \le m + n \le 374\right\}$ is regular. ⟨⟨***F14***⟩⟩

F.7 $\left\{0^m 1^n \;\middle|\; 0 \le m - n \le 374\right\}$ is regular. ⟨⟨***F14***⟩⟩

F.8 $\{0^i 1^j \mid i, j \ge 0\}$ is not regular. ⟨⟨***F16***⟩⟩

F.9 $\{0^i 1^j \mid (i - j) \text{ is divisible by } 374\}$ is regular. ⟨⟨***S14***⟩⟩

F.10 $\{0^i 1^j \mid (i + j) \text{ is divisible by } 374\}$ is regular.

F.11 $\left\{0^{n^2} \;\middle|\; n \ge 0\right\}$ is regular.

F.12 $\left\{0^{37n+4} \;\middle|\; n \ge 0\right\}$ is regular.

F.13 $\left\{0^n 1 0^n \;\middle|\; n \ge 0\right\}$ is regular.

F.14 $\left\{0^m 1 0^n \;\middle|\; m \ge 0 \text{ and } n \ge 0\right\}$ is regular.

F.15 $\left\{0^{374n} \;\middle|\; n \ge 0\right\}$ is regular. ⟨⟨***F19***⟩⟩

F.16 $\left\{0^{37n} 1^{4n} \;\middle|\; n \ge 374\right\}$ is regular. ⟨⟨***F19***⟩⟩

F.17 $\left\{0^{37n}1^{4n} \;\middle|\; n \le 374\right\}$ is regular. ⟨⟨*F19*⟩⟩

F.18 $\{w \in \{0,1\}^* \mid |w| \text{ is divisible by } 374\}$ is regular.

F.19 $\{w \in \{0,1\}^* \mid w \text{ represents a integer divisible by } 374 \text{ in binary}\}$ is regular.

F.20 $\{w \in \{0,1\}^* \mid w \text{ represents a integer divisible by } 374 \text{ in base } 473\}$ is regular.

F.21 $\left\{w \in \{0,1\}^* \;\middle|\; |\#(0,w) - \#(1,w)| < 374\right\}$ is regular.

F.22 $\left\{w \in \{0,1\}^* \;\middle|\; |\#(0,x) - \#(1,x)| < 374 \text{ for every prefix } x \text{ of } w\right\}$ is regular.

F.23 $\left\{w \in \{0,1\}^* \;\middle|\; |\#(0,x) - \#(1,x)| < 374 \text{ for every substring } x \text{ of } w\right\}$ is regular.

F.24 $\left\{w0^{\#(0,w)} \;\middle|\; w \in \{0,1\}^*\right\}$ is regular.

F.25 $\left\{w0^{\#(0,w) \bmod 374} \;\middle|\; w \in \{0,1\}^*\right\}$ is regular.

Playing with Automata

G.1 Let $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q, s, Q \setminus A, \delta)$ be arbitrary **DFA**s with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then $L(M) \cap L(M') = \emptyset$. ⟨⟨*F16*⟩⟩

G.2 Let $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q, s, Q \setminus A, \delta)$ be arbitrary **NFA**s with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then $L(M) \cap L(M') = \emptyset$. ⟨⟨*F16*⟩⟩

G.3 Let M be a **DFA** over the alphabet Σ. Let M' be identical to M, except that accepting states in M are non-accepting in M' and vice versa. Each string in Σ^* is accepted by exactly one of M and M'.

G.4 Let M be an **NFA** over the alphabet Σ. Let M' be identical to M, except that accepting states in M are non-accepting in M' and vice versa. Each string in Σ^* is accepted by exactly one of M and M'.

G.5 If a language L is recognized by a DFA with n states, then the complementary language $\Sigma^* \setminus L$ is recognized by a DFA with at most $n+1$ states.

G.6 If a language L is recognized by an NFA with n states, then the complementary language $\Sigma^* \setminus L$ is recognized by a NFA with at most $n+1$ states.

G.7 If a language L is recognized by a DFA with n states, then L^* is recognized by a DFA with at most $n+1$ states.

G.8 If a language L is recognized by an NFA with n states, then L^* is recognized by a NFA with at most $n+1$ states.

Language Transformations

H.1 For every regular language L, the language $\left\{w^R \;\middle|\; w \in L\right\}$ is also regular.

H.2 For every language L, if the language $\left\{w^R \;\middle|\; w \in L\right\}$ is regular, then L is also regular. ⟪***F14***⟫

H.3 For every language L, if the language $\left\{w^R \;\middle|\; w \in L\right\}$ is not regular, then L is also not regular. ⟪***F14***⟫

H.4 For every regular language L, the language $\left\{w \;\middle|\; ww^R \in L\right\}$ is also regular.

H.5 For every regular language L, the language $\left\{ww^R \;\middle|\; w \in L\right\}$ is also regular.

H.6 For every language L, if the language $\left\{w \;\middle|\; ww^R \in L\right\}$ is regular, then L is also regular. *[Hint: Consider the language* $L = \{0^n 1 1 0^n \mid n \geq 0\}$*.]*

H.7 For every regular language L, the language $\left\{0^{|w|} \;\middle|\; w \in L\right\}$ is also regular.

H.8 For every language L, if the language $\left\{0^{|w|} \;\middle|\; w \in L\right\}$ is regular, then L is also regular.

H.9 For every context-free language L, the language $\left\{w^R \;\middle|\; w \in L\right\}$ is also context-free.

You have 120 minutes to answer five questions.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. Let compress0s(w) be a function that takes a string w as input, and returns the string formed by compressing every run of 0s in w by half. Specifically, every run of $2n$ 0s is compressed to length n, and every run of $2n+1$ 0s is compressed to length $n+1$. For example:

$$\begin{aligned}\text{compress0s}(\underline{00000}11\underline{0001}) &= \underline{000}11\underline{001}\\ \text{compress0s}(11\underline{000010}) &= 11\underline{00}1\underline{0}\\ \text{compress0s}(11111) &= 11111\end{aligned}$$

Let L be an arbitrary regular language.

   (a) ***Prove*** that $\left\{w \in \Sigma^* \;\middle|\; \text{compress0s}(w) \in L\right\}$ is regular.

   (b) ***Prove*** that $\left\{\text{compress0s}(w) \;\middle|\; w \in L\right\}$ is regular.

2. For each of the following languages L over the alphabet $\Sigma = \{0, 1\}$, describe a DFA that accepts L **and** give a regular expression that represents L. You do not need to justify your answers.

   (a) All strings in which at least one run has length divisible by 3.

   (b) All strings that do not contain either 100 or 011 as a substring.

3. Consider the following recursive function Bond, which doubles the length of any run of 0s in its input string.

$$\text{Bond}(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ 00 \cdot \text{Bond}(x) & \text{if } w = 0 \cdot x \text{ for some string } x \\ 1 \cdot \text{Bond}(x) & \text{if } w = 1 \cdot x \text{ for some string } x \end{cases}$$

   (a) ***Prove*** that $|\text{Bond}(w)| \geq |w|$ for all strings w.

   (b) ***Prove*** that $\text{Bond}(x \bullet y) = \text{Bond}(x) \bullet \text{Bond}(y)$ for all strings x and y.

   As usual, you can assume any result proved in class, in the lecture notes, in labs, or in homework solutions.

4. Let L be the language $\left\{ 0^a 1^b 0^c \;\middle|\; a = b \text{ or } a = c \text{ or } b = c \right\}$

   (a) ***Prove*** that L is *not* a regular language.

   (b) Describe a context-free grammar for L.

5. For each statement below, check "True" if the statement is always true and check "False" otherwise, and give a brief (one short sentence) explanation of your answer. Read these statements very carefully—small details matter!

   (a) If $2 + 2 = 5$, then zero is odd.

   (b) $\{0^n 1 \mid n > 0\}$ is the only infinite fooling set for the language $\{0^n 1 0^n \mid n > 0\}$.

   (c) $\{0^n 1 0^n \mid n > 0\}$ is a context-free language.

   (d) The context-free grammar $S \to 00S \mid S11 \mid 01$ generates the language $0^n 1^n$.

   (e) Every regular language is recognized by a DFA with exactly one accepting state.

   (f) Any language that can be decided by an NFA with ε-transitions can also be decided by an NFA without ε-transitions.

   (g) If L is a regular language over the alphabet $\{0, 1\}$, then $\{xy^C \mid x, y \in L\}$ is also regular.

   (h) If L is a regular language over the alphabet $\{0, 1\}$, then $\{ww^C \mid w \in L\}$ is also regular.

   (i) The regular expression $(00 + 11)^*$ represents the language of all strings over $\{0, 1\}$ of even length.

   (j) Let L_1, L_2 be two regular languages. The language $(L_1 + L_2)^*$ is also regular.

You have 120 minutes to answer five questions.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. For each of the following languages L over the alphabet $\Sigma = \{0, 1\}$, describe a DFA that accepts L **and** give a regular expression that represents L. You do not need to justify your answers.

   (a) All strings in which the number of runs is divisible by 3. (Recall that a *run* is a maximal non-empty substring where all symbols are equal.)

   (b) All strings that do not contain the substring 0110.

2. Let take2skip2(w) be a function takes an input string w and returns the subsequence of symbols at positions $1, 2, 5, 6, 9, 10, \ldots 4i+1, 4i+2, \ldots$ in w. In other words, take2skip2(w) takes the first two symbols of w, skip the next two, takes the next two, skips the next two, and so on. For example:

$$\text{take2skip2}(\underline{1}) = 1$$
$$\text{take2skip2}(\underline{01}0) = 01$$
$$\text{take2skip2}(\underline{01}\underline{0}0\underline{11}1\underline{00}\underline{1}1) = 0111001$$

   Let L be an arbitrary regular language.

   (a) ***Prove*** that the language $\{w \in \Sigma^* \mid \text{take2skip2}(w) \in L\}$ is regular.

   (b) ***Prove*** that the language $\{\text{take2skip2}(w) \mid w \in L\}$ is regular.

3. Consider the following recursive function censor, which deletes all 1s in its input string.

$$\text{censor}(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \text{censor}(x) & \text{if } w = 0 \cdot x \text{ for some string } x \\ 1 \cdot \text{censor}(x) & \text{if } w = 1 \cdot x \text{ for some string } x \end{cases}$$

   (a) ***Prove*** that $|\text{censor}(w)| \le |w|$ for all strings w.

   (b) ***Prove*** that $\text{censor}(\text{censor}(w)) = \text{censor}(w)$ for all strings w.

   As usual, you can assume any result proved in class, in the lecture notes, in labs, or in homework solutions.

4. Consider the language $L = \left\{ \mathtt{0}^a \mathtt{1}^b \;\middle|\; a > 2b \text{ or } 2a < b \right\}$

   (a) ***Prove*** that L is *not* a regular language.

   (b) Describe a context-free grammar for L.

5. For each statement below, check "True" if the statement is always true and check "False" otherwise, and give a brief (one short sentence) explanation of your answer. Read these statements very carefully—small details matter!

   (a) For every language L, the language L^* is infinite.

   (b) If a language L is finite, the complement of L is context-free.

   (c) The language $\{\mathtt{0}^{374n} \mid n \geq 374\}$ is regular.

   (d) The language $\left\{ wxw^R \;\middle|\; w, x \in \Sigma^* \right\}$ is regular.

   (e) The context-free grammar $S \to \mathtt{0}S\mathtt{1}S \mid S\mathtt{1}S\mathtt{0} \mid \varepsilon$ generates the set of all binary strings with the same number of $\mathtt{0}$s and $\mathtt{1}$s.

   (f) Every regular language is recognized by a DFA with at least 374 states.

   (g) If the languages L and L' are regular, their intersection $L \cap L'$ is also regular.

   (h) If a language has an infinite fooling set, then it is context-free.

   (i) Let M be a **DFA** over the alphabet Σ. Let M' be identical to M, except that accepting states in M are non-accepting in M' and vice versa. Each string in Σ^* is accepted by exactly one of M and M'.

   (j) Let M be an **NFA** over the alphabet Σ. Let M' be identical to M, except that accepting states in M are non-accepting in M' and vice versa. Each string in Σ^* is accepted by exactly one of M and M'.

Write your answers in the separate answer booklet.

You have 120 minutes (after you get the answer booklet) to answer five questions.

Please return this question sheet and your cheat sheet with your answers.

Questions 2 and 4 were swapped in the question sheet—but not in the answer booklet(!)—that was distributed during the exam.

1. For each statement below, check "True" if the statement is always true and check "False" otherwise, and give a brief (one short sentence) explanation of your answer. Read these statements very carefully—small details matter!

   (a) Every irregular language is infinite.

   (b) The language $(\mathtt{0} + \mathtt{1}(\mathtt{01}^*\mathtt{0})^*\mathtt{1})^*$ is context-free.

   (c) Every subset of a regular language is regular.

   (d) The language $\{\mathtt{0}^a\mathtt{1}^b \mid a + b \text{ is divisible by } 374\}$ is regular.

   (e) If language L is regular, then L has a finite fooling set.

   (f) For every language L, if for every string $w \in L$ there is a DFA that accepts w, then L is regular.

   (g) If language L is accepted by an NFA with n states, then its complement $\Sigma^* \setminus L$ is also accepted by an NFA with n states.

   (h) $\mathtt{0}^*\mathtt{1}^*$ is a fooling set for the language $\left\{\mathtt{0}^i\mathtt{1}^j\mathtt{0}^{i+j} \;\middle|\; i, j \geq 0\right\}$.

   (i) Every regular language is accepted by a DFA with an odd number of accepting states.

   (j) The context-free grammar $S \to \mathtt{1}T \mid T\mathtt{1} \mid \varepsilon;\ T \to \mathtt{0}S \mid S\mathtt{0}$ generates all strings in which the number of $\mathtt{0}$s equals the number of $\mathtt{1}$s.

2. Recall that a *run* in a string w is a maximal non-empty substring of w in which all symbols are equal. For example, the string $\underbrace{\mathtt{0}}\underbrace{\mathtt{111111}}\underbrace{\mathtt{000}}\underbrace{\mathtt{1}}\underbrace{\mathtt{0000}}$ consists of five runs.

   Let L be the set of all strings in $\{\mathtt{0}, \mathtt{1}\}^*$ in which every run of $\mathtt{0}$s is followed immediately by a longer run of $\mathtt{1}$s. For example, the strings $\mathtt{000111111011}$ and $\mathtt{11101100011111}$ and $\mathtt{11111}$ are in L, but the strings $\mathtt{00000111}$ and $\mathtt{0111000000}$ are not.

   (a) ***Prove*** that L is *not* a regular language.

   (b) Describe a context-free grammar for L.

3. For any string $w \in \{0,1\}^*$, let sortpairs(w) denote the string obtained by dividing w into pairs of symbols, sorting each pair into non-decreasing order, and leaving the last symbol if w has odd length. We can define sortpairs recursively as follows:

$$\text{sortpairs}(w) := \begin{cases} w & \text{if } w = \varepsilon \text{ or } w = 0 \text{ or } w = 1 \\ 01 \cdot \text{sortpairs}(x) & \text{if } w = 10x \text{ for some string } x \\ ab \cdot \text{sortpairs}(x) & \text{if } w = abx \text{ for some string } x \text{ and bits } a, b \text{ where } ab \neq 10 \end{cases}$$

For example,

$$\text{sortpairs}(\underbrace{00}\,\underbrace{10}\,\underbrace{11}\,\underbrace{10}\,\underbrace{01}\,1) = \underbrace{00}\,\underbrace{01}\,\underbrace{11}\,\underbrace{01}\,\underbrace{01}\,1$$

Recall that $\#(1,w)$ denotes the number 1s in the string w. For example, $\#(1,\varepsilon) = 0$ and $\#(1,00101100011) = 5$.

(a) ***Prove*** that $\#(1,\text{sortpairs}(w)) = \#(1,w)$ for every string w.

(b) ***Prove*** that $\text{sortpairs}(\text{sortpairs}(w)) = \text{sortpairs}(w)$ for every string w.

As usual, you can assume any result proved in class, in the lecture notes, in labs, in lab solutions, or in homework solutions. In particular, you may use the fact that $\#(1,xy) = \#(1,x) + \#(1,y)$ for all strings x and y.

4. Recall that a *run* in a string w is a maximal non-empty substring of w in which all symbols are equal. For example, the string $\underbrace{0}\,\underbrace{111111}\,\underbrace{000}\,\underbrace{1}\,\underbrace{0000}$ consists of five runs.

For any string $w \in \{0,1\}^*$, let compact(w) denote the string obtained by replacing each run with a single symbol from that run. For example, compact$(\varepsilon) = \varepsilon$ and

$$\text{compact}(011111100010000) = 01010.$$

Let L be an arbitrary regular language.

(a) ***Prove*** that the language $\textsc{Compact}(L) = \{\text{compact}(w) \mid w \in L\}$ is regular.

(b) ***Prove*** that the language $\textsc{Uncompact}(L) = \{w \in \Sigma^* \mid \text{compact}(w) \in L\}$ is regular.

5. For each of the following languages L over the alphabet $\Sigma = \{0,1\}$, describe a DFA that accepts L **and** give a regular expression that represents L. You do not need to justify your answers.

(a) Strings that do not contain the substring 01110.

(b) Strings that contain at least one odd-length run of 0s that is followed immediately by an even-length run of 1s.

Write your answers in the separate answer booklet.

You have 120 minutes (after you get the answer booklet) to answer five questions.

Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, check "Yes" if the statement is always true and check "No" otherwise, and give a brief (one short sentence) explanation of your answer. Read these statements very carefully—small details matter!

   (a) Every infinite language is regular.

   (b) The language $(\mathtt{0} + \mathtt{1}(\mathtt{01}^*\mathtt{0})^*\mathtt{1})^*$ is not context-free.

   (c) Every subset of an irregular language is irregular.

   (d) The language $\left\{\mathtt{0}^a\mathtt{1}^b \;\middle|\; a - b \text{ is divisible by } 374\right\}$ is regular.

   (e) If language L is not regular, then L has a finite fooling set.

   (f) If there is a DFA that rejects every string in language L, then L is regular.

   (g) If language L is accepted by an DFA with n states, then its complement $\Sigma^* \setminus L$ is also accepted by a DFA with n states.

   (h) $\mathtt{1}^*\mathtt{0}^*$ is a fooling set for the language $\left\{\mathtt{1}^i\mathtt{0}^{i+j}\mathtt{1}^j \;\middle|\; i, j \geq 0\right\}$.

   (i) Every regular language is accepted by a DFA with an odd number of accepting states.

   (j) The context-free grammar $S \to \varepsilon \mid \mathtt{0}S\mathtt{1}S \mid \mathtt{1}S\mathtt{0}S$ generates all strings in which the number of $\mathtt{0}$s equals the number of $\mathtt{1}$s.

2. For any string w, let $\mathsf{cycleleft}(w)$ denote the string obtained by moving the first symbol of w (if any) to the end. More formally:

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

   For example, $\mathsf{cycleleft}(\mathtt{001111}) = \mathtt{011110}$.

   Let L be an arbitrary regular language.

   (a) Prove that $\textsc{CycleLeft}(L) = \{\mathsf{cycleleft}(w) \mid w \in L\}$ is a regular language.

   (b) Prove that $\textsc{CycleRight}(L) = \{w \in \Sigma^* \mid \mathsf{cycleleft}(w) \in L\}$ is a regular language.

3. For any string $w \in \{0,1\}^*$, let squish(w) denote the string obtained by dividing w into pairs of symbols, replacing each pair with 0 if the symbols are equal and 1 otherwise, and keeping the last symbol if w has odd length. We can define sortpairs recursively as follows:

$$\text{squish}(w) := \begin{cases} w & \text{if } w = \varepsilon \text{ or } w = 0 \text{ or } w = 1 \\ 0 \cdot \text{squish}(x) & \text{if } w = 00x \text{ or } w = 11x \text{ for some string } x \\ 1 \cdot \text{squish}(x) & \text{if } w = 01x \text{ or } w = 10x \text{ for some string } x \end{cases}$$

For example,

$$\text{squish}(\underbrace{00}\,\underbrace{10}\,\underbrace{11}\,\underbrace{10}\,\underbrace{01}\,1) = 010111$$

(a) ***Prove*** that $\#(1, \text{squish}(w)) \le \#(1, w)$ for every string w.

(b) ***Prove*** that $\#(1, \text{squish}(w))$ is even if and only if $\#(1, w)$ is even (or equivalently, that $\#(1, \text{squish}(w)) \bmod 2 = \#(1, w) \bmod 2$) for every string w.

As usual, you can assume any result proved in class, in the lecture notes, in labs, in lab solutions, or in homework solutions. In particular, you may use the fact that $\#(1, xy) = \#(1, x) + \#(1, y)$ for all strings x and y.

4. Let L be the set of all strings in $\{0,1\}^*$ in which every run of 0s is followed immediately by a *shorter* run of 1s. For example, the strings 0001100000111 and 11100100000111 and 11111 are in L, but the strings 00011111 and 000110000 are not.

(a) ***Prove*** that L is *not* a regular language.

(b) Describe a context-free grammar for L.

5. For each of the following languages L over the alphabet $\Sigma = \{0,1\}$, describe a DFA that accepts L **and** give a regular expression that represents L. You do not need to justify your answers.

(a) Strings that do not contain the subsequence 01110.

(b) Strings that contain at least two even-length runs of 1s.

