• Each student must submit individual solutions for this homework. For all future homeworks, groups of up to three students can submit joint solutions.

• Submit your solutions electronically on the course Gradescope site as PDF files. Submit a separate PDF file for each numbered problem. If you plan to typeset your solutions, please 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, not just a phone camera).

• You are not required to sign up on Gradescope (or Piazza) with your real name and your illinois.edu email address; you may use any email address and alias of your choice. However, to give you credit for the homework, we need to know who Gradescope thinks you are. Please fill out the web form linked from the course web page.

---

Some important course policies

• You may use any source at your disposal—paper, electronic, or human—but you must cite every source that you use, and you must write everything yourself in your own words. See the academic integrity policies on the course web site for more details.

• The answer “I don’t know” (and nothing else) is worth 25% partial credit on any required problem or subproblem, on any homework or exam. We will accept synonyms like “No idea” or “WTF” or “¯\_(ツ)_/¯”, but you must write something.

• Avoid the Three Deadly Sins! Any homework or exam solution that breaks any of the following rules will be given an automatic zero, unless the solution is otherwise perfect. Yes, we really mean it. We’re not trying to be scary or petty (Honest!), but we do want to break a few common bad habits that seriously impede mastery of the course material.

  – Always give complete solutions, not just examples.
  – Always declare all your variables, in English. In particular, always describe the specific problem your algorithm is supposed to solve.
  – Never use weak induction.

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. The famous Czech professor Jiřina Z. Džunglová has a favorite 23-node binary tree, in which each node is labeled with a unique letter of the alphabet. Preorder and inorder traversals of the tree visit the nodes in the following order:

- Inorder: P E U V B G X N I Y F O J L R H D C S A M Z T

(a) List the nodes in Professor Džunglová’s tree in post-order.
(b) Draw Professor Džunglová’s tree.

2. The complement \( w^c \) of a string \( w \in \{0,1\}^* \) is obtained from \( w \) by replacing every 0 in \( w \) with a 1 and vice versa; for example, \( 111011000100^c = 000100111011 \). The complement function is formally defined as follows:

\[
\begin{align*}
\text{if } w &= \varepsilon \\
\text{then } w^c &= \varepsilon \\
\text{if } w &= 0x \\
\text{then } w^c &= 1 \cdot x^c \\
\text{if } w &= 1x \\
\text{then } w^c &= 0 \cdot x^c
\end{align*}
\]

(a) Prove by induction that \( |w| = |w^c| \) for every string \( w \).
(b) Prove by induction that \( (x \cdot y)^c = x^c \cdot y^c \) for all strings \( x \) and \( y \).

Your proofs must be formal and self-contained, and they must invoke the formal definitions of length \( |w| \), concatenation \( x \cdot y \), and complement \( w^c \). Do not appeal to intuition!

3. Recursively define a set \( L \) of strings over the alphabet \( \{0,1\} \) as follows:

- The empty string \( \varepsilon \) is in \( L \).
- For all strings \( x \) and \( y \) in \( L \), the string \( 0x1y \) is also in \( L \).
- For all strings \( x \) and \( y \) in \( L \), the string \( 1x0y \) is also in \( L \).
- These are the only strings in \( L \).

Let \( #(a,w) \) denote the number of times symbol \( a \) appears in string \( w \); for example,

\[
#(0,01000110111001) = #(1,01000110111001) = 7.
\]

(a) Prove that the string \( 01000110111001 \) is in \( L \).
(b) Prove by induction that every string in \( L \) has exactly the same number of 0s and 1s.
   (You may assume without proof that \( #(a,xy) = #(a,x) + #(a,y) \) for any symbol \( a \) and any strings \( x \) and \( y \).)
(c) Prove by induction that \( L \) contains every string with the same number of 0s and 1s.
Each homework assignment will include at least one solved problem, similar to the problems assigned in that homework, together with the grading rubric we would apply if this problem appeared on a homework or exam. These model solutions illustrate our recommendations for structure, presentation, and level of detail in your homework solutions. Of course, the actual content of your solutions won’t match the model solutions, because your problems are different!

Solved Problems

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

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

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

In parts (b) and (c), you may assume without proof that $(x \cdot y)^R = y^R \cdot x^R$ and $(x^R)^R = x$ for all strings $x$ and $y$.

Solution:

(a) A string $w \in \Sigma^*$ is a palindrome if and only if either
• $w = \epsilon$, 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 = $\frac{1}{2}$ for each base case + 1 for the recursive case. No credit for the rest of the problem unless this is correct.

(b) 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 three cases in the definition):
• If $w = \epsilon$, then $w^R = \epsilon$ by definition, so $w = w^R$.
• If $w = a$ for some symbol $a \in \Sigma$, then $w^R = a$ by definition, so $w = w^R$.
• Suppose $w = axa$ for some symbol $a \in \Sigma$ and some palindrome $x \in P$. Then

$$w^R = (axa)^R$$
$$= (x \cdot a)^R \cdot a$$
$$= a^R \cdot x^R \cdot a$$
$$= a \cdot x^R \cdot a$$
$$= a \cdot x \cdot a$$
$$= w$$

by definition of reversal
You said we could assume this.
by definition of reversal
by the inductive hypothesis
by assumption
In all three cases, we conclude that \( w = w^R \).

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

(c) 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 \( w = \epsilon \), 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 \( w^R = (ax)^R = x^Ra \).
  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 \( w = (w^R)^R = (bya)^R = ay^Rb \).
  [At this point, we need to prove that \( a = b \) and that \( y \) is a palindrome.]

Our assumption that \( w = w^R \) implies that \( bya = ay^Rb \).
The recursive definition of string equality immediately implies \( a = b \).
Because \( a = b \), we have \( w = ay^R \) and \( w^R = ay a \).
The recursive definition of string equality implies \( y^Ra = ya \).
It immediately follows that \( (y^R)^R = ay \) and \( (y^R)^R = ay^R \).
Known properties of reversal imply \( (y^R)^R = ay^R \).
It follows that \( ay^R = ay \), and therefore \( y = y^R \).
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).
• No penalty for jumping from \( ay a = ay^R a \) directly to \( y = y^R \).
Rubric (induction): For problems worth 10 points:

+ 1 for explicitly considering an arbitrary object
+ 2 for a valid strong induction hypothesis
  - Deadly Sin! Automatic zero for stating a weak induction hypothesis, unless the rest