CS/ECE 374 A ✧ Fall 2023

Midterm 2 Study Questions

This is a "core dump" of potential questions for Midterm 2. This should give you a good idea of the *types* of questions that we will ask on the exam, but the actual exam questions may or may not appear in this list. This list intentionally includes a few questions that are too long or difficult for exam conditions; *most* of these are indicated with a *star.

Questions from Jeff's past exams are labeled with the semester they were used, for example, ⟨⟨*S18*⟩⟩ or ⟨⟨*F19*⟩⟩. Questions from this semester's homework are labeled ⟨⟨*HW*⟩⟩. Questions from this semester's labs are labeled ⟨⟨*Lab*⟩⟩. Some unflagged questions may have been used in exams by other instructors.

How to Use These Problems

Solving every problem in this handout is ***not*** the best way to study for the exam. Memorizing the solutions to every problem in this handout is the ***absolute worst*** way to study for the exam.

What we recommend instead is to work on a *sample* of the problems. Choose one or two problems at random from each section and try to solve them from scratch under exam conditions—by yourself, in a quiet room, with a 30-minute timer, *without* your notes, *without* the internet, and if possible, even without your cheat sheet. If you're comfortable solving a few problems in a particular section, you're probably ready for that type of problem on the exam. Move on to the next section.

Discussing problems with other people (in your study groups, in the review sessions, in office hours, or on Piazza) and/or looking up old solutions can be *extremely* helpful, but ***only after*** you have (1) made a good-faith effort to solve the problem on your own, and (2) you have either a candidate solution or some idea about where you're getting stuck.

If you find yourself getting stuck on a particular type of problem, try to figure out *why* you're stuck. Do you understand the problem statement? Have you tried several example inputs to see what the correct output *should* be? Are you stuck on choosing the right high-level approach, are you stuck on the details, or are you struggling to express your ideas clearly?

Similarly, if feedback suggests that your solutions to a particular type of problem are incorrect or incomplete, try to figure out what you missed. For recursion/dynamic programming: Are you solving the right recursive generalization of the stated problem? Are you having trouble writing a specification of the function, as opposed to a description of the algorithm? Are you struggling to find a good evaluation order? Are you trying to use a greedy algorithm? *[Hint: Don't.]* For graph algorithms: Are you aiming for the right problem? Are you having trouble figuring out the interesting states of the problem (otherwise known as vertices) and the transitions between them (otherwise known as edges)? Do you keep trying to modify the algorithm instead of modifying the graph? *[Hint: Don't.]*

Remember that your goal is *not* merely to "understand" the solution to any particular problem, but to become more comfortable with solving a certain *type* of problem on your own. **"Understanding" is a trap; aim for mastery.** If you can identify specific steps that you find problematic, read more *about those steps*, focus your practice *on those steps*, and try to find helpful information *about those steps* to write on your cheat sheet. Then work on the next problem!

Short answers

1. Solve the recurrence $T(n) = 3T(n/2) + O(n^2)$.
2. Solve the recurrence $T(n) = 7T(n/2) + O(n^2)$.
3. Solve the recurrence $T(n) = 4T(n/2) + O(n^2)$.
4. Solve the recurrence $T(n) = 2T(n/3) + O(\sqrt{n})$.
5. Solve the recurrence $T(n) = 2T(n/7) + O(\sqrt{n})$.
6. Solve the recurrence $T(n) = 2T(n/4) + O(\sqrt{n})$.
7. Solve the recurrence $T(n) = 16\,T(n/2) + O(n^3)$.
8. Solve the recurrence $T(n) = 16\,T(n/2) + O(n^7)$.
9. Solve the recurrence $T(n) = 16\,T(n/2) + O(n^4)$.
10. Describe an appropriate memoization structure and evaluation order for the following (meaningless) recurrence, and give the running time of the resulting iterative algorithm to compute $Reeee(1, n)$. (Assume all array accesses are legal.)

$$Reeee(i,k) = \begin{cases} 0 & \text{if } i > k \\ A[i] & \text{if } i = k \\ Reeee(i+1,k-1) + A[i] + A[k] & \text{if } A[i] = A[k] \\ \max \left\{ \begin{array}{c} Reeee(i+2,k), \\ Reeee(i+1,k-1), \\ Reeee(i,k-2) \end{array} \right\} & \text{otherwise} \end{cases}$$

11. Describe an appropriate memoization structure and evaluation order for the following (meaningless) recurrence, and give the running time of the resulting iterative algorithm to compute $Huh(1, n)$.

$$Huh(i,k) = \begin{cases} 0 & \text{if } i > n \text{ or } k < 0 \\ \min \left\{ \begin{array}{c} Huh(i+1,k-2), \\ Huh(i+2,k-1) \end{array} \right\} + A[i,k] & \text{if } A[i,k] \text{ is even} \\ \min \left\{ \begin{array}{c} Huh(i+1,k-2), \\ Huh(i+2,k-1) \end{array} \right\} - A[i,k] & \text{if } A[i,k] \text{ is odd} \end{cases}$$

12. Describe an appropriate memoization structure and evaluation order for the following (meaningless) recurrence, and give the running time of the resulting iterative algorithm to compute $Spoopy(1, 1)$.

$$Spoopy(i,k) = \begin{cases} 0 & \text{if } i > n \text{ or } k > n \\ \min \left\{ \begin{array}{l} Spoopy(i,k+j) \\ \quad + (k-i) \cdot A[j] \\ \quad + Spoopy(i+j,k) \end{array} \middle| \; 1 \le j \le n \right\} & \text{otherwise} \end{cases}$$

13. Describe an appropriate memoization structure and evaluation order for the following (meaningless) recurrence, and give the running time of the resulting iterative algorithm to compute $What(1, n)$.

$$What(i,j) = \begin{cases} 0 & \text{if } i > n \text{ or } j < 0 \\ \max \begin{Bmatrix} What(i, j+1) \\ What(i-1, j) \\ A[i] \cdot A[j] + What(i-1, j+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

Recursion and Dynamic Programming

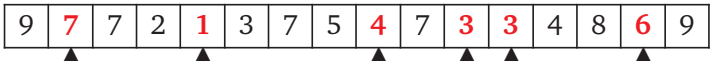
Elementary Recursion/Divide and Conquer

1. ⟨⟨*Lab*⟩⟩

   (a) Suppose $A[1..n]$ is an array of n distinct integers, sorted so that $A[1] < A[2] < \cdots < A[n]$. Each integer $A[i]$ could be positive, negative, or zero. Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists..

   (b) Now suppose $A[1..n]$ is a sorted array of n distinct ***positive*** integers. 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. ⟨⟨*Lab*⟩⟩ 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:



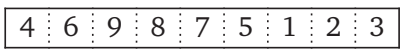
   Describe and analyze a fast algorithm that returns the index of one local minimum. For example, given the array above, your algorithm could return the integer 5, because $A[5]$ is a local minimum. *[Hint: With the given boundary conditions, any array* ***must*** *contain at least one local minimum. Why?]*

3. ⟨⟨*Lab*⟩⟩ 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

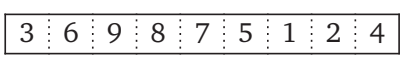
$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1..8] = [2, 4, 5, 8, 17, 19, 21, 23]$$

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

4. ⟨⟨*F14, S14*⟩⟩ An array $A[0..n-1]$ of n distinct numbers is ***bitonic*** if there are unique indices i and j such that $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ and $A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,



   is bitonic, but



   is *not* bitonic.

   Describe and analyze an algorithm to find the index of the *smallest* element in a given bitonic array $A[0..n-1]$ in $O(\log n)$ time. You may assume that the numbers in the input

array are distinct. For example, given the first array above, your algorithm should return 6, because $A[6] = 1$ is the smallest element in that array.

5. ⟨⟨***F16***⟩⟩ Suppose you are given a sorted array $A[1..n]$ of distinct numbers that has been *rotated* k steps, for some ***unknown*** integer k between 1 and $n-1$. That is, the prefix $A[1..k]$ is sorted in increasing order, the suffix $A[k+1..n]$ is sorted in increasing order, and $A[n] < A[1]$. For example, you might be given the following 16-element array (where $k = 10$):

9	13	16	18	19	23	28	31	37	42	−4	0	2	5	7	8

Describe and analyze an algorithm to determine if the given array contains a given number x. The input to your algorithm is the array $A[1..n]$ and the number x; your algorithm is ***not*** given the integer k.

6. ⟨⟨***S22***⟩⟩ Suppose you are given an array $A[1..n]$ of distinct numbers that contains exactly one local maximum. That is, for some ***unknown*** index k, the prefix $A[1..k]$ is sorted in increasing order, and the suffix $A[k..n]$ is sorted in decreasing order. You are also given a real number x. Describe and analyze an algorithm to determine whether $A[i] = x$ for any index i.

7. ⟨⟨***F16***⟩⟩ Suppose you are given two unsorted arrays $A[1..n]$ and $B[1..n]$ containing $2n$ distinct integers, such that $A[1] < B[1]$ and $A[n] > B[n]$. Describe and analyze an efficient algorithm to compute an index i such that $A[i] < B[i]$ and $A[i+1] > B[i+1]$. *[Hint: Why does such an index i always exist?]*

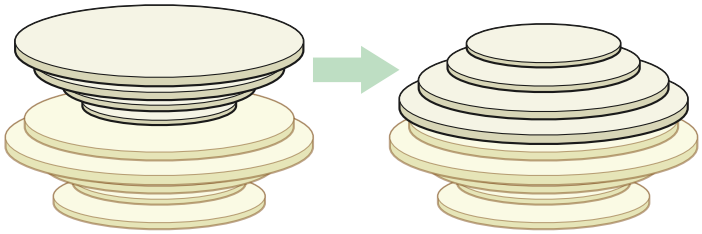
8. (a) Describe an algorithm to determine in $O(n)$ time whether an arbitrary array $A[1..n]$ contains more than $n/4$ copies of any value.

   (b) Describe and analyze an algorithm to determine, given an arbitrary array $A[1..n]$ and an integer k, whether A contains more than k copies of any value. Express the running time of your algorithm as a function of both n and k.

   Do not use hashing, or radix sort, or any other method that depends on the precise input values, as opposed to their order.

9. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a *flip*—insert a spatula under the top k pancakes, for some integer k between 1 and n, and flip them all over.

   (a) Describe an algorithm to sort an arbitrary stack of n pancakes using as few flips as possible. *Exactly* how many flips does your algorithm perform in the worst case?

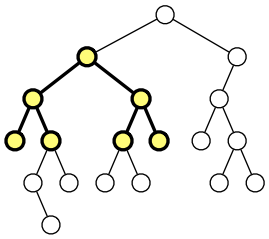


Flipping the top four pancakes.

(b) Now suppose one side of each pancake is burned. Describe an algorithm to sort an arbitrary stack of n pancakes, so that the burned side of every pancake is facing down, using as few flips as possible. *Exactly* how many flips does your algorithm perform in the worst case?

[Hint: This problem has ***nothing*** *to do with the Tower of Hanoi!]*

10. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return both the root and the depth of this subtree.



The largest complete subtree of this binary tree has depth 2.

*11. ⟨⟨*S18*⟩⟩ Suppose you have an integer array $A[1..n]$ that *used* to be sorted, but Swedish hackers have overwritten k entries of A with random numbers. Because you carefully monitor your system for intrusions, you know *how many* entries of A are corrupted, but not *which* entries or what the values are.

Describe an algorithm to determine whether your corrupted array A contains an integer x. Your input consists of the array A, the integer k, and the target integer x. For example, if A is the following array, $k = 4$, and $x = 17$, your algorithm should return TRUE. (The corrupted entries of the array are shaded.)

2	3	99	7	11	13	17	19	25	29	31	−5	41	43	47	53	8	61	67	71

Assume that x is not equal to any of the the corrupted values, and that all n array entries are distinct. Report the running time of your algorithm as a function of n and k. A solution only for the special case $k = 1$ is worth 5 points; a complete solution for arbitrary k is worth 10 points. *[Hint: First consider $k = 0$; then consider $k = 1$.]*

12. ⟨⟨*F21*⟩⟩ Your grandmother dies and leaves you her treasured collection of n radioactive Beanie Babies. Her will reveals that one of the Beanie Babies is a rare specimen worth 374 million dollars, but all the others are worthless. The valuable Beanie Baby is either slightly more or slightly less radioactive than the others, but you don't know which. Otherwise, as far as you can tell, they are all identical.

    You have access to a state-of-the-art Radiation Comparator at your job. The Comparator has two chambers. You can place any two disjoint sets of Beanie Babies in Comparator's two chambers; the Detector will then indicate which of the two subsets emits more radiation, or that the two subsets are equally radioactive. (The two subsets are equally radioactive if and only if they contain the same number of Beanie Babies, and they are all worthless.) The Comparator is slow and consumes a *lot* of power, and you really aren't supposed to use it for personal projects, so you *really* want to use it as few times as possible.

    Describe an efficient algorithm to identify the valuable Beanie Baby. How many times does your algorithm use the Comparator in the worst case, as a function of n?

Dynamic Programming

1. ⟨⟨*Lab*⟩⟩ Describe and analyze efficient algorithms for the following problems.

   (a) 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 $i \geq 2$.

   (b) 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 $i \geq 2$.

   (c) 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 $i \geq 3$.

   (d) 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 $i \geq 3$.

   (e) 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.

2. ⟨⟨*F19*⟩⟩

   (a) Recall that a *palindrome* is any string that is equal to its reversal, like REDIVIDER or POOP. Describe an algorithm to find the length of the longest subsequence of a given string that is a palindrome.

   (b) A *double palindrome* is the concatenation of two *non-empty* palindromes, like AREDIVIDER or POOPPOOP. Describe an algorithm to find the length of the longest subsequence of a given string that is a *double* palindrome. *[Hint: Use your algorithm from part (a).]*

   For both algorithms, the input is an array $A[1..n]$, and the output is an integer. For example, given the string MAYBEDYNAMICPROGRAMMING as input, your algorithm for part (a) should return 7 (for the palindrome subsequence NMRORMN), and your algorithm for part (b) should return 12 (for the double palindrome subsequence MAYBYAMIRORI).

3. Recall that a *palindrome* is any string that is the same as its reversal. For example, I, DAD, HANNAH, AIBOHPHOBIA (fear of palindromes), and the empty string are all palindromes.

   (a) ⟨⟨*S14*⟩⟩ Describe and analyze an algorithm to find the length of the longest substring (not *subsequence*!) of a given input string that is a palindrome. For example, BASEESAB is the longest palindrome substring of BUBBASEESABANANA ("Bubba sees a banana."). Thus, given the input string BUBBASEESABANANA, your algorithm should return the integer 8.

   (b) ⟨⟨*Lab, F16*⟩⟩ Describe and analyze an algorithm to find the length of the longest subsequence (not *substring*!) of a given input string that is a palindrome. For example, the longest palindrome subsequence of MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM

is MHYMRORMYHM, so given that string as input, your algorithm should output the number 11.

(c) ⟨⟨*S14*⟩⟩ Any string can be decomposed into a sequence of palindrome substrings. For example, the string BUBBASEESABANANA can be broken into palindromes in the following ways (and many others):

$$\mathtt{BUB} + \mathtt{BASEESAB} + \mathtt{ANANA}$$
$$\mathtt{B} + \mathtt{U} + \mathtt{BB} + \mathtt{A} + \mathtt{SEES} + \mathtt{ABA} + \mathtt{NAN} + \mathtt{A}$$
$$\mathtt{B} + \mathtt{U} + \mathtt{BB} + \mathtt{A} + \mathtt{SEES} + \mathtt{A} + \mathtt{B} + \mathtt{ANANA}$$
$$\mathtt{B} + \mathtt{U} + \mathtt{B} + \mathtt{B} + \mathtt{A} + \mathtt{S} + \mathtt{E} + \mathtt{E} + \mathtt{S} + \mathtt{A} + \mathtt{B} + \mathtt{A} + \mathtt{N} + \mathtt{A} + \mathtt{N} + \mathtt{A}$$

Describe and analyze an algorithm to find the smallest number of palindromes that make up a given input string. For example:

- Given the string BUBBASEESABANANA, your algorithm should return the integer 3.
- Given the string PALINDROME, your algorithm should return the integer 10.
- Given the string RACECAR, your algorithm should return the integer 1.

(d) A ***metapalindrome*** is a decomposition of a string into a sequence of non-empty palindromes, such that the sequence of palindrome lengths is itself a palindrome. For example, the decomposition

$$\mathtt{BUB} \bullet \mathtt{B} \bullet \mathtt{ALA} \bullet \mathtt{SEES} \bullet \mathtt{ABA} \bullet \mathtt{N} \bullet \mathtt{ANA}$$

is a metapalindrome for the string BUBBALASEESABANANA, with the palindromic length sequence $(3, 1, 3, 4, 3, 1, 3)$. Describe and analyze an efficient algorithm to find the length of the shortest metapalindrome for a given string. For example:

- Given the string BUBBALASEESABANANA, your algorithm should return the integer 7.
- Given the string PALINDROME, your algorithm should return the integer 10.
- Given the string DEPOPED, your algorithm should return the integer 1.

4. ⟨⟨*F16*⟩⟩ It's almost time to show off your flippin' sweet dancing skills! Tomorrow is the big dance contest you've been training for your entire life, except for that summer you spent with your uncle in Alaska hunting wolverines. You've obtained an advance copy of the the list of n songs that the judges will play during the contest, in chronological order.

You know all the songs, all the judges, and your own dancing ability extremely well. For each integer k, you know that if you dance to the kth song on the schedule, you will be awarded exactly $Score[k]$ points, but then you will be physically unable to dance for the next $Wait[k]$ songs (that is, you cannot dance to songs $k+1$ through $k + Wait[k]$). The dancer with the highest total score at the end of the night wins the contest, so you want your total score to be as high as possible.

Describe and analyze an efficient algorithm to compute the maximum total score you can achieve. The input to your sweet algorithm is the pair of arrays $Score[1..n]$ and $Wait[1..n]$.

5. ⟨⟨***S16***⟩⟩ After the Revolutionary War, Alexander Hamilton's biggest rival as a lawyer was Aaron Burr. (Sir!) In fact, the two worked next door to each other. Unlike Hamilton, Burr cannot work non-stop; every case he tries exhausts him. The bigger the case, the longer he must rest before he is well enough to take the next case. (Of course, he is willing to wait for it.) If a case arrives while Burr is resting, Hamilton snatches it up instead.

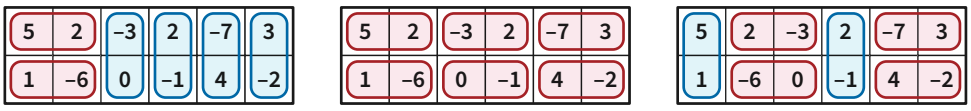
   Burr has been asked to consider a sequence of n upcoming cases. He quickly computes two arrays $profit[1..n]$ and $skip[1..n]$, where for each index i,

   - $profit[i]$ is the amount of money Burr would make by taking the ith case, and
   - $skip[i]$ is the number of consecutive cases Burr must skip if he accepts the ith case. That is, if Burr accepts the ith case, he cannot accept cases $i+1$ through $i+skip[i]$.

   Design and analyze an algorithm that determines the maximum total profit Burr can secure from these n cases, using his two arrays as input.

6. ⟨⟨***F21***⟩⟩ Suppose you are asked to tile a $2 \times n$ grid of squares with dominos (1×2 rectangles). Each domino must cover exactly two grid squares, either horizontally or vertically, and each grid square must be covered by exactly one domino.

   Each grid square is worth some number of points, which could be positive, negative, or zero. The ***total value*** of a domino tiling is the sum of the points in squares covered by vertical dominos, *minus* the sum of the points in squares covered by horizontal dominos. The following figure shows three examples of domino tilings of the same 2×6 grid, along with their total values.



   Describe and analyze an efficient algorithm to compute the largest possible value of a domino tiling of a given $2 \times n$ grid. Your input is an array $Points[1..2, 1..n]$ of point values. For example, given the grid shown above, your algorithm should return the integer 16.

7. ⟨⟨***F16***⟩⟩ A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways.

$$\texttt{BANANA}_{\texttt{ANANAS}} \qquad \texttt{BAN}_{\texttt{ANA}}{}^{\texttt{ANA}}{}_{\texttt{NAS}} \qquad \texttt{B}_{\texttt{AN}}{}^{\texttt{AN}}{}_{\texttt{A}}{}^{\texttt{A}}{}_{\texttt{NA}}{}^{\texttt{NA}}{}_{\texttt{S}}$$

   Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of DYNAMIC and PROGRAMMING:

$$\texttt{PRO}^{\texttt{D}}{}_{\texttt{G}}{}^{\texttt{Y}}{}_{\texttt{R}}{}^{\texttt{NAM}}{}_{\texttt{AMMI}}{}^{\texttt{I}}{}_{\texttt{N}}{}^{\texttt{C}}{}_{\texttt{G}} \qquad \texttt{DY}_{\texttt{PRO}}{}^{\texttt{N}}{}_{\texttt{G}}{}^{\texttt{A}}{}_{\texttt{R}}{}^{\texttt{M}}{}_{\texttt{AMM}}{}^{\texttt{IC}}{}_{\texttt{ING}}$$

   Describe and analyze an efficient algorithm to determine, given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, whether C is a shuffle of A and B.

8. ⟨⟨*F21*⟩⟩ Suppose we are given an n-digit integer X. Repeatedly remove one digit from either end of X (your choice) until no digits are left. The *square-depth* of X is the maximum number of perfect squares that you can see during this process. For example, the number 32492 has square-depth 3, by the following sequence of removals:

$$32492 \to \overset{57^2}{3249} \to \overset{18^2}{324} \to 24 \to \overset{2^2}{4} \to \varepsilon.$$

Describe and analyze an algorithm to compute the square-depth of a given integer X, represented as an array $X[1..n]$ of n decimal digits. Assume you have access to a subroutine IsSquare that determines whether a given k-digit number (represented by an array of digits) is a perfect square ***in $O(k^2)$ time***.

9. Suppose you are given a sequence of non-negative integers separated by $+$ and $\times$ signs; for example:

$$2 \times 3 + 0 \times 6 \times 1 + 4 \times 2$$

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

$$2 \times (3 + (0 \times (6 \times (1 + (4 \times 2))))) = 6$$
$$(((((2 \times 3) + 0) \times 6) \times 1) + 4) \times 2 = 80$$
$$((2 \times 3) + (0 \times 6)) \times (1 + (4 \times 2)) = 108$$
$$(((2 \times 3) + 0) \times 6) \times ((1 + 4) \times 2) = 360$$

Describe and analyze an algorithm to compute, given a list of integers separated by $+$ and $\times$ signs, the smallest possible value we can obtain by inserting parentheses.

Your input is an array $A[0..2n]$ where each $A[i]$ is an integer if i is even and $+$ or $\times$ if i is odd. Assume any arithmetic operation in your algorithm takes $O(1)$ time.

10. Suppose you are given three strings $A[1..n]$, $B[1..n]$, and $C[1..n]$.

    (a) Describe and analyze an algorithm to find the length of the longest common subsequence of all three strings. For example, given the input strings

    $$A = \texttt{AxxBxxCDxEF}, \qquad B = \texttt{yyABCDyEyFy}, \qquad C = \texttt{zAzzBCDzEFz},$$

    your algorithm should output the number 6, which is the length of the longest common subsequence `ABCDEF`.

    (b) Describe and analyze an algorithm to find the length of the shortest common supersequence of all three strings. For example, given the input strings

    $$A = \texttt{AxxBxxCDxEF}, \qquad B = \texttt{yyABCDyEyFy}, \qquad C = \texttt{zAzzBCDzEFz},$$

    your algorithm should output the number 21, which is the length of the shortest common supersequence `yzyAxzzxBxxCDxyzEyFzy`.

11. (a) Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

    (b) Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which ***every*** pair of segments intersects.

12. ⟨⟨*S18*⟩⟩ Suppose we want to split an array $A[1..n]$ of integers into k contiguous intervals that partition the sum of the values as evenly as possible. Specifically, define the *cost* of such a partition as the maximum, over all k intervals, of the sum of the values in that interval; our goal is to minimize this cost. Describe and analyze an algorithm to compute the minimum cost of a partition of A into k intervals, given the array A and the integer k as input.

    For example, given the array $A = [1, 6, -1, 8, 0, 3, 3, 9, 8, 8, 7, 4, 9, 8, 9, 4, 8, 4, 8, 2]$ and the integer $k = 3$ as input, your algorithm should return the integer 37, which is the cost of the following partition:

$$[\overbrace{1, 6, -1, 8, 0, 3, 3, 9, 8}^{37} \mid \overbrace{8, 7, 4, 9, 8}^{36} \mid \overbrace{9, 4, 8, 4, 8, 2}^{35}]$$

    The numbers above each interval show the sum of the values in that interval.

13. ⟨⟨*S18,HW*⟩⟩ The City Council of Sham-Poobanana needs to partition Purple Street into voting districts. A total of n people live on Purple Street, at consecutive addresses $1, 2, \ldots, n$. Each voting district must be a contiguous interval of addresses $i, i+1, \ldots, j$ for some $1 \le i < j \le n$. By law, each Purple Street address must lie in exactly one district, and the number of addresses in each district must be between k and $2k$, where k is some positive integer parameter.

    Every election in Sham-Poobanana is between two rival factions: Oceania and Eurasia. A majority of the City Council are from Oceania, so they consider a district to be *good* if more than half the residents of that district voted for Oceania in the previous election. Naturally, the City Council has complete voting records for all n residents.

    For example, the figure below shows a legal partition of 22 addresses into 4 good districts and 3 bad districts, where $k = 2$ (so each district contains either 2, 3, or 4 addresses). Each O indicates a vote for Oceania, and each X indicates a vote for Eurasia.

    Describe an algorithm to find the largest possible number of *good* districts in a legal partition. Your input consists of the integer k and a boolean array $GoodVote[1..n]$ indicating which residents previously voted for Oceania (TRUE) or Eurasia (FALSE). You can assume that a legal partition exists. Analyze the running time of your algorithm in terms of the parameters n and k.

14. Suppose you are given an $m \times n$ bitmap, represented by an array $M[1..n, 1..n]$ of 0s and 1s. A *solid square block* in M is a subarray of the form $M[i..i+w, j..j+w]$ containing only 1-bits. Describe and analyze an algorithm to find the largest solid square block in M.

15. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

    Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it's his turn, Elmo *always* takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

    (a) Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do *not* follow the same greedy strategy as Elmo.

    (b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

    (c) Five years later, thirteen-year-old Elmo has become a *much* stronger player. Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against a *perfect* opponent.

16. ⟨⟨*S16*⟩⟩ Your nephew Elmo is visiting you for Christmas, and he's brought a different card game. Like your previous game with Elmo, this game is played with a row of n cards, each labeled with an integer (which could be positive, negative, or zero). Both players can see all n card values. Otherwise, the game is almost completely different.

    On each turn, the current player must take the leftmost card. The player can either keep the card or give it to their opponent. If they keep the card, their turn ends and their opponent takes the next card; however, if they give the card to their opponent, the current player's turn continues with the next card. In short, the player that does *not* get the ith card decides who gets the $(i+1)$th card. The game ends when all cards have been played. Each player adds up their card values, and whoever has the higher total wins.

    For example, suppose the initial cards are $[3, -1, 4, 1, 5, 9]$, and Elmo plays first. Then the game might proceed as follows:

    - Elmo keeps the 3, ending his turn.
    - You give Elmo the -1.
    - You keep the 4, ending your turn.
    - Elmo gives you the 1.
    - Elmo gives you the 5.

- Elmo keeps the 9, ending his turn. All cards are gone, so the game is over.
- Your score is $1+4+5=10$ and Elmo's score is $3-1+9=11$, so Elmo wins.

Describe an algorithm to compute the highest possible score you can earn from a given row of cards, assuming Elmo plays first and plays perfectly. Your input is the array $C[1..n]$ of card values. For example, if the input is $[3,-1,4,1,5,9]$, your algorithm should return the integer 10.

17. ⟨⟨*F14*⟩⟩ The new mobile game *Candy Swap Saga XIII* involves n cute animals numbered 1 through n. Each animal holds one of three types of candy: circus peanuts, Heath bars, and Cioccolateria Gardini chocolate truffles. You also have a candy in your hand; at the start of the game, you have a circus peanut.

To earn points, you visit each of the animals in order from 1 to n. For each animal, you can either keep the candy in your hand or exchange it with the candy the animal is holding.

- If you swap your candy for another candy of the *same* type, you earn one point.
- If you swap your candy for a candy of a *different* type, you lose one point. (Yes, your score can be negative.)
- If you visit an animal and decide not to swap candy, your score does not change.

You *must* visit the animals in order, and once you visit an animal, you can never visit it again.

Describe and analyze an efficient algorithm to compute your maximum possible score. Your input is an array $C[1..n]$, where $C[i]$ is the type of candy that the ith animal is holding.

18. ⟨⟨*F14*⟩⟩ Farmers Boggis, Bunce, and Bean have set up an obstacle course for Mr. Fox. The course consists of a row of n booths, each with an integer painted on the front with bright red paint, which could be positive, negative, or zero. Let $A[i]$ denote the number painted on the front of the ith booth. Everyone has agreed to the following rules:

- At each booth, Mr. Fox ***must*** say either "Ring!" or "Ding!".
- If Mr. Fox says "Ring!" at the ith booth, he earns a reward of $A[i]$ chickens. (If $A[i]<0$, Mr. Fox pays a penalty of $-A[i]$ chickens.)
- If Mr. Fox says "Ding!" at the ith booth, he pays a penalty of $A[i]$ chickens. (If $A[i]<0$, Mr. Fox earns a reward of $-A[i]$ chickens.)
- Mr. Fox is forbidden to say the same word more than three times in a row. For example, if he says "Ring!" at booths 6, 7, and 8, then he *must* say "Ding!" at booth 9.
- All accounts will be settled at the end; Mr. Fox does not actually have to carry chickens through the obstacle course.
- If Mr. Fox violates any of the rules, or if he ends the obstacle course owing the farmers chickens, the farmers will shoot him.

Describe and analyze an algorithm to compute the largest number of chickens that Mr. Fox can earn by running the obstacle course, given the array $A[1..n]$ of booth numbers as input.

19. ⟨⟨***F19***⟩⟩ Satya is in charge of establishing a new testing center for the Standardized Awesomeness Test (SAT), and he found an old conference hall that is perfect. The conference hall has n rooms of various sizes along a single long hallway, numbered in order from 1 through n. Satya knows exactly how many students fit into each room, and he wants to use a subset of the rooms to host as many students as possible for testing.

Unfortunately, there have been several incidents of students cheating at other testing centers by tapping secret codes through walls. To prevent this type of cheating, Satya can use two adjacent rooms only if he demolishes the wall between them. For example, if Satya wants to use rooms 1, 3, 4, 5, 7, 8, and 10, he must demolish three walls: between rooms 3 and 4, between rooms 4 and 5, and between rooms 7 and 8.

(a) ⟨⟨***HW***⟩⟩ The city's chief architect has determined that demolishing the walls on both sides of the same room would threaten the building's structural integrity. For this reason, Satya can never host students in three consecutive rooms.

Describe an efficient algorithm that computes the largest number of students that Satya can host for testing without using three consecutive rooms.

The input to your algorithm is an array $S[1..n]$, where each $S[i]$ is the (non-negative integer) number of students that can fit in room i.

(b) The city's chief architect has determined that demolishing more than k walls would threaten the structural integrity of the building.

Describe an efficient algorithm that computes the largest number of students that Satya can host for testing without demolishing more than k walls.

The input to your algorithm is the integer k and an array $S[1..n]$, where each $S[i]$ is the (non-negative integer) number of students that can fit in room i. Analyze your algorithm as a function of both n and k.

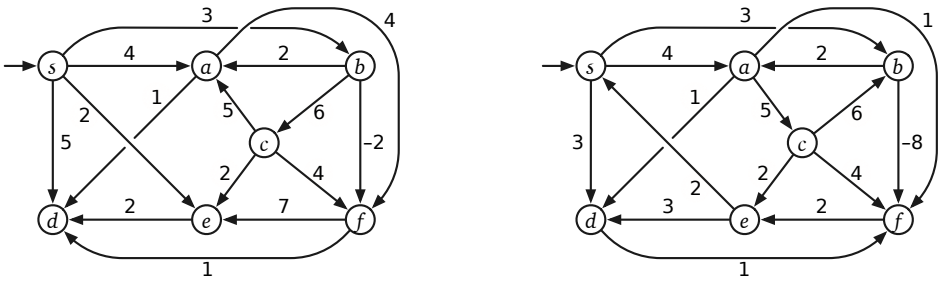
Parts (a) and (b) appeared as complete problems in different versions of the same exam.

Graph Algorithms

Sanity Check

1. ⟨⟨*S14, F14, F16, F19*⟩⟩ Indicate the following structures in the example graphs below.

   - To indicate a subgraph (such as a path, a spanning tree, or a cycle), draw over every edge in the subgraph with a **heavy black line**. Your subgraph should be visible from across the room.
   - To indicate a subset of vertices, either draw a heavy black line around the entire subset, completely blacken the vertices in the subset, or list the vertex labels.
   - If the requested structure does not exist, just write the word NONE.



(a) A depth-first spanning tree rooted at node s.

(b) A breadth-first spanning tree rooted at node s.

(c) A shortest-path tree rooted at node s.

(d) The set of all vertices reachable from node c.

(e) The set of all vertices that can reach node c.

(f) The strong components. (Circle each strong component.)

(g) A simple cycle containing vertex s.

(h) A directed cycle with the minimum number of edges.

(i) A directed cycle with the smallest total weight.

(j) A walk from s to d with the maximum number of edges.

(k) A walk from s to d with the largest total weight.

(l) A depth-first pre-ordering of the vertices. (List the vertices in order.)

(m) A depth-first post-ordering of the vertices. (List the vertices in order.)

(n) A topological ordering of the vertices. (List the vertices in order.)

(o) A breadth-first ordering of the vertices. (List the vertices in order.)

(p) Draw the strong-component graph.

[On an actual exam, we would only ask about one graph, we would give you several copies of the graph in the answer booklet, and we would ask for only a few of these structures.]

2. ⟨⟨*F21*⟩⟩ Draw one example of each of the following graphs.

   (a) A connected undirected graph G with at most ten vertices, such that every vertex has degree at least 2, and *no* depth-first spanning tree of G is a path.

   (b) A directed acyclic graph with one source, and one sink, and more than one topological order.

   (c) A strongly connected directed graph with at least four vertices that contains no directed cycle with more than three edges.

   (d) A directed graph whose edges have distinct weights, but that has more than one shortest path from some vertex s to some other vertex t.

   (e) A strongly connected directed graph, with more than three but less than ten vertices, that contains no directed cycle with *exactly* three edges.

   (f) A strongly connected directed graph, with more than three but less than ten vertices, that contains no directed cycle with *more than* three edges.

[On an actual exam, we would only ask for a small number of these graphs.]

Reachability/Connectivity/Traversal

1. Describe and analyze algorithms for the following problems; in each problem, you are given a graph $G = (V, E)$ with unweighted edges, which may be directed or undirected. You may or may not need different algorithms for directed and undirected graphs.

   (a) Find two vertices that are (strongly) connected.

   (b) Find two vertices that are not (strongly) connected.

   (c) Find two vertices, such that neither vertex can reach the other.

   (d) Find all vertices reachable from a given vertex s.

   (e) Find all vertices that can reach a given vertex s.

   (f) Find all vertices that are strongly connected to a given vertex s.

   (g) Find a simple cycle, or correctly report that the graph has no cycles. (A simple cycle is a closed walk that visits each vertex at most once.)

   (h) Find the *shortest* simple cycle, or correctly report that the graph has no cycles.

   (i) Determine whether deleting a given vertex v would disconnect the graph.

   (j) ⟨⟨*F22*⟩⟩ Find a vertex that can reach the smallest number of other vertices.

   [On an actual exam, we would ask for at most a few of these structures, and we would specify whether the input graph is directed or undirected.]

2. ⟨⟨*F21*⟩⟩ Suppose you are given a directed graph $G = (V, E)$, each of whose vertices is either red, green, or blue. Edges in G do not have weights. G is not necessarily a dag.

   (a) Describe and analyze an algorithm to find every blue vertex b in G such that (1) at least one red vertex can reach b, and (2) b can reach at least one green vertex.

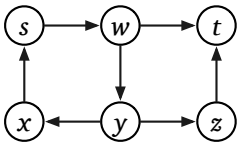
   (b) A vertex of G is *good* if it can reach vertices of all three colors in G. Describe and analyze an algorithm to find every good vertex in G.

   (c) Describe and analyze an algorithm to find a shortest path in G from any red vertex to any green vertex. (In particular, your algorithm must choose the best start and end vertices for the path.)

   (d) Describe and analyze an algorithm to find a shortest path in G that contains at least one vertex of each color. (In particular, your algorithm must choose the best start and end vertices for the path.)

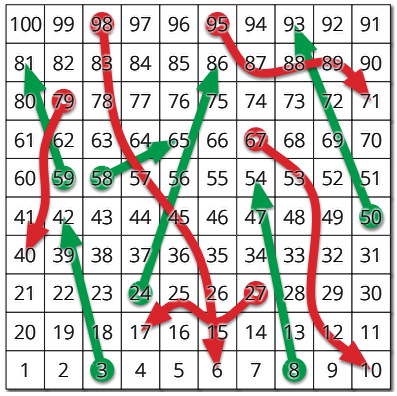
3. ⟨⟨*F14*⟩⟩ Suppose you are given a directed graph $G = (V, E)$ and two vertices s and t. Describe and analyze an algorithm to determine if there is a walk in G from s to t (possibly repeating vertices and/or edges) whose length is divisible by 3.

   For example, given the graph below, with the indicated vertices s and t, your algorithm should return TRUE, because the walk $s \to w \to y \to x \to s \to w \to t$ has length 6.



[Hint: Build a (different) graph.]

4. ***Snakes and Ladders*** is a classic board game, originating in India no later than the 16th century. The board consists of an $n \times n$ grid of squares, numbered consecutively from 1 to n^2, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares, always in different rows, are connected by either "snakes" (leading down) or "ladders" (leading up). Each square can be an endpoint of at most one snake or ladder.



A typical Snakes and Ladders board.
Upward straight arrows are ladders; downward wavy arrows are snakes.

You start with a token in cell 1, in the bottom left corner. In each move, you advance your token up to k positions, for some fixed constant k (typically 6). If the token ends the move at the *top* end of a snake, you ***must*** slide the token down to the bottom of that snake. If the token ends the move at the *bottom* end of a ladder, you ***may*** move the token up to the top of that ladder.

Describe and analyze an algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.

5. Let G be a connected undirected graph. Suppose we start with two coins on two arbitrarily chosen vertices of G, and we want to move the coins so that they lie on the same vertex using as few moves as possible. At every step, each coin *must* move to an adjacent vertex.

   (a) Describe and analyze an algorithm to compute the minimum number of steps to reach a configuration where both coins are on the same vertex, or to report correctly that no such configuration is reachable. The input to your algorithm consists of the graph $G = (V, E)$ and two vertices $u, v \in V$ (which may or may not be distinct).

(b) Now suppose there are *forty-two* coins. Describe and analyze an algorithm to determine whether it is possible to move all 42 coins to the same vertex. Again, *every* coin must move at *every* step. The input to your algorithm consists of the graph $G = (V, E)$ and an array of 42 vertices (which may or may not be distinct). For full credit, your algorithm should run in $O(V + E)$ time.

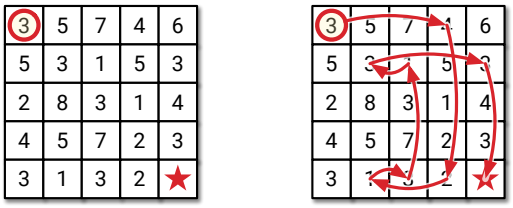
6. A graph (V, E) is bipartite if the vertices V can be partitioned into two subsets L and R, such that every edge has one vertex in L and the other in R.

   (a) Prove that every tree is a bipartite graph.

   (b) Describe and analyze an efficient algorithm that determines whether a given undirected graph is bipartite.

7. ⟨⟨***F14, S18, Lab***⟩⟩ 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.

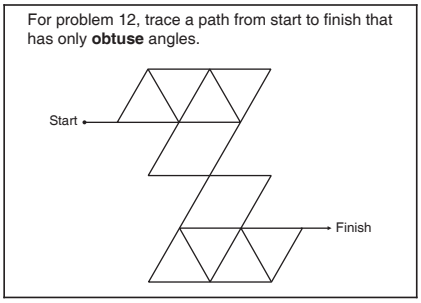
   Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution. For example, given the maze shown below, your algorithm would return the number 8.

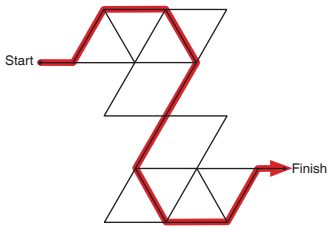


A 5 × 5 number maze that can be solved in eight moves.

8. ⟨⟨***F16***⟩⟩ The following puzzle appeared in my daughter's math workbook several years ago.[1] (I've put the solution on the right so you don't waste time solving it during the exam.)

[1] Jason Batterson and Shannon Rogers, *Beast Academy Math: Practice 3A*, 2012. See https://www.beastacademy.com/resources/printables.php for more examples.



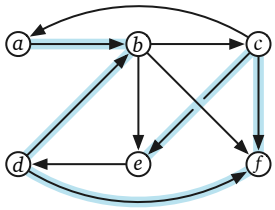


Describe and analyze an algorithm to solve arbitrary obtuse-angle mazes.

You are given a connected undirected graph G, whose vertices are points in the plane and whose edges are line segments. Edges do not intersect, except at their endpoints. For example, a drawing of the letter X would have five vertices and four edges; the maze above has 17 vertices and 26 edges. You are also given two vertices Start and Finish.

Your algorithm should return TRUE if G contains a walk from Start to Finish that has only obtuse angles, and FALSE otherwise. Formally, a walk through G is valid if $\pi/2 < \angle uvw \leq \pi$ for every pair of consecutive edges $u \to v \to w$ in the walk. Assume you have a subroutine that can determine whether the angle between any two segments is acute, right, obtuse, or straight in $O(1)$ time.

9. A *zigzag walk* in in a directed graph G is a sequence of vertices connected by edges in G, but the edges alternately point forward and backward along the sequence. For example, the following graph contains the zigzag walk $a \to b \leftarrow d \to f \leftarrow c \to e$:



If the edges of G have weights, the *length* of a zigzag walk is the sum of the weights of its edges (both forward edges and backward edges).

(a) Suppose you are given a directed graph G with non-negatively weighted edges, along with two vertices s and t. Describe and analyze an algorithm to find the shortest zigzag walk from s to t in G.

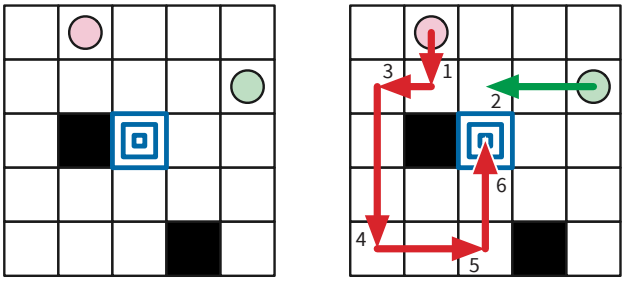
(b) Give an example where the shortest zigzag walk from s to t must visit a vertex more than once.

*(c) ***Prove*** that if all edge weights are non-negative, the shortest zigzag walk never traverses the same edge in both directions.

10. The famous puzzle-maker Kaniel the Dane invented a solitaire game played with two tokens on an $n \times n$ square grid. Some squares of the grid are marked as *obstacles*, and one

grid square is marked as the *target*. In each turn, the player must move one of the tokens from is current position *as far as possible* upward, downward, right, or left, stopping just before the token hits (1) the edge of the board, (2) an obstacle square, or (3) the other token. The goal is to move either of the tokens onto the target square.

For example, in the instance below, we move the red token down until it hits the obstacle, then move the green token left until it hits the red token, and then move the red token left, down, right, and up. In the last move, the red token stops at the target *because* the green token is on the next square above.



An instance of the Kaniel Dane puzzle that can be solved in six moves.
Circles indicate the initial token positions; black squares are obstacles; the center square is the target.

Describe and analyze an algorithm to determine whether an instance of this puzzle is solvable. Your input consist of the integer n, a list of obstacle locations, the target location, and the initial locations of the tokens. The output of your algorithm is a single boolean: TRUE if the given puzzle is solvable and FALSE otherwise. The running time of your algorithm should be a small polynomial in n. *[Hint: Don't forget about the time required to construct the graph!]*

Depth-First Search, Dags, Strong Connectivity

1. ⟨⟨*Lab*⟩⟩ Inspired by an earlier question, you decided 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 needs not be equal among players.

   At the end of the competition, m games have been played. You realized that you had forgotten to implement a proper rating system, and therefore decided 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 any game, then A must rank better than B in the overall ranking.

   You are given the list of players involved and the ranking in each of the m games. Describe and analyze an algorithm to produce an overall ranking of the n players that satisfies the condition, or correctly reports that it is impossible.

2. Let G be a directed acyclic graph with a unique source s and a unique sink t.

   (a) A *Hamiltonian path* in G is a directed path in G that contains every vertex in G. Describe an algorithm to determine whether G has a Hamiltonian path.

   (b) Suppose the vertices of G have weights. Describe an efficient algorithm to find the path from s to t with maximum total weight.

   (c) Suppose we are also given an integer ℓ. Describe an efficient algorithm to find the maximum-weight path from s to t, such that the path contains at most ℓ edges. (Assume there is at least one such path.)

   (d) Suppose several vertices in G are marked *essential*, and we are given an integer k. Design an efficient algorithm to determine whether there is a path from s to t that passes through at least k essential vertices.

   (e) Suppose the vertices of G have integer labels, where $label(s) = -\infty$ and $label(t) = \infty$. Describe an algorithm to find the path from s to t with the maximum number of edges, such that the vertex labels define an increasing sequence.

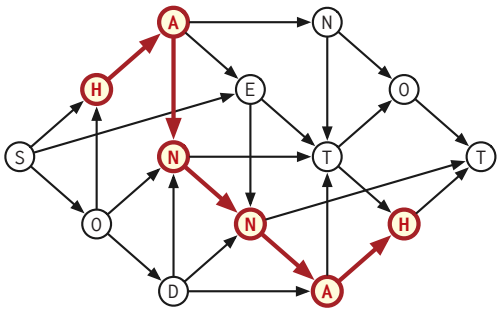
   (f) Describe an algorithm to compute the number of distinct paths from s to t in G. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

3. Suppose you are given a directed acyclic graph G whose nodes represent jobs and whose edges represent *precedence constraints*: Each edge $u \to v$ indicates that job u must be completed before job v begins. Each node v stores a non-negative number $v.duration$ indicating the time required to execute job v. All jobs are executed in parallel; any job can start or end while any number of other jobs are executing, provided all the precedence constraints are satisfied. You'd like to get all these jobs done as quickly as possible.

   Describe an algorithm to determine, for every vertex v in G, the earliest time that job v can ***begin***, assuming the first job starts at time 0 and no precedence constraints are violated. Your algorithm should record the answer for each vertex v in a new field $v.earliest$.

4. 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.

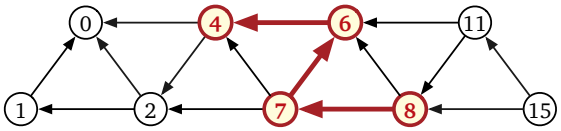
   Describe and analyze an algorithm to find the length of the longest palindrome that is the label of a path in G. For example, given the dag below, your algorithm should return the integer 6, which is the length of the palindrome HANNAH.



5. ⟨⟨*S18*⟩⟩ Let G be a ***directed*** graph, where every vertex v has an associated height $h(v)$, and for every edge $u \to v$ we have the inequality $h(u) > h(v)$. Assume all heights are distinct. The *span* of a path from u to v is the height difference $h(u) - h(v)$.

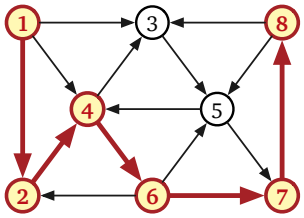
   Describe and analyze an algorithm to find the ***minimum span*** of a path in G with ***at least*** k edges. Your input consists of the graph G, the vertex heights $h(\cdot)$, and the integer k. Report the running time of your algorithm as a function of V, E, and k.

   For example, given the following labeled graph and the integer $k = 3$ as input, your algorithm should return the integer 4, which is the span of the path $8 \to 7 \to 6 \to 4$.



6. ⟨⟨*S18*⟩⟩ Let G be an arbitrary (*not* necessarily acyclic) directed graph in which every vertex v has an integer label $\ell(v)$. Describe an algorithm to find the longest directed path in G whose vertex labels define an increasing sequence. Assume all labels are distinct.

   For example, given the following graph as input, your algorithm should return the integer 5, which is the length of the increasing path $1 \to 2 \to 4 \to 6 \to 7 \to 8$.



7. ⟨⟨***F16***⟩⟩ Suppose you have a collection of n lockboxes and m gold keys. Each key unlocks *at most* one box; however, each box might be unlocked by one key, by multiple keys, or by no keys at all. There are only two ways to open each box once it is locked: Unlock it properly (which requires having a matching key in your hand), or smash it to bits with a hammer.

   Your baby brother, who loves playing with shiny objects, has somehow managed to lock all your keys inside the boxes! Luckily, your home security system recorded everything, so you know exactly which keys (if any) are inside each box. You need to get all the keys back out of the boxes, because they are made of gold. Clearly you have to smash at least one box.

   (a) Your baby brother has found the hammer and is eagerly eyeing one of the boxes. Describe and analyze an algorithm to determine if smashing the box your brother has chosen would allow you to retrieve all m keys.

   (b) Describe and analyze an algorithm to compute the minimum number of boxes that must be smashed to retrieve all the keys.

8. ⟨⟨***F21***⟩⟩ Suppose you are given a height map of a mountain, in the form of an $n \times n$ grid of evenly spaced points, each labeled with an elevation value. You can safely hike directly from any point to any neighbor immediately north, south, east, or west, but only if the elevations of those two points differ by at most Δ. (The value of Δ depends on your hiking experience and your physical condition.)

   Describe and analyze an algorithm to determine the longest hike from some point s to some other point t, where the hike consists of an uphill climb (where elevations must increase at each step) followed by a downhill climb (where elevations must decrease at each step). Your input consists of an array $Elevation[1..n, 1..n]$ of elevation values, the starting point s, the target point t, and the parameter Δ.

9. ⟨⟨***HW***⟩⟩ Ronnie and Hyde are a professional robber duo who plan to rob exactly *one* house in the Leverwood neighborhood of Sham-Boobanana. They have managed to obtain a map of the neighborhood in the form of a directed graph G, whose vertices represent houses and whose edges represent one-way streets.

   - One vertex s represents the house that Ronnie and Hyde plan to rob.
   - A set X of special vertices designate eXits from the neighborhood.
   - For each directed edge $u \to v$, Ronnie can drive directly from house u to house v in $w(u \to v)$ minutes.
   - Driving backwards along any street immediately triggers traffic drones.

   Describe and analyze an algorithm to compute the shortest time needed to exit the neighborhood, starting at house s. The input to your algorithm is the directed graph $G = (V, E)$, with clearly marked subset of exit vertices $X \subseteq V$, and non-negative weights $w(u \to v)$ for every edge $u \to v$.

10. ⟨⟨*F21*⟩⟩ Aladdin and Badroulbadour are playing a cooperative game. Each player has an array of positive integers, arranged in a row of squares from left to right. Each player has a token, which starts at the leftmost square of their row; their goal is to move *both* tokens to the rightmost squares.

    On each turn, *both* players move their tokens *in the same direction*, either left or right. The distance each token travels is equal to the number under that token at the beginning of the turn. For example, if a token starts on a square labeled 5, then it moves either five squares to the right or five squares to the left. If *either* token moves past either end of its row, then both players immediately lose.

    For example, if Aladdin and Badroulbadour are given the arrays

A:	7	5	4	1	2	3	3	2	3	1	4	2
B:	5	1	2	4	7	3	5	2	4	6	3	1

    they can win the game by moving right, left, left, right, right, left, right. On the other hand, if they are given the arrays

A:	2	3	5	1	3
B:	3	4	1	2	1

    they cannot win the game. (The first move must be to the right; then Aladdin's token moves out of bounds on the second turn.)

    Describe and analyze an algorithm to determine whether Aladdin and Badroulbadour can solve their puzzle, given the input arrays $A[1..n]$ and $B[1..n]$.

11. ⟨⟨*S23*⟩⟩ Let $G = (V, E)$ be a directed graph, where every edge $e \in E$ has a non-negative *width* $w(e)$. The *bottleneck width* of a directed cycle C in G is the *minimum* width among all edges in C.

    (a) Describe and analyze an algorithm that, given a graph G, a vertex s, and a real number ω, determines whether there is a cycle in G containing s with bottleneck width at least ω.

    (b) Describe and analyze an algorithm that, given a graph G and a vertex s, either finds the minimum value ω such that there is a cycle in G containing s with bottleneck width ω, or reports correctly that no cycle in G contains s. (You can skip part (a) if you can answer this part directly.)

Shortest Paths

1. Suppose you are given a directed graph G with weighted edges and a vertex s of G.

   (a) ⟨⟨***F14***⟩⟩ Suppose every vertex $v \neq s$ stores a pointer $pred(v)$ to another vertex in G. Describe and analyze an algorithm to determine whether these predecessor pointers correctly define a single-source shortest path tree rooted at s.

   (b) Suppose every vertex v stores a finite real value $dist(v)$. (In particular, $dist(v)$ is never equal to ∞ or $-\infty$.) Describe and analyze an algorithm to determine whether these real values are correct shortest-path distances from s.

   Do ***not*** assume that G has no negative cycles.

2. ⟨⟨***F14***⟩⟩ Suppose we are given an undirected graph G in which every *vertex* has a positive weight.

   (a) Describe and analyze an algorithm to find a *spanning tree* of G with minimum total weight. (The total weight of a spanning tree is the sum of the weights of its vertices.)

   (b) Describe and analyze an algorithm to find a *path* in G from one given vertex s to another given vertex t with minimum total weight. (The total weight of a path is the sum of the weights of its vertices.)

3. ⟨⟨***S14, S18, Lab***⟩⟩ You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cites 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 quickly as possible.

4. ⟨⟨***F16***⟩⟩ There are n galaxies connected by m intergalactic teleport-ways. Each teleport-way joins two galaxies and can be traversed in both directions. Also, each teleport-way uv has an associated cost of $c(uv)$ galactic credits, for some positive integer $c(uv)$. The same teleport-way can be used multiple times in either direction, but the same toll must be paid every time it is used.

   Judy wants to travel from galaxy s to galaxy t, but teleportation is rather unpleasant, so she wants to minimize the number of times she has to teleport. However, she also wants the total cost to be a multiple of 10 galactic credits, because carrying small change is annoying.

Describe and analyze an algorithm to compute the minimum number of times Judy must teleport to travel from galaxy s to galaxy t so that the total cost of all teleports is an integer multiple of 10 galactic credits. Your input is a graph $G = (V, E)$ whose vertices are galaxies and whose edges are teleport-ways; every edge uv in G stores the corresponding cost $c(uv)$.

*[Hint: This is **not** the same Intergalactic Judy problem that you saw in lab.]*

5. ⟨⟨***Lab***⟩⟩ 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 non-negative weight.

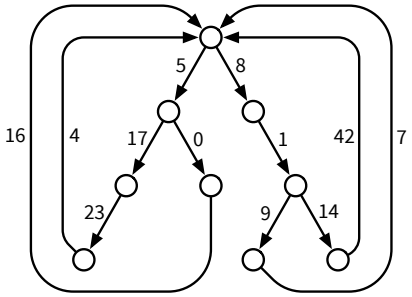


Figure 1. A looped tree.

   (a) How much time would Dijkstra's algorithm require to compute the shortest path from one vertex s to another vertex t in a looped tree with n nodes?

   (b) Describe and analyze a faster algorithm.

6. ⟨⟨***F17, Lab***⟩⟩ Suppose you are given a directed graph G with weighted edges, where *exactly one* edge has negative weight and all other edge weights are positive, along with two vertices s and t. Describe and analyze an algorithm that either computes a shortest path in G from s to t, or reports correctly that the G contains a negative cycle. (As always, faster algorithms are worth more points.)

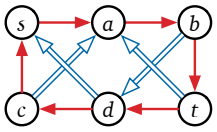
7. ⟨⟨***HW***⟩⟩ When there is more than one shortest path from one node s to another node t, it is often convenient to choose a shortest path with the fewest edges; call this the ***best*** path from s to t. Suppose we are given a directed graph G with positive edge weights and a source vertex s in G. Describe and analyze an algorithm to compute best paths in G from s to every other vertex.

8. After graduating you accept a job with Aerophobes-Я-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of all the flights on the planet.

Suppose one of your customers wants to fly from city X to city Y. Describe an algorithm to find a sequence of flights that minimizes the *total time in transit*—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights.

9. ⟨⟨*S18*⟩⟩ Suppose you are given a directed graph G where some edges are red and the remaining edges are blue. Describe an algorithm to find the shortest walk in G from one vertex s to another vertex t in which no three consecutive edges have the same color. That is, if the walk contains two red edges in a row, the next edge must be blue, and if the walk contains two blue edges in a row, the next edge must be red.

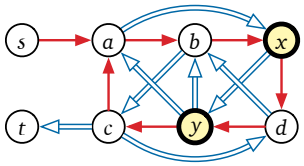
   For example, if you are given the graph below (where single arrows are red and double arrows are blue), your algorithm should return the integer 7, because the shortest legal walk from s to t is $s \to a \to b \Rightarrow d \to c \Rightarrow a \to b \to c$.



10. ⟨⟨*S18*⟩⟩ Suppose you are given a directed graph G in which every edge is either red or blue, and a subset of the vertices are marked as *special*. A walk in G is *legal* if color changes happen only at special vertices. That is, for any two consecutive edges $u \to v \to w$ in a legal walk, if the edges $u \to v$ and $v \to w$ have different colors, the intermediate vertex v must be special.

    Describe and analyze an algorithm that either returns the length of the shortest legal walk from vertex s to vertex t, or correctly reports that no such walk exists.[2]

    For example, if you are given the following graph below as input (where single arrows are red, double arrows are blue), with special vertices x and y, your algorithm should return the integer 8, which is the length of the shortest legal walk $s \to x \to a \to b \to x \Rightarrow y \Rightarrow b \Rightarrow c \Rightarrow t$. The shorter walk $s \to a \to b \Rightarrow c \Rightarrow t$ is not legal, because vertex b is not special.

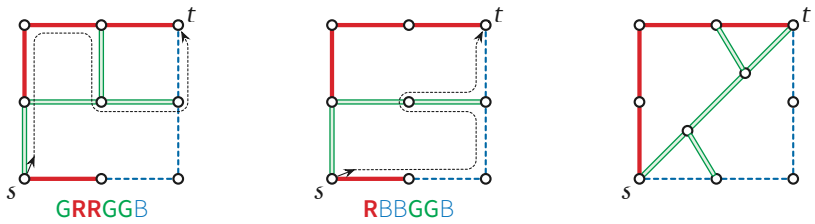


11. ⟨⟨*S18*⟩⟩ Let G be a directed graph with weighted edges, in which every vertex is colored either red, green, or blue. Describe and analyze an algorithm to compute the length of the shortest walk in G that starts at a red vertex, then visits any number of vertices of any color, then visits a green vertex, then visits any number of vertices of any color, then visits a blue vertex, then visits any number of vertices of any color, and finally ends at a red vertex. Assume all edge weights are positive.

[2] If you've read China Miéville's excellent novel *The City & the City*, this problem should look familiar. If you haven't read *The City & the City*, I can't tell you why this problem should look familiar without spoiling the book.

12. ⟨⟨*F19*⟩⟩ Suppose you are given an undirected graph G in which every edge is either red, green, or blue, along with two vertices s and t. Call a walk from s to t *awesome* if the walk does not contain three consecutive edges with the same color.

    Describe and analyze an algorithm to find the length of the shortest awesome walk from s to t. For example, given either the left or middle input below, your algorithm should return the integer 6, and given the input on the right, your algorithm should return ∞.



13. ⟨⟨*F19*⟩⟩ During her walk to work every morning, Rachel likes to buy a cappuccino at a local coffee shop, and a croissant at a local bakery. Her home town has *lots* of coffee shops and lots of bakeries, but strangely never in the same building. Punctuality is not Rachel's strongest trait, so to avoid losing her job, she wants to follow the shortest possible route.

    Rachel has a map of her home town in the form of an undirected graph G, whose vertices represent intersections and whose edges represent roads between them. A subset of the vertices are marked as bakeries; another disjoint subset of vertices are marked as coffee shops. The graph has two special nodes s and t, which represent Rachel's home and work, respectively.

    Describe an algorithm that computes the shortest path in G from s to t that visits both a bakery and a coffee shop, or correctly reports that no such path exists.

14. ⟨⟨*F19*⟩⟩ As the days get shorter in winter, Eggsy Hutmacher is increasingly worried about his walk home from work. The city has recently been invaded by the notorious Antimilliner gang, whose members hang out on dark street corners and steal hats from unwary passers-by, and a gentleman is simply *not* seen out in public without a hat. The city council is slowly installing street lamps at intersections to deter the Antimilliners, whose uncovered faces can be easily identified in the light. Eggsy keeps k extra hats in his briefcase in case of theft or other millinery emergencies.

    Eggsy has a map of the city in the form of an undirected graph G, whose vertices represent intersections and whose edges represent streets between them. A subset of the vertices are marked to indicate that the corresponding intersections are lit. Every edge e has a non-negative length $\ell(e)$. The graph has two special nodes s and t, which represent Eggsy's work and home, respectively.

    Describe an algorithm that computes the shortest path in G from s to t that visits at most k unlit vertices, or correctly reports that no such path exists. Analyze your algorithm as a function of the parameters V, E, and k.

15. ⟨⟨*F19, HW*⟩⟩ You and your friends are planning a hiking trip in Jellystone National Park over winter break. You have a map of the park's trails that lists all the scenic views in the park but also warns that certain trail segments have a high risk of bear encounters. To make the hike worthwhile, you want to see at least three scenic views. You also don't want to get eaten by a bear, so you are willing to hike at most one high-bear-risk segment. Because the trails are narrow, each trail segment allows traffic in only one direction.

    Your friend has converted the map into a directed graph $G = (V, E)$, where V is the set of intersections and E is the set of trail segments. A subset S of the edges are marked as *Scenic*; another subset B of the edges are marked as *high-Bear-risk*. You may assume that $S \cap B = \emptyset$. Each segment $e \in E$ is also labeled with a positive length $\ell(e)$ in miles. Your campsite appears on the map as a particular vertex $s \in V$, and the visitor center is another vertex $t \in V$.

    Describe and analyze an algorithm to compute the shortest hike from your campsite s to the visitor center t that includes *at least* three scenic trail segments and *at most* one high-bear-risk trail segment. You may assume such a hike exists.

You have 120 minutes to answer five questions.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. Short answers:

   (a) Solve the following recurrences:
   - $A(n) = 3A(n/2) + O(n^2)$
   - $B(n) = 7B(n/2) + O(n^2)$
   - $C(n) = 4C(n/2) + O(n^2)$

   (b) Draw a directed acyclic graph with at most ten vertices, exactly one source, exactly one sink, and more than one topological order.

   (c) Draw a directed graph with at most ten vertices, with distinct positive edge weights, that has more than one shortest path from some vertex s to some other vertex t.

   (d) Describe an appropriate memoization structure and evaluation order for the following (meaningless) recurrence, and give the running time of the resulting iterative algorithm to compute $Huh(1, n)$.

$$Huh(i,k) = \begin{cases} 0 & \text{if } i > n \text{ or } k < 0 \\ \min \begin{Bmatrix} Huh(i+1,k-2) \\ Huh(i+2,k-1) \end{Bmatrix} + A[i,k] & \text{if } A[i,k] \text{ is even} \\ \max \begin{Bmatrix} Huh(i+1,k-2) \\ Huh(i+2,k-1) \end{Bmatrix} - A[i,k] & \text{if } A[i,k] \text{ is odd} \end{cases}$$

2. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points, which could be positive, negative, or zero. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

   Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it's his turn, Elmo *always* takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

   (a) ***Prove*** that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do *not* follow the same greedy strategy as Elmo. Assume Elmo plays first.

   (b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

3. Suppose you are given a directed graph $G = (V, E)$, whose vertices are either red, green, or blue. Edges in G do not have weights, and G is not necessarily a dag. The ***remoteness*** of a vertex v is the *maximum* of three shortest-path lengths:

   - The length of a shortest path to v from the closest red vertex
   - The length of a shortest path to v from the closest blue vertex
   - The length of a shortest path to v from the closest green vertex

   In particular, if v is not reachable from vertices of all three colors, then v is infinitely remote.

   Describe and analyze an algorithm to find a vertex of G with *minimum* remoteness.

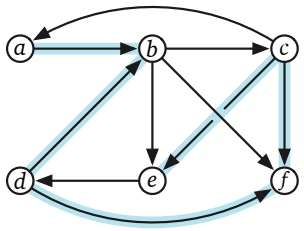
4. Suppose you are given an array $A[1..n]$ of integers such that $A[i] + A[i+1]$ is even for *exactly one* index i. In other words, the elements of A alternate between even and odd, except for exactly one adjacent pair that are either both even or both odd.

   Describe and analyze an efficient algorithm to find the unique index i such that $A[i] + A[i+1]$ is even. For example, given the following array as input, your algorithm should return the integer 6, because $A[6] + A[7] = 88 + 62$ is even. (Cells containing even integers are shaded blue.)

17	40	23	72	39	88	62	13	40	53	92	21	10	73	68

5. A *zigzag walk* in in a directed graph G is a sequence of vertices connected by edges in G, but the edges alternately point forward and backward along the sequence. Specifically, the first edge points forward, the second edge points backward, and so on. The *length* of a zigzag walk is the sum of the weights of its edges, both forward and backward.

   For example, the following graph contains the zigzag walk $a \to b \leftarrow d \to f \leftarrow c \to e$. Assuming every edge in the graph has weight 1, this zigzag walk has length 5.



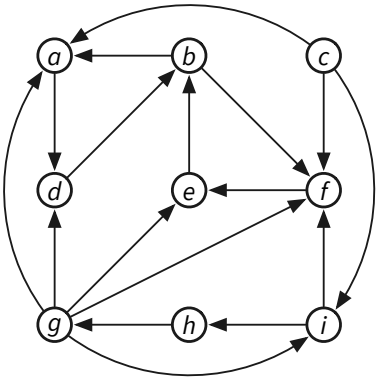
   Suppose you are given a directed graph G with non-negatively weighted edges, along with two vertices s and t. Describe and analyze an algorithm to find the shortest zigzag walk from s to t in G.

You have 120 minutes to answer five questions.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. ***Clearly*** indicate the following structures in the directed graph below, or write NONE if the indicated structure does not exist. Don't be subtle; to indicate a collection of edges, draw a heavy black line along the entire length of each edge.



   (a) A depth-first search tree rooted at vertex a.

   (b) A breadth-first tree rooted at vertex c.

   (c) The strong components of G. (Circle each strong component.)

   (d) Draw the strong-component graph of G.

2. Suppose we are given an n-digit integer X. Repeatedly remove one digit from either end of X (your choice) until no digits are left. The *square-depth* of X is the maximum number of perfect squares that you can see during this process. For example, the number 32492 has square-depth 3, by the following sequence of removals:

$$32492 \xrightarrow{} 3249 \xrightarrow{57^2} 324 \xrightarrow{18^2} 24 \xrightarrow{} 4 \xrightarrow{2^2} \varepsilon.$$

   Describe and analyze an algorithm to compute the square-depth of a given integer X, represented as an array $X[1..n]$ of n decimal digits. Assume you have access to a subroutine IsSquare that determines whether a given k-digit number (represented by an array of digits) is a perfect square ***in $O(k^2)$ time***.

3. Suppose you are given a directed graph $G = (V, E)$, each of whose edges are colored red, green, or blue. Edges in G do not have weights, and G is not necessarily a dag. A *rainbow walk* is a walk in G that does *not* contain two consecutive edges with the same color.

   Describe and analyze an algorithm to find all vertices in G that are reachable from a given vertex s through a rainbow walk.

4. Suppose you are given k sorted arrays $A_1[1..n], A_2[1..n], \ldots, A_k[1..n]$, all with the same length n. Describe an algorithm to merge the given arrays into a single sorted array. Analyze the running time of your algorithm as a function of n and k.

5. After moving to a new city, you decide to walk from your home to your new office. To get a good daily workout, you want to reach the highest possible altitude during your walk (to maximize exercise), while keeping the total length of your walk below some threshold (to get to your office on time). Describe and analyze an algorithm to compute the best possible walking route.

   Your input consists of an undirected graph G, where each vertex v has a height $h(v)$ and each edge e has a positive length $\ell(e)$, along with a start vertex s, a target vertex t, and a maximum length L. Your algorithm should return the maximum height reachable by a walk from s to t in G, whose total length is at most L.

You have 120 minutes to answer five questions.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. Short answers:

   (a) Solve the following recurrences:

   - $A(n) = A(5n/11) + O(\sqrt{n})$
   - $B(n) = 8B(n/2) + O(n^2)$
   - $C(n) = C(n/2) + C(n/3) + C(n/6) + O(n)$

   (b) Describe an appropriate memoization structure and evaluation order for the following (meaningless) recurrences, and state the running time of the resulting iterative algorithm to compute the requested function value.

   - Compute $Foo(1, n)$ where

$$Foo(i,k) = \begin{cases} 0 & \text{if } i \geq k-1 \\ \max\left\{ \begin{array}{c} Foo(i,j) \\ +\, Foo(j,k) \end{array} \;\middle|\; i < j < k \right\} + \sum_{j=i}^{k} A[j] & \text{otherwise} \end{cases}$$

   - Compute $Bar(n, 1)$ where

$$Bar(i,s) = \begin{cases} \infty & \text{if } i < 0 \text{ or } s > n \\ 0 & \text{if } i = 0 \\ \min\left\{ \begin{array}{c} Bar(i, 2s), \\ X[i] \cdot s + Bar(i-s, s) \end{array} \right\} & \text{otherwise} \end{cases}$$

2. Chandler is moving to Tulsa, Oklahoma, to start a new job, and he wants to plan a walk from home to work. He wants to continue his habit of buying a ~~hot milkshake~~ salted caramel latte from a local coffee shop and a paper from a local newsstand every morning. (Yes, an actual *paper* paper. Could he *be* any more retro?)

   Chandler has a map of his new neighborhood in the form of an undirected graph G, whose vertices represent intersections and whose edges represent roads between them. Every edge e has a positive length $\ell(e)$. A subset of the vertices are marked as newsstands; another disjoint subset of vertices are marked as coffee shops. The graph has two special vertices s and t, which represent Chandler's home and work, respectively.

   Describe an algorithm that computes the shortest route that Chandler can follow from home to work that visits both a coffee shop and a newsstand, or correctly reports that no such route exists (which means Chandler should move back to New York).

Problems 3 and 4 are on the back of this page.

3. Recall that an *arithmetic progression* is any sequence of real numbers $x_1, x_2, \ldots, x_n$ such that $x_{i+1} - x_i = x_i - x_{i-1}$ for every index $2 \le i \le n-1$.

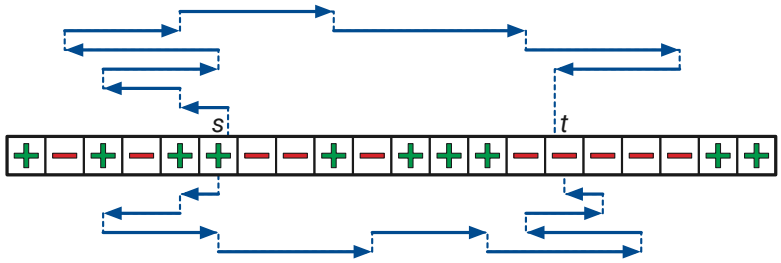
   Suppose we are given a sorted array $X[1..n]$ that contains an arithmetic sequence ***with one element deleted***. Describe and analyze an algorithm to find the deleted element as quickly as possible. (If there are multiple correct answers, your algorithm can return any one of them.)

   For example, given the input array $X = [2, 4, 8, 10, 12]$, your algorithm should return 6, and given the array $X = [21, 18, 15, 12]$, your algorithm should return either 9 or 24.

4. ***Ink-deck*** is a solitaire puzzle game played on a row of n squares, each marked with either $+$ or $-$. Your goal is to move a token from a specified start square s to a specified target square t using a sequence of *moves*. Each move translates the token either left or right along the row to a new square. The *length* of a move is the distance that the token moves; for example, a move from square 5 to square 12 has length 7. Moves are subject to the following rules:

   - The first move must have length 1.
   - If the token is on a square marked $+$, your next move must be one square longer than your previous move.
   - If the token is on a square marked $-$, your next move must be one square shorter than your previous move. (In particular, if your previous move had length 1, then you cannot move at all!)
   - You are never allowed to move the token off either end of the row.

   The following figure shows an example ink-deck puzzle, along with two solutions (which might not be optimal).



An ink-deck puzzle with two nine-move solutions.

   (a) Describe an algorithm that either finds a solution *with the minimum number of moves* for a given ink-deck puzzle, or correctly reports that the given puzzle has no solution.

   (b) Describe an algorithm that either finds a solution *whose final move is as long as possible* for a given ink-deck puzzle, or correctly reports that the given puzzle has no solution.

   Your input to both algorithms consists of an array $ID[1..n]$, where $ID[i] \in \{-1, +1\}$ for each index i, along with two indices $1 \le s \le n$ and $1 \le t \le n$.

Problem 5 is on the next page.

5. Suppose you are given a string of symbols, representing a message in some foreign language that you do not understand, in an array $T[1..n]$. You have access to a black-box subroutine IsWord that takes a string w as input and decides in $O(|w|)$ time whether w is a word.

   You eagerly implement and run the text-splitting algorithm we saw in class, only to discover that the given string *cannot* be split into words! Apparently, as a crude form of cryptography, the message has been corrupted by adding extra symbols between words.

   So you decide instead to look for as many non-overlapping words in T as possible. A *verbal subsequence* of T is a sequence of non-overlapping substrings of T, each of which is a word. The *length* of a verbal subsequence is the number of words it contains. Describe and analyze an algorithm to find the length of the longest verbal subsequence of a given string T.

   For example, suppose IsWord(w) returns True if and only if w is an English word with at least four letters. Then (STUDY, MICE, TRAP, RAMEN) and (DYNAMIC, EXTRA, PROGRAM) are verbal subsequences of the string STUDYNAMICEXTRAPROGRAMEN:

   Given the input string STUDYNAMICEXTRAPROGRAMEN, your algorithm should return the integer 4, which is the length of the verbal subsequence (STUDY, MICE, TRAP, RAMEN).

Nothing to see here.

You have 120 minutes to answer five questions.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. (a) Solve the following recurrences:
   - $A(n) = A(2n/3) + O(\sqrt{n})$
   - $B(n) = 8B(n/4) + O(n^{3/2})$
   - $C(n) = C(n/2) + C(n/3) + O(n)$

   (b) Describe an appropriate memoization structure and evaluation order for the following (meaningless) recurrences, and state the running time of the resulting iterative algorithm to compute the requested function value.
   - Compute $Pleb(n, 1)$ where

$$Pleb(i,k) = \begin{cases} i & \text{if } k \le 0 \\ k & \text{if } i > n \\ \max \left\{ \begin{array}{c} i + k + Pleb(i-1, k+1) \\ i + Pleb(i-1, k) \\ k + Pleb(i, k+1) \end{array} \right\} & \text{otherwise} \end{cases}$$

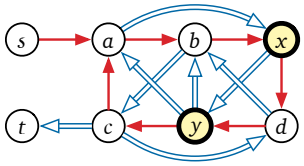
   - Compute $Nom(1, n)$ where

$$Nom(i,k) = \begin{cases} 0 & \text{if } k = i \\ (X[k] - X[i]) + \min \left\{ \begin{array}{l} Nom(i,j) \\ + Nom(j,k) \end{array} \middle| \; i < j < k \right\} & \text{otherwise} \end{cases}$$

2. Suppose you are given a directed graph G in which every edge is either red or blue, and a subset of the vertices are marked as *special*. A walk in G is *legal* if color changes happen only at special vertices. That is, for any two consecutive edges $u \to v \to w$ in a legal walk, if the edges $u \to v$ and $v \to w$ have different colors, the intermediate vertex v must be special.

   Describe and analyze an algorithm that either returns the length of the shortest legal walk in G from vertex s to vertex t, or correctly reports that no such walk exists.

   For example, if you are given the following graph as input (where single arrows are "red" and double arrows are "blue"), with special vertices x and y, your algorithm should return the integer 8, which is the length of the shortest legal walk $s \to x \to a \to b \to x \Rightarrow y \Rightarrow b \Rightarrow c \Rightarrow t$. The shorter walk $s \to a \to b \Rightarrow c \Rightarrow t$ is not legal, because vertex b is not special.

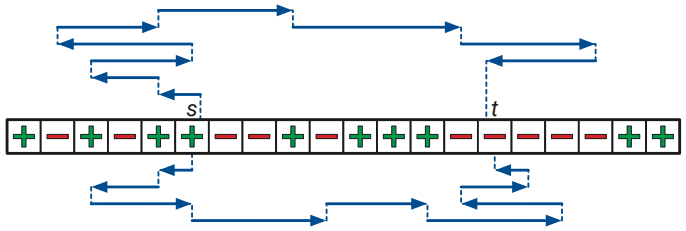


Problem 3 is on the back of this page.

3. ***Ink-deck*** is a solitaire puzzle game played on a row of n squares, each marked with either $+$ or $-$. Your goal is to move a token from a specified start square s to a specified target square t using a sequence of *moves*. Each move translates the token either left or right along the row to a new square. The *length* of a move is the distance that the token moves; for example, a move from square 5 to square 12 has length 7. Moves are subject to the following rules:

   - The first move must have length 1.
   - If the token is on a square marked $+$, your next move must be one square longer than your previous move.
   - If the token is on a square marked $-$, your next move must be one square shorter than your previous move. (In particular, if your previous move had length 1, then you cannot move at all!)
   - You are never allowed to move the token off either end of the row.

   The following figure shows an example ink-deck puzzle, along with two solutions.



An ink-deck puzzle with two solutions.

   (a) The *total length* of a solution is the sum of the lengths of the moves. For example, the top solution in the figure above has total length $1+2+3+4+3+4+5+4+3 = 29$, and the bottom solution has total length $1+2+3+4+3+4+3+2+1 = 23$.

   Describe an algorithm that, given an ink-deck puzzle, either finds a solution whose *total* length is as *small* as possible, or correctly reports that there is no solution.

   (b) The *maximum length* of a solution is length of its longest move. For example, the top solution above has maximum length 5, and the bottom solution has maximum length 4.

   Describe an algorithm that, given an ink-deck puzzle, either finds a solution whose *maximum* length is as *large* as possible, or correctly reports that there is no solution.

Your input to both algorithms consists of an array $ID[1..n]$, where $ID[i] \in \{-1, +1\}$ for each index i, along with two indices $1 \le s \le n$ and $1 \le t \le n$.

Problems 4 and 5 are on the next page.

4. Recall that an *arithmetic progression* is any sequence of real numbers $x_1, x_2, \ldots, x_n$ such that $x_{i+1} - x_i = x_i - x_{i-1}$ for every index $2 \le i \le n-1$.

   Suppose we are given a sorted array $X[1..n]$ containing an arithmetic sequence *with one element repeated once*. Describe and analyze an algorithm to find the repeated element as quickly as possible.

   For example, given the input array $X = [2, 4, 6, 6, 8, 10, 12]$, your algorithm should return 6, and given the input array $X = [1, 1, 1, 1]$, your algorithm should return 1.

5. Suppose you are given a string of symbols, representing a message in some foreign language that you do not understand, in an array $T[1..n]$. You have access to a black-box subroutine IsWord that takes a string w as input and decides in $O(|w|)$ time whether w is a word.

   You eagerly implement and run the text-splitting algorithm we saw in class, only to discover that the given string *cannot* be split into words! Apparently, as a crude form of cryptography, the author of the message added extra symbols at the beginning and end.

   Describe and analyze an algorithm to find the length of the *longest substring* of T that can be split into words.

   For example, suppose IsWord(w) returns True if and only if w is an English word with at least four letters. Given the input string STURDYNAMICEXTRAPROGRAMBLE, your algorithm should return the integer 19, which is the length of the substring DYNAMICEXTRAPROGRAM, which can be split into the words DYNAMIC, EXTRA, and PROGRAM:

   STUR | DYNAMIC | EXTRA | PROGRAM | BLE

   The words STURDY, MICE, TRAP, and RAMBLE have larger total length 20, but there are gaps between them; so they can't be formed by splitting a *substring* of T.

   | STURDY | NA | MICE | X | TRAP | ROG | RAMBLE |

Nothing to see here.

CS/ECE 374 A ✧ Fall 2023

☙ Final Exam Study Questions ❧

This is a "core dump" of potential questions for the final exam. This should give you a good idea of the *types* of questions that we will ask on the exam. In particular, there will be a series of True/False or short-answer questions—but the actual exam questions may or may not appear in this handout. This list intentionally includes a few questions that are too long or difficult for exam conditions; these are indicated with a *star.

Don't forget to review the study problems for Midterms 1 and 2; the final exam is cumulative!

☙ How to Use These Problems ❧

Solving every problem in this handout is ***not*** the best way to study for the exam. Memorizing the solutions to every problem in this handout is the ***absolute worst*** way to study for the exam.

What we recommend instead is to work on a *sample* of the problems. Choose one or two problems at random from each section and try to solve them from scratch under exam conditions—by yourself, in a quiet room, with a 30-minute timer, *without* your notes, *without* the internet, and if possible, even without your cheat sheet. If you're comfortable solving a few problems in a particular section, you're probably ready for that type of problem on the exam. Move on to the next section.

Discussing problems with other people (in your study groups, in the review sessions, in office hours, or on Piazza) and/or looking up old solutions can be *extremely* helpful, but ***only after*** you have (1) made a good-faith effort to solve the problem on your own, and (2) you have either a candidate solution or some idea about where you're getting stuck.

If you find yourself getting stuck on a particular type of problem, try to figure out *why* you're stuck. Do you understand the problem statement? Are you stuck on choosing the right high-level approach? Are you stuck on the technical details? Or are you struggling to express your ideas clearly? (We *strongly* recommend writing solutions that follow the homework grading rubrics bullet-by-bullet.)

Similarly, if feedback from other people suggests that your solutions to a particular type of problem are incorrect or incomplete, try to figure out what you missed. For NP-hardness proofs: Are you choosing a good problem to reduce from? Are you reducing in the correct direction? Are you designing your reduction with both good instances and bad instances in mind? You're not trying *solve* the problem, are you? For undecidability proofs: Does the problem have the right structure to apply Rice's theorem? If you are arguing by reduction, are you reducing in the correct direction? You're not using *pronouns*, are you?

Remember that your goal is *not* merely to "understand" the solution to any particular problem, but to become more comfortable with solving a certain *type* of problem on your own. **"Understanding" is a trap; aim for mastery.** If you can identify specific steps that you find problematic, read more *about those steps*, focus your practice *on those steps*, and try to find helpful information *about those steps* to write on your cheat sheet. Then work on the next problem!

True or False? (All from previous final exams)

For each statement below, write "YES" or "True" if the statement is ***always*** true and "NO" or "False" otherwise, and give a brief (at most one short sentence) explanation of your answer. **Assume $P \neq NP$.** If there is any other ambiguity or uncertainty about an answer, write "NO" or "False". For example:

- $x + y = 5$
  NO — Suppose $x = 3$ and $y = 4$.
- 3SAT can be solved in polynomial time.
  NO — 3SAT is NP-hard.
- If P = NP then Jeff is the Queen of England.
  YES — The hypothesis is false, so the implication is true.

---

1. Which of the following are clear English specifications of a recursive function that could possibly be used to compute the edit distance between two strings $A[1..n]$ and $B[1..n]$?

   (a) $Edit(i, j)$ is the answer for i and j.

   (b) $Edit(i, j)$ is the edit distance between $A[i]$ and $B[j]$.

   (c) $$Edit[i,j] = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ Edit[i-1, j-1] & \text{if } A[i] = B[j] \\ \max \left\{ \begin{array}{c} 1 + Edit[i, j-1] \\ 1 + Edit[i-1, j] \\ 1 + Edit[i-1, j-1] \end{array} \right\} & \text{otherwise} \end{cases}$$

   (d) $Edit[1..n, 1..n]$ stores the edit distances for all prefixes.

   (e) $Edit(i, j)$ is the edit distance between $A[i..n]$ and $B[j..n]$.

   (f) $Edit[i, j]$ is the value stored at row i and column j of the table.

   (g) $Edit(i, j)$ is the edit distance between the last i characters of A and the last j characters of B.

   (h) $Edit(i, j)$ is the edit distance when i and j are the current characters in A and B.

   (i) Iterate through both strings and update $Edit[\cdot, \cdot]$ at each character.

   (j) $Edit(i, j, k, l)$ is the edit distance between substrings $A[i..j]$ and $B[k..l]$.

   (k) *[I don't need an English description; my pseudocode is clear enough!]*

---

2. Which of the following statements is true for ***every*** directed graph $G = (V, E)$?

   (a) $E \neq \emptyset$.

(b) Given the graph G as input, Floyd-Warshall runs in $O(E^3)$ time.

(c) If G has at least one source and at least one sink, then G is a dag.

(d) We can compute a spanning tree of G using whatever-first search.

(e) If the edges of G are weighted, we can compute the shortest path from any node s to any node t in $O(E \log V)$ time using Dijkstra's algorithm.

---

3. Which of the following statements are true for ***every*** language $L \subseteq \{0,1\}^*$?

(a) L is non-empty.

(b) L is infinite.

(c) L contains the empty string ε.

(d) L^* is infinite.

(e) L^* is regular.

(f) L is accepted by some DFA if and only if L is accepted by some NFA.

(g) L is described by some regular expression if and only if L is rejected by some NFA.

(h) L is accepted by some DFA with 42 states if and only if L is accepted by some NFA with 42 states.

(i) If L is decidable, then L is infinite.

(j) If L is not decidable, then L is infinite.

(k) If L is not regular, then L is undecidable.

(l) If L has an infinite fooling set, then L is undecidable.

(m) If L has a finite fooling set, then L is decidable.

(n) If L is the union of two regular languages, then its complement $\overline{L}$ is regular.

(o) If L is the union of two regular languages, then its complement $\overline{L}$ is context-free.

(p) If L is the union of two decidable languages, then L is decidable.

(q) If L is the union of two undecidable languages, then L is undecidable.

(r) If $L \notin \mathrm{P}$, then L is not regular.

(s) L is decidable if and only if its complement $\overline{L}$ is undecidable.

(t) Both L and its complement $\overline{L}$ are decidable.

---

4. Which of the following statements are true for ***at least one*** language $L \subseteq \{0,1\}^*$?

(a) L is non-empty.

(b) L is infinite.

(c) L contains the empty string ε.

(d) L^* is finite.

(e) L^* is not regular.

(f) L is not regular but L^* is regular.

(g) L is finite and L is undecidable.

(h) L is decidable but L^* is not decidable.

(i) L is not decidable but L^* is decidable.

(j) L is the union of two decidable languages, but L is not decidable.

(k) L is the union of two undecidable languages, but L is decidable.

(l) L is accepted by an NFA with 374 states, but L is not accepted by a DFA with 374 states.

(m) L is accepted by an DFA with 374 states, but L is not accepted by a NFA with 374 states.

(n) L is regular and $L \notin \mathrm{P}$.

(o) There is a Turing machine that accepts L.

(p) There is an algorithm to decide whether an arbitrary given Turing machine accepts L.

---

5. Which of the following languages over the alphabet $\{0, 1\}$ are ***regular***?

(a) $\{0^m 1^n \mid m \geq 0 \text{ and } n \geq 0\}$

(b) All strings with the same number of 0s and 1s

(c) Binary representations of all positive integers divisible by 17

(d) Binary representations of all prime numbers less than 10^{100}

(e) $\{ww \mid w \text{ is a palindrome}\}$

(f) $\{wxw \mid w \text{ is a palindrome and } x \in \{0, 1\}^*\}$

(g) $\{\langle M \rangle \mid M \text{ accepts a regular language}\}$

(h) $\{\langle M \rangle \mid M \text{ accepts a finite number of non-palindromes}\}$

---

6. Which of the following languages/decision problems are ***decidable***?

(a) $\emptyset$

(b) $\{0^n 1^{2n} 0^n 1^{2n} \mid n \geq 0\}$

(c) $\{ww \mid w \text{ is a palindrome}\}$

(d) $\{\langle M \rangle \mid M \text{ accepts } \langle M \rangle \bullet \langle M \rangle\}$

(e) $\{\langle M \rangle \mid M \text{ accepts a finite number of non-palindromes}\}$

(f) $\{\langle M \rangle \bullet w \mid M \text{ accepts } ww\}$

(g) $\{\langle M \rangle \bullet w \mid M \text{ accepts } ww \text{ after at most } |w|^2 \text{ steps}\}$

(h) Given an NFA N, is the language $L(N)$ infinite?

(i) CircuitSat

(j) Given an undirected graph G, does G contain a Hamiltonian cycle?

(k) Given encodings of two Turing machines M and M', is there a string w that is accepted by both M and M'?

---

7. Which of the following languages can be proved undecidable ***using Rice's Theorem***?

   (a) $\emptyset$

   (b) $\{0^n 1^{2n} 0^n 1^{2n} \mid n \geq 0\}$

   (c) $\{ww \mid w \text{ is a palindrome}\}$

   (d) $\{\langle M \rangle \mid M \text{ accepts an infinite number of strings}\}$

   (e) $\{\langle M \rangle \mid M \text{ accepts a finite number of strings}\}$

   (f) $\{\langle M \rangle \mid M \text{ accepts either } \langle M \rangle \text{ or } \langle M \rangle^R\}$

   (g) $\{\langle M \rangle \mid M \text{ accepts both } \langle M \rangle \text{ and } \langle M \rangle^R\}$

   (h) $\{\langle M \rangle \mid M \text{ does not accept exactly 374 palindromes}\}$

   (i) $\{\langle M \rangle \mid M \text{ accepts some string } w \text{ after at most } |w|^2 \text{ steps}\}$

   (j) $\{\langle M \rangle \bullet w \mid M \text{ rejects } w \text{ after at most } |w|^2 \text{ steps}\}$

   (k) Given the encodings of two Turing machines M and M', is there a string w that is accepted by both M and M'?

---

8. Recall the halting language $\textsc{Halt} = \{\langle M \rangle \bullet w \mid M \text{ halts on input } w\}$. Which of the following statements about its complement $\overline{\textsc{Halt}} = \Sigma^* \setminus \textsc{Halt}$ are true?

   (a) $\overline{\textsc{Halt}}$ is empty.

   (b) $\overline{\textsc{Halt}}$ is regular.

   (c) $\overline{\textsc{Halt}}$ is infinite.

   (d) $\overline{\textsc{Halt}}$ is in NP.

   (e) $\overline{\textsc{Halt}}$ is decidable.

---

9. Suppose some language $A \in \{0,1\}^*$ reduces to another language $B \in \{0,1\}^*$. Which of the following statements ***must*** be true?

   (a) A Turing machine that recognizes A can be used to construct a Turing machine that recognizes B.

   (b) A is decidable.

   (c) If B is decidable then A is decidable.

   (d) If A is decidable then B is decidable.

   (e) If B is NP-hard then A is NP-hard.

   (f) If A has no polynomial-time algorithm then neither does B.

---

10. Suppose there is a ***polynomial-time*** reduction from problem A to problem B. Which of the following statements ***must*** be true?

    (a) Problem B is NP-hard.

    (b) A polynomial-time algorithm for B can be used to solve A in polynomial time.

    (c) If B has no polynomial-time algorithm then neither does A.

    (d) If A is NP-hard and B has a polynomial-time algorithm then P = NP.

    (e) If B is NP-hard then A is NP-hard.

    (f) If B is undecidable then A is undecidable.

---

11. Consider the following pair of languages:

    - HAMPATH $:= \{G \mid G$ is an undirected graph with a Hamiltonian path$\}$
    - CONNECTED $:= \{G \mid G$ is a connected undirected graph$\}$

    (For concreteness, assume that in both of these languages, graphs are represented by their adjacency matrices.) Which of the following ***must*** be true, assuming P≠NP?

    (a) CONNECTED $\in$ NP

    (b) HAMPATH $\in$ NP

    (c) HAMPATH is decidable.

    (d) There is no polynomial-time reduction from HAMPATH to CONNECTED.

    (e) There is no polynomial-time reduction from CONNECTED to HAMPATH.

---

12. Consider the following pair of languages:

    - DIRHAMPATH $:= \{G \mid G$ is a directed graph with a Hamiltonian path$\}$
    - ACYCLIC $:= \{G \mid G$ is a directed acyclic graph$\}$

    (For concreteness, assume that in both of these languages, graphs are represented by their adjacency matrices.) Which of the following ***must*** be true, assuming P≠NP?

    (a) ACYCLIC $\in$ NP

    (b) ACYCLIC $\cap$ DIRHAMPATH $\in$ P

    (c) DIRHAMPATH is decidable.

    (d) There is a polynomial-time reduction from DIRHAMPATH to ACYCLIC.

    (e) There is a polynomial-time reduction from ACYCLIC to DIRHAMPATH.

---

13. Consider the following pair of languages:

    - 3COLOR $:= \{G \mid G$ is a 3-colorable undirected graph$\}$

- TREE $:= \{G \mid G \text{ is a connected acyclic undirected graph}\}$

(For concreteness, assume that in both of these languages, graphs are represented by adjacency matrices.) Which of the following ***must*** be true, assuming P≠NP?

(a) TREE is NP-hard.

(b) TREE $\cap$ 3COLOR $\in$ P

(c) 3COLOR is undecidable.

(d) There is a polynomial-time reduction from 3COLOR to TREE.

(e) There is a polynomial-time reduction from TREE to 3COLOR.

---

14. Suppose we want to prove that the following language is undecidable.

$$\text{ALWAYSHALTS} := \{\langle M\rangle \mid M \text{ halts on every input string}\}$$

Rocket J. Squirrel suggests a reduction from the standard halting language

$$\text{HALT} := \{(\langle M\rangle, w) \mid M \text{ halts on inputs } w\}.$$

Specifically, given a Turing machine DECIDEALWAYSHALTS that decides ALWAYSHALTS, Rocky claims that the following Turing machine DECIDEHALT decides HALT.

```
DecideHalt(⟨M⟩, w):
  Write code for the following algorithm:
    Bullwinkle(x):
      if M accepts w
        reject
      if M rejects w
        accept
  return DecideAlwaysHalts(⟨Bullwinkle⟩)
```

Which of the following statements is true ***for all*** inputs $\langle M\rangle\#w$?

(a) If M accepts w, then M' halts on every input string.

(b) If M rejects w, then M' halts on every input string.

(c) If M diverges on w, then M' halts on every input string.

(d) If M accepts w, then DECIDEALWAYSHALTS accepts $\langle$BULLWINKLE$\rangle$.

(e) If M rejects w, then DECIDEHALT rejects $(\langle M\rangle, w)$.

(f) If M diverges on w, then DECIDEALWAYSHALTS diverges on $\langle$BULLWINKLE$\rangle$.

(g) DECIDEHALT decides HALT. (That is, Rocky's reduction is correct.)

---

15. Suppose we want to prove that the following language is undecidable.

$$\textsc{Muggle} := \{\langle M\rangle \mid M \text{ accepts } \texttt{SCIENCE} \text{ but rejects } \texttt{MAGIC}\}$$

Professor Potter, your instructor in Defense Against Models of Computation and Other Dark Arts, suggests a reduction from the standard halting language

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

Specifically, suppose there is a Turing machine DetectoMuggletum that decides Muggle. Professor Potter claims that the following algorithm decides Halt.

```
DecideHalt(⟨M⟩, w):
  Write code for the following algorithm:
      RubberDuck(x):
        run M on input w
        if x = MAGIC
            return False
        else
            return True
  return DetectoMuggletum(⟨RubberDuck⟩)
```

Which of the following statements ***must be*** true ***for all*** inputs $\langle M\rangle\#w$?

(a) If M accepts w, then RubberDuck accepts SCIENCE.

(b) If M accepts w, then RubberDuck accepts CHOCOLATE.

(c) If M rejects w, then RubberDuck rejects MAGIC.

(d) If M rejects w, then RubberDuck halts on every input string.

(e) If M diverges on w, then RubberDuck rejects every input string.

(f) If M accepts w, then DetectoMuggletum accepts $\langle$RubberDuck$\rangle$.

(g) If M rejects w, then DecideHalt rejects $(\langle M\rangle, w)$.

(h) If M diverges on w, then DecideHalt rejects $(\langle M\rangle, w)$.

(i) DecideHalt decides the language Halt. (That is, Professor Potter's reduction is actually correct.)

(j) DecideHalt actually runs (or simulates) RubberDuck.

(k) Muggle is decidable.

---

16. Suppose we want to prove that the following language is undecidable.

$$\textsc{Marvin} := \{\langle M\rangle \mid M \text{ rejects an infinite number of strings}\}$$

Professor Beeblebrox, your instructor in Infinitely Improbable Galactic Presidencies, suggests a reduction from the standard halting language

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

Specifically, suppose there is a program ParanoidAndroid that decides Marvin. Professor Beeblebrox claims that the following algorithm decides Halt.

```
DecideHalt(⟨M⟩, w):
  Write code for the following algorithm:
      HeartOfGold(x):
        run M on input w
        if x = VOGONPOETRY
            return False
        else
            return True
  return ParanoidAndroid(⟨HeartOfGold⟩)
```

Which of the following statements is true for all inputs $(\langle M \rangle, w)$?

(a) If M accepts w, then HeartOfGold accepts VOGONPOETRY.

(b) If M accepts w, then HeartOfGold accepts IMPROBABILITY.

(c) If M rejects w, then HeartOfGold rejects VOGONPOETRY.

(d) If M rejects w, then HeartOfGold rejects IMPROBABILITY.

(e) If M hangs on w, then HeartOfGold accepts VOGONPOETRY.

(f) If M hangs on w, then HeartOfGold rejects IMPROBABILITY.

(g) If M hangs on w, then HeartOfGold hangs on NEILYOUNG.

(h) ParanoidAndroid accepts OKCOMPUTER.

(i) ParanoidAndroid accepts $\langle$HeartOfGold$\rangle$.

(j) ParanoidAndroid rejects $\langle$HeartOfGold$\rangle$.

(k) ParanoidAndroid actually runs (or simulates running) HeartOfGold.

(l) DecideHalt decides Halt; that is, Professor Beeblebrox's proof is correct.

NP-hardness

1. A boolean formula is in *disjunctive normal form* (or *DNF*) if it consists of a *disjunction* (OR) or several *terms*, each of which is the conjunction (AND) of one or more literals. For example, the formula

$$(\overline{x} \wedge y \wedge \overline{z}) \vee (y \wedge z) \vee (x \wedge \overline{y} \wedge \overline{z})$$

is in disjunctive normal form. DNF-SAT asks, given a boolean formula in disjunctive normal form, whether that formula is satisfiable.

   (a) Describe a polynomial-time algorithm to solve DNF-SAT.

   (b) What is the error in the following argument that P=NP?

> *Suppose we are given a boolean formula in conjunctive normal form with at most three literals per clause, and we want to know if it is satisfiable. We can use the distributive law to construct an equivalent formula in disjunctive normal form. For example,*
>
> $$(x \vee y \vee \overline{z}) \wedge (\overline{x} \vee \overline{y}) \iff (x \wedge \overline{y}) \vee (y \wedge \overline{x}) \vee (\overline{z} \wedge \overline{x}) \vee (\overline{z} \wedge \overline{y})$$
>
> *Now we can use the algorithm from part (a) to determine, in polynomial time, whether the resulting DNF formula is satisfiable. We have just solved 3SAT in polynomial time. Since 3SAT is NP-hard, we must conclude that P=NP!*

2. A ***relaxed 3-coloring*** of a graph G assigns each vertex of G one of three colors (for example, red, green, and blue), such that ***at most one*** edge in G has both endpoints the same color.

   (a) Give an example of a graph that has a relaxed 3-coloring, but does not have a proper 3-coloring (where every edge has endpoints of different colors).

   (b) ***Prove*** that it is NP-hard to determine whether a given graph has a relaxed 3-coloring.

3. An ***ultra-Hamiltonian tour*** in G is a closed walk W that visits every vertex of G exactly once, except for *at most one* vertex that W visits more than once.

   (a) Give an example of a graph that contains a ultra-Hamiltonian tour, but does not contain a Hamiltonian cycle (which visits every vertex exactly once).

   (b) ***Prove*** that it is NP-hard to determine whether a given graph contains a ultra-Hamiltonian tour.

4. An ***infra-Hamiltonian cycle*** in G is a closed walk W that visits every vertex of G exactly once, except for *at most one* vertex that W does not visit at all.

   (a) Give an example of a graph that contains a infra-Hamiltonian cycle, but does not contain a Hamiltonian cycle (which visits every vertex exactly once).

   (b) ***Prove*** that it is NP-hard to determine whether a given graph contains a infra-Hamiltonian cycle.

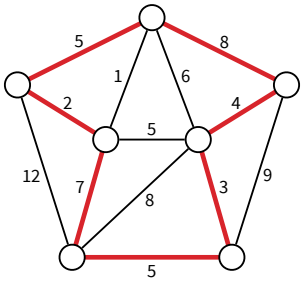
5. A ***quasi-satisfying assignment*** for a 3CNF boolean formula Φ is an assignment of truth values to the variables such that *at most one* clause in Φ does not contain a true literal. ***Prove*** that it is NP-hard to determine whether a given 3CNF boolean formula has a quasi-satisfying assignment.

6. A subset S of vertices in an undirected graph G is ***half-independent*** if each vertex in S is adjacent to *at most one* other vertex in S. Prove that finding the size of the largest half-independent set of vertices in a given undirected graph is NP-hard.

7. A subset S of vertices in an undirected graph G is ***sort-of-independent*** if if each vertex in S is adjacent to *at most 374* other vertices in S. Prove that finding the size of the largest sort-of-independent set of vertices in a given undirected graph is NP-hard.

8. A subset S of vertices in an undirected graph G is ***almost independent*** if at most 374 edges in G have both endpoints in S. Prove that finding the size of the largest almost-independent set of vertices in a given undirected graph is NP-hard.

9. Let G be an undirected graph with weighted edges. A ***heavy Hamiltonian cycle*** is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is more than half of the total weight of all edges in G. Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-hard.



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

10. (a) A ***tonian path*** in a graph G is a path that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian path is NP-hard.

    (b) A ***tonian cycle*** in a graph G is a cycle that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian cycle is NP-hard. *[Hint: Use part (a). Or not.]*

11. Prove that the following variants of SAT is NP-hard. *[Hint: Describe reductions from 3SAT.]*

    (a) Given a boolean formula Φ in conjunctive normal form, where *each variable appears in at most three clauses*, determine whether Φ has a satisfying assignment. *[Hint: First consider the variant where each variable appears in at most* ***five*** *clauses.]*

    (b) Given a boolean formula Φ in conjunctive normal form *and given one satisfying assignment for* Φ, determine whether Φ has at least one other satisfying assignment.

12. Jerry Springer and Maury Povich have decided not to compete with each other over scheduling guests during the next talk-show season. There is only one set of Weird People who either host would consider having on their show. The hosts want to divide the Weird People into two disjoint subsets: those to appear on Jerry's show, and those to appear on Maury's show. (Neither wants to "recycle" a guest that appeared on the other's show.)

Both Jerry and Maury have preferences about which Weird People they are particularly interested in. For example, Jerry wants at least one guest who fits the description "was abducted by a flying saucer". Thus, on his list of preferences, he writes "w_1 or w_3 or w_{45}", since weird people numbered 1, 3, and 45 are the only ones who fit that description. Jerry has other preferences as well, so he lists those also. Similarly, Maury might like to include at least one guest who "really enjoys Rice's theorem". Each potential guest may fall into any number of different categories, such as the person who enjoys Rice's theorem more than their involuntary flying-saucer voyage.

Jerry and Maury each prepare a list reflecting all of their preferences. Each list contains a collection of statements of the form "(w_i or w_j or w_k)". Your task is to prove that it is NP-hard to find an assignment of weird guests to the two shows that satisfies all of Jerry's preferences and all of Maury's preferences.

(a) The problem NoMixedClauses3Sat is the special case of 3Sat where the input formula cannot contain a clause with both a negated variable and a non-negated variable. Prove that NoMixedClauses3Sat is NP-hard. *[Hint: Reduce from the standard 3Sat problem.]*

(b) Describe a polynomial-time reduction from NoMixedClauses3Sat to 3Sat.

13. The president of Sham-Poobanana University is planning An Unofficial St. Brigid's Day party for the university staff.[1] His staff has a hierarchical structure; that is, the supervisor relation forms a directed, acyclic graph, with the president as the only source, and with an edge from person i to person j in the graph if and only if person i is an immediate supervisor of person j. (Many staff members have multiple positions, and thus have several immediate supervisors.) In order to make the party fun for all guests, the president wants to ensure that if a person i attends, then none of i's immediate supervisors can attend.

By mining each staff member's email and social media accounts, Sham-Poobanana University Human Resources has determined a "party-hound" rating for each staff member, which is a non-negative real number reflecting how likely it is that the person will leave the party wearing a monkey suit and a lampshade.

Show that it is NP-hard to determine a guest-list that *maximizes* the sum of the party-hound ratings of all invited guests, subject to the supervisor constraint.

[Hint: This problem can be solved in polynomial time when the input graph is a tree!]

14. Prove that the following problem (which we call Match) is NP-hard. The input is a finite set S of strings, all of the same length n, over the alphabet $\{0, 1, 2\}$. The problem is to determine whether there is a string $w \in \{0, 1\}^n$ such that for every string $s \in S$, the strings s and w have the same symbol in at least one position.

For example, given the set $S = \{01220, 21110, 21120, 00211, 11101\}$, the correct output is True, because the string $w = 01001$ matches the first three strings of S in the second position, and matches the last two strings of S in the last position. On the other hand, given the set $S = \{00, 11, 01, 10\}$, the correct output is False.

[Hint: Describe a reduction from SAT (or 3SAT)]

[1] As I'm sure you already know, St. Brigid of Kildare is one of the patron saints of Ireland, chicken and dairy farmers, and academics.

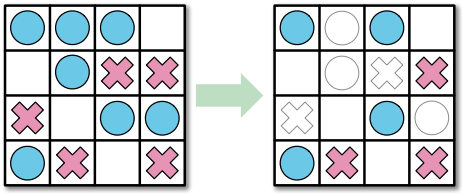
15. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions:

    (1) Every row contains at least one stone.

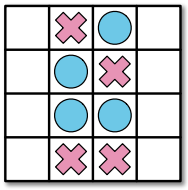
    (2) No column contains stones of both colors.

    For some initial configurations of stones, reaching this goal is impossible; see the example below.

    Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether this puzzle can be solved.



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

16. To celebrate the end of the semester, Professor Jarling wants to treat himself to an ice-cream cone at the *Polynomial House of Flavors*. For a fixed price, he can build a cone with as many scoops as he'd like. Because he has good balance (and because we want this problem to work out), Prof. Jarling can balance any number of scoops on top of the cone without it tipping over. He plans to eat the ice cream one scoop at a time, from top to bottom, and doesn't want more than one scoop of any flavor.

    However, he realizes that eating a scoop of bubblegum ice cream immediately after the scoop of potatoes-and-gravy ice cream would be unpalatable; these two flavors clearly should not be placed next to each other in the stack. He has other similar constraints; certain pairs of flavors cannot be adjacent in the stack.

    He'd like to get as much ice cream as he can for the one fee by building the tallest cone possible that meets his flavor-incompatibility constraints. Prove that Prof. Jarling's problem is NP-hard.

17. Prove that the following problems are NP-hard.

    (a) Given an undirected graph G, does G contain a simple path that visits all but 17 vertices?

    (b) Given an undirected graph G, does G have a spanning tree in which every node has degree at most 23?

    (c) Given an undirected graph G, does G have a spanning tree with at most 42 leaves?

18. Prove that the following problems are NP-hard.

    (a) Given an undirected graph G, is it possible to color the vertices of G with three different colors, so that at most 31337 edges have both endpoints the same color?

(b) Given an undirected graph G, is it possible to color the vertices of G with three different colors, so that each vertex has at most 8675309 neighbors with the same color?

19. At the end of every semester, Jeff needs to solve the following ExamDesign problem. He has a list of problems, and he knows for each problem which students will *really enjoy* that problem. He needs to choose a subset of problems for the exam such that for each student in the class, the exam includes at least one question that student will really enjoy. On the other hand, he does not want to spend the entire summer grading an exam with dozens of questions, so the exam must also contain as few questions as possible. Prove that the ExamDesign problem is NP-hard.

20. Which of the following results would resolve the P vs. NP question? Justify each answer with a short sentence or two.

(a) The construction of a polynomial time algorithm for some problem in NP.

(b) A polynomial-time reduction from 3Sat to the language $\{0^n 1^n \mid n \geq 0\}$.

(c) A polynomial-time reduction from $\{0^n 1^n \mid n \geq 0\}$ to 3Sat.

(d) A polynomial-time reduction from 3Color to MinVertexCover.

(e) The construction of a nondeterministic Turing machine that cannot be simulated by any deterministic Turing machine with the same running time.

Some useful NP-hard problems. You are welcome to use any of these in your own NP-hardness proofs, except of course for the specific problem you are trying to prove NP-hard. **The final exam will include a copy of this list.**

CircuitSat: Given a boolean circuit, are there any input values that make the circuit output True?

3Sat: Given a boolean formula in conjunctive normal form, with exactly three distinct literals per clause, does the formula have a satisfying assignment?

MaxIndependentSet: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MaxClique: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MinVertexCover: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

MinSetCover: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subcollection whose union is S?

MinHittingSet: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subset of S that intersects every subset S_i?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

HamiltonianPath: Given graph G (either directed or undirected), is there a path in G that visits every vertex exactly once?

HamiltonianCycle: Given a graph G (either directed or undirected), is there a cycle in G that visits every vertex exactly once?

TravelingSalesman: Given a graph G (either directed or undirected) with weighted edges, what is the minimum total weight of any Hamiltonian path/cycle in G?

LongestPath: Given a graph G (either directed or undirected, possibly with weighted edges), what is the length of the longest simple path in G?

SteinerTree: Given an undirected graph G with some of the vertices marked, what is the minimum number of edges in a subtree of G that contains every marked vertex?

SubsetSum: Given a set X of positive integers and an integer k, does X have a subset whose elements sum to k?

Partition: Given a set X of positive integers, can X be partitioned into two subsets with the same sum?

3Partition: Given a set X of $3n$ positive integers, can X be partitioned into n three-element subsets, all with the same sum?

IntegerLinearProgramming: Given a matrix $A \in \mathbb{Z}^{n \times d}$ and two vectors $b \in \mathbb{Z}^n$ and $c \in Z^d$, compute $\max\{c \cdot x \mid Ax \leq b, x \geq 0, x \in \mathbb{Z}^d\}$.

FeasibleILP: Given a matrix $A \in \mathbb{Z}^{n \times d}$ and a vector $b \in \mathbb{Z}^n$, determine whether the set of feasible integer points $\max\{x \in \mathbb{Z}^d \mid Ax \leq b, x \geq 0\}$ is empty.

Draughts: Given an $n \times n$ international draughts configuration, what is the largest number of pieces that can (and therefore must) be captured in a single move?

SteamedHams: Aurora borealis? At this time of year, at this time of day, in this part of the country, localized entirely within your kitchen? May I see it?

Turing Machines and Undecidability

The only undecidability questions on this semester's final exam will be True/False or short-answer, but the following problems might still be useful to build intuition.

For each of the following languages, either ***sketch*** an algorithm to decide that language or ***prove*** that the language is undecidable, using a diagonalization argument, a reduction argument, Rice's theorem, closure properties, or some combination of the above. Recall that w^R denotes the reversal of string w.

1. $\emptyset$
2. $\{\mathtt{0}^n\mathtt{1}^n\mathtt{2}^n \mid n \geq 0\}$
3. $\{A \in \{\mathtt{0},\mathtt{1}\}^{n\times n} \mid n \geq 0 \text{ and } A \text{ is the adjacency matrix of a dag with } n \text{ vertices}\}$
4. $\{A \in \{\mathtt{0},\mathtt{1}\}^{n\times n} \mid n \geq 0 \text{ and } A \text{ is the adjacency matrix of a 3-colorable graph with } n \text{ vertices}\}$
5. $\{\langle M\rangle \mid M \text{ accepts } \langle M\rangle^R\}$
6. $\{\langle M\rangle \mid M \text{ accepts } \langle M\rangle^R\} \cap \{\langle M\rangle \mid M \text{ rejects } \langle M\rangle^R\}$
7. $\{\langle M\rangle \# w \mid M \text{ accepts } ww^R\}$
8. $\{\langle M\rangle \mid M \text{ accepts } \texttt{RICESTHEOREM}\}$
9. $\{\langle M\rangle \mid M \text{ rejects } \texttt{RICESTHEOREM}\}$
10. $\{\langle M\rangle \mid M \text{ accepts at least one palindrome}\}$
11. $\Sigma^* \setminus \{\langle M\rangle \mid M \text{ accepts at least one palindrome}\}$
12. $\{\langle M\rangle \mid M \text{ rejects at least one palindrome}\}$
13. $\{\langle M\rangle \mid M \text{ accepts exactly one string of length } \ell, \text{ for each integer } \ell \geq 0\}$
14. $\{\langle M\rangle \mid \textsc{Accept}(M) \text{ has an infinite fooling set}\}$
15. $\{\langle M\rangle \# \langle M'\rangle \mid \textsc{Accept}(M) \cap \textsc{Accept}(M') \neq \emptyset\}$
16. $\{\langle M\rangle \# \langle M'\rangle \mid \textsc{Accept}(M) \oplus \textsc{Reject}(M') \neq \emptyset\}$ — Here $\oplus$ means exclusive-or.

Some useful undecidable problems. You are welcome to use any of these in your own undecidability proofs, except of course for the specific problem you are trying to prove undecidable.

$$\begin{aligned}
\textsc{SelfReject} &:= \{\langle M\rangle \mid M \text{ rejects } \langle M\rangle\} \\
\textsc{SelfAccept} &:= \{\langle M\rangle \mid M \text{ accepts } \langle M\rangle\} \\
\textsc{SelfHalt} &:= \{\langle M\rangle \mid M \text{ halts on } \langle M\rangle\} \\
\textsc{SelfDiverge} &:= \{\langle M\rangle \mid M \text{ does not halt on } \langle M\rangle\}
\end{aligned}$$

$$\begin{aligned}
\textsc{Reject} &:= \{\langle M\rangle \# w \mid M \text{ rejects } w\} \\
\textsc{Accept} &:= \{\langle M\rangle \# w \mid M \text{ accepts } w\} \\
\textsc{Halt} &:= \{\langle M\rangle \# w \mid M \text{ halts on } w\} \\
\textsc{Diverge} &:= \{\langle M\rangle \# w \mid M \text{ does not halt on } w\}
\end{aligned}$$

$$\begin{aligned}
\textsc{NeverReject} &:= \{\langle M\rangle \mid \textsc{Reject}(M) = \emptyset\} \\
\textsc{NeverAccept} &:= \{\langle M\rangle \mid \textsc{Accept}(M) = \emptyset\} \\
\textsc{NeverHalt} &:= \{\langle M\rangle \mid \textsc{Halt}(M) = \emptyset\} \\
\textsc{NeverDiverge} &:= \{\langle M\rangle \mid \textsc{Diverge}(M) = \emptyset\}
\end{aligned}$$

You have 180 minutes to answer six numbered questions.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, there are two boxes in the answer booklet labeled "Yes" and "No". Check "Yes" if the statement is ***always*** true and "No" otherwise, and give a ***brief*** (at most one short sentence) explanation of your answer. **Assume $P \neq NP$.** If there is any other ambiguity or uncertainty about an answer, check "No". For example:

   - $x + y = 5$

     [X] Yes [ ] No Suppose $x = 3$ and $y = 4$.

   - 3SAT can be solved in polynomial time.

     [ ] Yes [X] No 3SAT is NP-hard.

   - If P = NP then Jeff is the Queen of England.

     [X] Yes [ ] No The hypothesis is false, so the implication is true.

   Read each statement *very* carefully; some of these are deliberately subtle!

   (a) Which of the following statements are true?

   - The solution to the recurrence $T(n) = 8T(n/2) + O(n^2)$ is $T(n) = O(n^2)$.
   - The solution to the recurrence $T(n) = 2T(n/8) + O(n^2)$ is $T(n) = O(n^2)$.
   - Every directed acyclic graph contains at least one sink.
   - Given *any* undirected graph G, we can compute a spanning tree of G in $O(V + E)$ time using whatever-first search.
   - Suppose $A[1..n]$ is an array of integers. Consider the following recursive function:

$$What(i,j) = \begin{cases} 0 & \text{if } i < 0 \text{ or } i > n \\ 0 & \text{if } j < 0 \text{ or } j > n \\ \max \left\{ \begin{array}{c} What(i, j-1) \\ What(i-1, j) \\ A[i] \cdot A[j] + What(i+1, j+1) \end{array} \right\} & \text{otherwise} \end{cases}$$

   We can memoize this function into an array $What[0..n, 0..n]$ in $O(n^2)$ time, by increasing i in the outer loop and increasing j in the inner loop.

Problem 1 continues onto the next page.

1. *[continued]*

   (b) Which of the following statements are true for ***at least one*** language $L \subseteq \{0,1\}^*$?

   - $L^* = (L^*)^*$
   - L is decidable, but L^* is undecidable.
   - L is neither regular nor NP-hard.
   - L is in P, and L has an infinite fooling set.
   - The language $\{\langle M \rangle \mid M \text{ accepts } L\}$ is undecidable.

   (c) Consider the following pair of languages:

   - DirHamPath $:= \{G \mid G \text{ is a directed graph with a Hamiltonian path}\}$
   - Acyclic $:= \{G \mid G \text{ is a directed acyclic graph}\}$

   (For concreteness, assume that in both of these languages, graphs are represented by their adjacency matrices.) Which of the following statements are true, assuming P $\neq$ NP?

   - Acyclic $\in$ NP
   - Acyclic $\cap$ DirHamPath $\in$ P
   - DirHamPath is decidable.
   - A polynomial-time reduction from DirHamPath to Acyclic would imply P=NP.
   - A polynomial-time reduction from Acyclic to DirHamPath would imply P=NP.

   (d) Suppose there is a ***polynomial-time*** reduction from some language A over the alphabet $\{0,1\}$ to some other language B over the alphabet $\{0,1\}$. Which of the following statements are ***always*** true, assuming P $\neq$ NP?

   - A is a subset of B.
   - If $B \in P$, then $A \in P$.
   - If B is NP-hard, then A is NP-hard.
   - If B is regular, then A is regular.
   - If B is regular, then A is decidable.

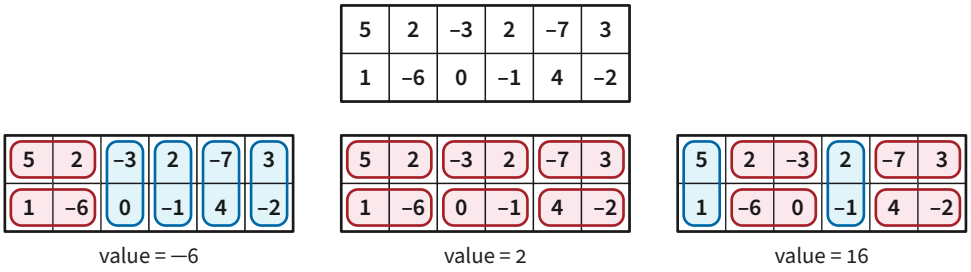
2. Describe and analyze an algorithm to determine whether the language accepted by a given DFA is finite or infinite. You can assume the input alphabet of the DFA is $\{0,1\}$. *[Hint: DFAs are directed graphs.]*

3. Suppose you are asked to tile a $2 \times n$ grid of squares with dominos (1×2 rectangles). Each domino must cover exactly two grid squares, either horizontally or vertically, and each grid square must be covered by exactly one domino.

   Each grid square is worth some number of points, which could be positive, negative, or zero. The ***value*** of a domino tiling is the sum of the points in squares covered by vertical dominos, ***minus*** the sum of the points in squares covered by horizontal dominos.

   Describe an algorithm to compute the largest possible value of a domino tiling of a given $2 \times n$ grid. Your input is an array $Points[1..2, 1..n]$ of point values.

   As an example, here are three domino tilings of the same 2×6 grid, along with their values. The third tiling is optimal; no other tiling of this grid has larger value. Thus, given this 2×6 grid as input, your algorithm should return the integer 16.



4. Submit a solution to *exactly one* of the following problems. Don't forget to tell us which problem you've chosen!

   (a) Let Φ be a boolean formula in conjunctive normal form, with exactly three literals per clause (or in other words, an instance of 3SAT). ***Prove*** that it is NP-hard to decide whether Φ has a satisfying assignment in which *exactly half* of the variables are TRUE.

   (b) Let $G = (V, E)$ be an arbitrary directed graph whose edges have colors. A *rainbow Hamiltonian cycle* in G is a cycle that visits every vertex of G exactly once, in which no pair of consecutive edges have the same color. ***Prove*** that it is NP-hard to decide whether G has a rainbow Hamiltonian cycle.

   (In fact, both of these problems are NP-hard, but we only want a proof for one of them.)

5. Suppose you are given a height map of a mountain, in the form of an $n \times n$ grid of evenly spaced points, each labeled with an elevation value. You can safely hike directly from any point to any neighbor immediately north, south, east, or west, but only if the elevations of those two points differ by at most Δ. (The value of Δ depends on your hiking experience and your physical condition.)

   Describe and analyze an algorithm to determine the longest hike from some point s to some other point t, where the hike consists of an uphill climb (where elevations must increase at each step) followed by a downhill climb (where elevations must decrease at each step). Your input consists of an array $Elevation[1..n, 1..n]$ of elevation values, the starting point s, the target point t, and the parameter Δ.

6. Recall that a ***run*** in a string $w \in \{0,1\}^*$ is a maximal substring of w whose characters are all equal. For example, the string `00011111110000` is the concatenation of three runs:

$$\texttt{00011111110000} = \texttt{000} \bullet \texttt{1111111} \bullet \texttt{0000}$$

   (a) Let L_a denote the set of all strings in $\{0,1\}^*$ where every `0` is followed immediately by at least one `1`.

   For example, L_a contains the strings `010111` and `1111` and the empty string ε, but does not contain either `001100` or `1111110`.

   - Describe a DFA or NFA that accepts L_a ***and***
   - Give a regular expression that describes L_a.

   (You do not need to prove that your answers are correct.)

   (b) Let L_b denote the set of all strings in $\{0,1\}^*$ whose run lengths are increasing; that is, every run except the last is followed immediately by a *longer* run.

   For example, L_b contains the strings `0110001111` and `1100000` and `000` and the empty string ε, but does not contain either `000111` or `100011`.

   Prove that L_b is not a regular language.

You have 180 minutes to answer six numbered questions.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. Recall that a ***run*** in a string $w \in \{0,1\}^*$ is a maximal substring of w whose characters are all equal. For example, the string `00011111110000` is the concatenation of three runs:

$$\texttt{00011111110000} = \texttt{000} \bullet \texttt{1111111} \bullet \texttt{0000}$$

   (a) Let L_a denote the set of all non-empty strings in $\{0,1\}^*$ where the length of the first run is equal to the number of runs. For example, L_a contains the strings `0` and `1100000` and `0001110`, but does not contain `000111` or `100011` or the empty string ε (because it has no first run).

   Prove that L_a is not a regular language.

   (b) Let L_b denote the set of all strings in $\{0,1\}^*$ that contain an even number of odd-length runs. For example, L_b contains the strings `010111` and `1111` and the empty string ε, but does not contain either `0011100` or `11110`.

   - Describe a DFA or NFA that accepts L_b ***and***
   - Give a regular expression that describes L_b.

   (You do not need to prove that your answers are correct.)

2. Aladdin and Badroulbadour are playing a cooperative game. Each player has an array of positive integers, arranged in a row of squares from left to right. Each player has a token, which starts at the leftmost square of their row; their goal is to move *both* tokens onto the rightmost squares at the same time.

   On each turn, *both* players move their tokens *in the same direction*, either left or right. The distance each token travels is equal to the number under that token at the beginning of the turn. For example, if a token starts on a square labeled 5, then it moves either five squares to the right or five squares to the left. If *either* token moves past either end of its row, then both players immediately lose.

   For example, if Aladdin and Badroulbadour are given the arrays

A:	7	5	4	1	2	3	3	2	3	1	4	2
B:	5	1	2	4	7	3	5	2	4	6	3	1

   they can win the game by moving right, left, left, right, right, left, right. On the other hand, if they are given the arrays

A:	2	3	5	1	3
B:	3	4	1	2	1

   they cannot win the game. (The first move must be to the right; then Aladdin's token moves out of bounds on the second turn.)

   Describe and analyze an algorithm to determine whether Aladdin and Badroulbadour can solve their puzzle, given the input arrays $A[1..n]$ and $B[1..n]$.

3. Submit a solution to ***exactly one*** of the following problems. Don't forget to tell us which problem you've chosen!

   (a) Let $G = (V, E)$ be an arbitrary undirected graph. A subset $S \subseteq V$ of vertices is *mostly independent* if more than half the vertices of S have no neighbors in S. ***Prove*** that finding the largest mostly independent set in G is NP-hard.

   (b) ***Prove*** that the following problem is NP-hard: Given an undirected graph G, find the largest integer k such that G contains *two disjoint* independent sets of size k.

   (In fact, both of these problems are NP-hard, but we only want a proof for one of them.)

4. Recall that a *palindrome* is any string that is equal to its reversal, like REDIVIDER or POOP.

   (a) Describe and analyze an algorithm to find the length of the longest subsequence of a given string that is a palindrome.

   (b) A *double palindrome* is the concatenation of two *non-empty* palindromes, like REFEREE = REFER • EE or POOPREDIVIDER = POOP • REDIVIDER. Describe and analyze an algorithm to find the length of the longest subsequence of a given string that is a *double* palindrome. *[Hint: Use your algorithm from part (a).]*

   For both algorithms, the input is an array $A[1..n]$, and the output is an integer. For example, given the input string MAYBEDYNAMICPROGRAMMING, your algorithm for part (a) should return 7 (for the subsequences NMRORMN and MAYBYAM, among others), and your algorithm for part (b) should return 12 (for the subsequence MAYBYAMIRORI).

5. You have a collection of n lockboxes and m gold keys. Each key unlocks *at most* one box. Without a matching key, the only way to open a box is to smash it with a hammer. Your baby brother has locked all your keys inside the boxes! Luckily, you know which keys (if any) are inside each box.

   (a) Your baby brother has found the hammer and is eagerly eyeing one of the boxes. Describe and analyze an algorithm to determine if it is possible to retrieve all the keys without smashing any box except the one your brother has chosen.

   (b) Describe and analyze an algorithm to compute the minimum number of boxes that must be smashed to retrieve all the keys.

Problem 6 begins on the next page.

6. For each statement below, there are two boxes in the answer booklet labeled "Yes" and "No". Check "Yes" if the statement is ***always*** true and "No" otherwise, and give a ***brief*** (at most one short sentence) explanation of your answer. **Assume $P \neq NP$.** If there is any other ambiguity or uncertainty about an answer, check "No". For example:

   - $x + y = 5$

     [X] Yes [ ] No Suppose $x = 3$ and $y = 4$.

   - 3SAT can be solved in polynomial time.

     [ ] Yes [X] No 3SAT is NP-hard.

   - If P = NP then Jeff is the Queen of England.

     [X] Yes [ ] No The hypothesis is false, so the implication is true.

   Read each statement *very* carefully; some of these are deliberately subtle!

   (a) Which of the following statements are true for ***all*** languages $L \subseteq \{0,1\}^*$?

   - $L^* = (L^*)^*$
   - If L is decidable, then L^* is decidable.
   - L is either regular or NP-hard.
   - If L is undecidable, then L has an infinite fooling set.
   - The language $\{\langle M \rangle \mid M \text{ decides } L\}$ is undecidable.

   (b) Which of the following statements are true?

   - The solution to the recurrence $T(n) = 4T(n/4) + O(n)$ is $T(n) = O(n \log n)$.
   - The solution to the recurrence $T(n) = 4T(n/4) + O(n^2)$ is $T(n) = O(n^2 \log n)$.
   - Every directed acyclic graph contains at most one source and at most one sink.
   - depth-first search explores every path from the source vertex s to every other vertex in the input graph.
   - Suppose $A[1..n]$ is an array of integers. Consider the following recursive function:

$$Huh(i,j) = \begin{cases} 0 & \text{if } i < 0 \text{ or } j > n \\ \max \begin{Bmatrix} Huh(i, j+1) \\ Huh(i-1, j) \\ A[i] \cdot A[j] + Huh(i-1, j+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

   We can compute $Huh(n, 0)$ by memoizing this function into an array $Huh[0..n, 0..n]$ in $O(n^2)$ time, increasing i in the outer loop and increasing j in the inner loop.

Problem 6 continues onto the next page.

1. *[continued]*

(c) Suppose we want to prove that the following language is undecidable.

$$\textsc{Muggle} := \left\{ \langle M \rangle \;\middle|\; M \text{ accepts } \texttt{SCIENCE} \text{ but rejects } \texttt{MAGIC} \right\}$$

Professor Potter, your instructor in Defense Against Models of Computation and Other Dark Arts, suggests a reduction from the standard halting language

$$\textsc{Halt} := \left\{ (\langle M \rangle, w) \;\middle|\; M \text{ halts on input } w \right\}.$$

Specifically, suppose there is a Turing machine DetectoMuggletum that decides Muggle. Professor Potter claims that the following algorithm decides Halt.

```
DecideHalt(⟨M⟩, w):
  Write code for the following algorithm:
      RubberDuck(x):
        run M on input w
        ⟨⟨ignore the output of M⟩⟩
        if x = MAGIC
            return False
        else
            return True
  return DetectoMuggletum(⟨RubberDuck⟩)
```

Which of the following statements ***must be*** true ***for all*** inputs $\langle M \rangle \# w$?

- If M accepts w, then RubberDuck accepts `MAGIC`.
- If M diverges on w, then RubberDuck rejects `MAGIC`.
- If M accepts w, then DetectoMuggletum accepts $\langle$RubberDuck$\rangle$.
- If M diverges on w, then DecideHalt rejects $(\langle M \rangle, w)$.
- DecideHalt decides the language Halt. (That is, Professor Potter's reduction is actually correct.)

(d) Suppose there is a ***polynomial-time*** reduction from some language $A \subseteq \{\texttt{0}, \texttt{1}\}$ reduces to some other language $B \subseteq \{\texttt{0}, \texttt{1}\}$. Which of the following statements are true, assuming P $\neq$ NP?

- $A \cap B \neq \emptyset$.
- There is an algorithm to transform any Python program that solves B in polynomial time into a Python program that solves A in polynomial time.
- If B is NP-hard, then A is NP-hard.
- If B is decidable, then A is decidable.
- If a Turing machine M accepts every string in B, the *same* Turing machine M also accepts every string in A.

You have 180 minutes to answer six numbered questions.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, there are two boxes in the answer booklet labeled "Yes" and "No". Check "Yes" if the statement is ***always*** true and "No" otherwise, and give a ***brief*** (at most one short sentence) explanation of your answer. **Assume $P \neq NP$.** If there is any other ambiguity or uncertainty about an answer, check "No". For example:

   - $x + y = 5$

     [Yes] [X No] Suppose $x = 3$ and $y = 4$.

   - 3SAT can be solved in polynomial time.

     [Yes] [X No] 3SAT is NP-hard.

   - If P = NP then Jeff is the Queen of England.

     [X Yes] [No] The hypothesis is false, so the implication is true.

   Read each statement *very* carefully; some of these are deliberately subtle!

   (a) Which of the following statements are true?

   - The solution to the recurrence $T(n) = 2T(n/2) + O(\sqrt{n})$ is $T(n) = O(\sqrt{n}\log n)$.
   - The solution to the recurrence $T(n) = 2T(n/4) + T(n/3) + O(n)$ is $T(n) = O(n \log n)$.
   - There is a forest with 374 vertices and 374 edges. (Recall that a *forest* is an undirected graph with no cycles.)
   - Given any directed graph G whose edges have positive weights, we can compute shortest paths from one vertex s to every other vertex of G in $O(VE)$ time using Bellman-Ford.
   - Suppose $A[1..n]$ is an array of integers. Consider the following recursive function:

$$Oops(i,k) = \begin{cases} 0 & \text{if } i > k \\ 1 & \text{if } i = k \\ \max\left\{A[i]\cdot A[j]\cdot A[k] + Oops(i,j) + Oops(j,k) \;\middle|\; i \le j \le k\right\} & \text{otherwise} \end{cases}$$

   We can compute $Oops(1,n)$ by memoizing this function into a two-dimensional array $Oops[1..n, 1..n]$, which we fill by decreasing i in the outer loop and increasing k in the inner loop, in $O(n^2)$ time.

Problem 1 continues onto the next page.

1. *[continued]*

(b) Which of the following statements are true for ***every*** language $L \subseteq \{0,1\}^*$?

- Either L is regular or L is infinite.
- L^* is regular.
- L contains arbitrarily long strings.
- If L is decidable, then its complement $\overline{L}$ is also decidable.
- If L is undecidable then L^* is undecidable.

(c) Consider the following pair of languages:

- 3Color $= \{G \mid G$ is an undirected graph with a proper 3-coloring$\}$
- Forest $= \{G \mid G$ is an undirected graph with no cycles$\}$

(For concreteness, assume that in both of these languages, graphs are represented by their adjacency matrices.) Which of the following statements are true, assuming P $\neq$ NP?

- Forest $\in P$
- Forest $\cap$ 3Color $\in P$
- 3Color is undecidable.
- 3Color is regular.
- A polynomial-time reduction from 3Color to Forest would imply P = NP.

(d) Suppose there is a ***polynomial-time*** reduction R from some language $A \in \{0,1\}^*$ to some other language $B \in \{0,1\}^*$. Which if the following statements are ***always*** true, assuming P $\neq$ NP?

- The reduction transforms every string in A into a string in B.
- The reduction transforms every string in B into a string in A.
- If A is infinite, then B is infinite.
- If B is undecidable, then A is undecidable.
- If B is undecidable, then A is NP-hard.

Problem 2 appears on the next page.

2. Several years after graduating from Sham-Poobanana University, you decide to open a one-day pop-up art gallery selling NFTs, using the following dynamic pricing strategy.

   All NFTs art your gallery have the same advertised price, which you set at the start of the day, but which you can *decrease* later. Customers visit your gallery one at a time. If a customer is willing to pay your current advertised price, they buy one NFT at that price. On the other hand, if your advertised price is too high, the customer will suggest a lower price that they are willing to pay. If you refuse to lower your advertised price, the customer will leave without buying anything. If you agree to lower your advertised price to match their offer, the customer will buy one NFT at the new lower price. Whenever you lower your advertised price, your new lower price stays in effect until you lower it again, or until the end of the day. *You can never increase your advertised price.*

   You know your customers extremely well, so you can accurately predict both when each customer will come to the gallery, and how much each customer is willing to pay for one of your NFTs.

   Describe and analyze an algorithm that computes the maximum amount of money you can earn using this dynamic pricing strategy. Your input consists of an array $Value[1..n]$, where $Value[i]$ is the amount that the ith customer (in chronological order) is willing to pay for one NFT.

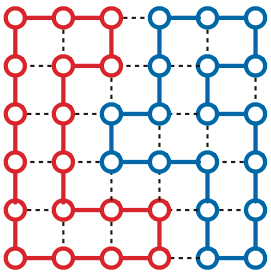
   For example, if the input array is $[5,3,1,4,2]$, your algorithm should output 13, because you can earn $5+3+0+3+2=13$ dollars using the prices $[5,3,3,3,2]$, and this is optimal.

3. Submit a solution to ***exactly one*** of the following problems. Don't forget to tell us which problem you've chosen!

   (a) A *mostly-3-coloring* of a graph $G=(V,E)$ is any function $C : V \to \{red, yellow, blue, none\}$ such that $C(v) = none$ for less than half the vertices in V. A mostly-3-coloring C is *proper* if, for every edge $uv \in E$, either $C(u) \neq C(v)$ or $C(u) = C(v) = none$.

   Prove that it is NP-hard to determine whether a given graph G has a proper mostly-3-coloring.

   (b) A *Hamiltonian bicycle* in a graph G is a pair of simple cycles in G, with identical lengths, such that every vertex of G lies on exactly one of the two cycles.



A Hamiltonian bicycle in the 6×6 grid graph.

   Prove that it is NP-hard to determine whether a given graph G has a Hamiltonian bicycle.

(In fact, both of these problems are NP-hard, but we only want a proof for one of them.)

4. (a) Let L_a denote the set of all strings in $\{0,1,2\}^*$ that do not contain any symbol twice in a row. For example, this language includes the strings `0101201202`, `1010101`, `2`, and the empty string ε, but it does not include the strings `01210220` or `11`.
   - Describe a DFA or NFA that accepts L_a ***and***
   - Give a regular expression that describes L_a.

   (You do not need to prove that your answers are correct.)

   (b) Let L_b denote the set of all strings $w \in \{0,1,2\}^*$ such that $\#(0,w)+\#(1,w) = \#(2,w)$. For example, this language includes the strings `01012222` and `20221020` and the empty string ε, but it does not include the string `01212` or `2120210`.

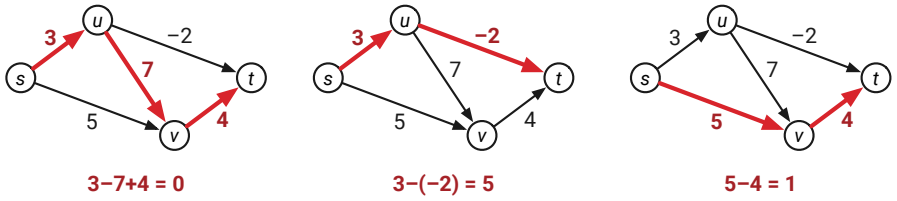
   Prove that L_b is not a regular language.

5. Let G be a directed **acyclic** graph, in which every edge $e \in E$ has a weight $w(e)$, which could be positive, negative, or zero. We define the ***alternating length*** of any path in G to be the weight of the first edge, ***minus*** the weight of the second edge, ***plus*** the weight of the third edge, and so on. More formally, for any path $P = v_0 \to v_1 \to v_2 \to \cdots \to v_\ell$ in G, we define

$$AltLen(P) = \sum_{i=0}^{\ell-1} (-1)^i \cdot w(v_i \to v_{i+1}).$$

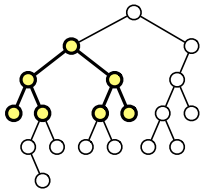
   Describe an algorithm to find a path from s to t with the largest alternating length, given the graph G, the edge weights $w(e)$, and vertices s and t as input.

   For example, given the graph shown below, your algorithm should return 5, which is the alternating length of the path $s \to u \to t$.



6. Recall that the *depth* of a vertex v in a binary tree T is the length of the unique path in T from v to the root of T. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. An *internal subtree* of a binary tree T is any connected subgraph of T.

   Describe and analyze a recursive algorithm to compute the *largest complete internal subtree* of a given binary tree. Your algorithm should return both the root and the depth of this internal subtree.



The largest complete internal subtree of this binary tree has depth 2.

You have 180 minutes to answer six numbered questions.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, there are two boxes in the answer booklet labeled "Yes" and "No". Check "Yes" if the statement is ***always*** true and "No" otherwise, and give a ***brief*** (at most one short sentence) explanation of your answer. **Assume $P \neq NP$.** If there is any other ambiguity or uncertainty about an answer, check "No". For example:

   - $x + y = 5$

     [Yes] [X No] Suppose $x = 3$ and $y = 4$.

   - 3SAT can be solved in polynomial time.

     [Yes] [X No] 3SAT is NP-hard.

   - If P = NP then Jeff is the Queen of England.

     [X Yes] [No] The hypothesis is false, so the implication is true.

   Read each statement *very* carefully; some of these are deliberately subtle!

   (a) Which of the following statements are true?

   - The solution to the recurrence $T(n) = 3T(n/3) + O(n^2)$ is $T(n) = O(n^2)$.
   - The solution to the recurrence $T(n) = 9T(n/3) + O(n)$ is $T(n) = O(n^2)$.
   - There is a forest with 374 vertices and 225 edges. (Recall that a *forest* is an undirected graph with no cycles.)
   - Given any directed graph G whose edges have positive weights, we can compute shortest paths from one vertex s to every other vertex of G in $O(VE)$ time using Bellman-Ford.
   - Suppose $A[1..n]$ is an array of integers. Consider the following recursive function:

$$Rizz(i,k) = \begin{cases} 0 & \text{if } i > k \\ 1 & \text{if } i = k \\ \max\left\{ \begin{array}{c} Rizz(i,j-1) + Rizz(j+1,k) \\ + A[i] \cdot A[j] \cdot A[k] \end{array} \;\middle|\; i \le j \le k \right\} & \text{otherwise} \end{cases}$$

   We can compute $Rizz(1,n)$ by memoizing this function into a two-dimensional array $Rizz[1..n, 1..n]$, which we fill by decreasing i in the outer loop and increasing k in the inner loop, in $O(n^2)$ time.

Problem 1 continues onto the next page.

1. *[continued]*

(b) Which of the following statements are true for ***at least one*** language $L \subseteq \{0,1\}^*$?

- $(L^*)^*$ is finite.
- L is decidable but its complement $\overline{L}$ is undecidable.
- $\{\langle M \rangle \mid M \text{ accepts } L\}$ is undecidable.
- L is the intersection of two NP-hard languages and L is finite.
- There is a polynomial-time reduction from L to the halting problem.

(c) Consider the following pair of languages:

- TREE $= \{G \mid G$ is a connected undirected graph with no cycles$\}$
- HAMPATH $= \{G \mid G$ is an undirected graph that contains a Hamiltonian path$\}$

(For concreteness, assume that in both of these languages, graphs are represented by their adjacency matrices.) Which of the following statements are true, assuming P $\neq$ NP?

- TREE is NP-hard.
- TREE $\cap$ HAMPATH is NP-hard.
- TREE $\cup$ HAMPATH is NP-hard.
- HAMPATH is undecidable.
- A reduction from TREE to HAMPATH would imply P $=$ NP.

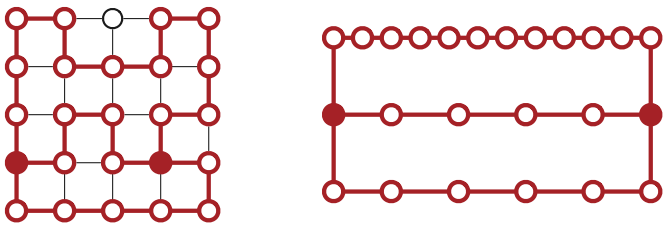
(d) Suppose there is a ***polynomial-time*** reduction R from some language $A \in \{0,1\}^*$ to some other language $B \in \{0,1\}^*$. Which if the following statements are ***always*** true, assuming P $\neq$ NP?

- Problem B is NP-hard.
- If A is finite, then B is finite.
- If A is NP-hard, then B is NP-hard.
- If A is undecidable, then B is undecidable.
- If $A \in P$, then $B \in P$.

Problems 2–6 appear on the next two pages.

2. Submit a solution to ***exactly one*** of the following problems.

   (a) A *theta-graph* is a connected undirected graph in which two vertices have degree 3, and all other vertices have degree 2. Equivalently, a theta-graph is the union of three undirected paths that have the same endpoints, but no other vertices in common. The *size* of a theta-graph is the total number of vertices.



The 5×5 grid grad contains a theta-subgraph of size 24.

   Prove that is is NP-hard to compute the size of the largest theta-graph that is a subgraph of a given undirected graph G.

   (b) A *clique-partition* of a graph $G = (V, E)$ is a partition of the vertices V into disjoint subsets $V_1 \cup V_2 \cup \cdots \cup V_k$, such that for each index i, every pair of vertices in subset V_i is connected by an edge in G. The *size* of a clique partition is the number of subsets V_i.

   Prove that it is NP-hard to compute the minimum-size clique partition of a given undirected graph G.

   In fact, both of these problems are NP-hard, but we only want a proof for one of them. Don't forget to tell us which problem you've chosen!

3. A *triumph* in a sequence of integers (from the Latin *tri-* meaning "three" and *-umph* meaning "bodacious") is a consecutive triple of sequence elements whose sum is a multiple of 3. For example, the sequence

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

   contains five triumphs (indicated by lines above and below).

   We say that one sequence A is *more triumphant* (or *less heinous*) than another sequence B if there are more triumphs in A than in B.

   Describe and analyze an algorithm to compute the number of triumphs in the most triumphant (or equivalently, least heinous) subsequence of a given array $A[1..n]$ of integers.

   For example, given the input array $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21 \rangle$, your algorithm should return the integer 4, which is the number of triumphs in the most triumphant subsequence $\langle 0, 1, 2, 3, 8, 13, 21 \rangle$. Excellent!

Problems 4–6 appear on the next page.

4. Suppose we are given a directed graph $G = (V, E)$, where every edge $e \in E$ has a *positive* weight $w(e)$, along with two vertices s and t.

   (a) Suppose each *vertex* of G is colored either orange, green, or purple. Describe and analyze an algorithm to find the shortest walk from s to t in G that never visits two consecutive *vertices* with the same color.

   (b) Now suppose each *edge* of G is colored either orange, green, or purple. Describe and analyze an algorithm to find the shortest walk from s to t in G that never traverses two consecutive *edges* with the same color.

5. let T be a *full* binary tree, meaning that every node has either two children or no children.

   - Recall that the *height* of a vertex v in T is the length of the longest path in T from v down to a leaf. In particular, every leaf of T has height zero.
   - A vertex v is *AVL-balanced* if v is a leaf, or if the heights of v's children differ by at most 1. (You might recall from CS 225 that an ***AVL-tree*** is a binary search tree in which *every* vertex is AVL-balanced.)

   Describe and analyze an algorithm to compute the number of AVL-balanced vertices in T.

6. (a) Let L_a denote the set of all strings $w \in \{0, 1, 2\}^*$ such that $\#(1, w) + 2 \cdot \#(2, w)$ is divisible by 3. For example, L_a contains the strings `0012` and `20210202` and the empty string ε, but L_a does not include the strings `121` or `0122210`.

   Describe a DFA of NFA that accepts L_a. (You do not need to prove that your answer is correct.)

   (b) Let L_b denote the set of all strings $w \in \{0, 1, 2\}^*$ such that no two symbols appear the same number of times, or in other words, the integers $\#(0, w)$ and $\#(1, w)$ and $\#(2, w)$ are all different. For example, L_b contains the strings `110212` and `20220`, but L_b does not include the string `01212` or `2120210` or the empty string ε.

   Prove that L_b is not a regular language.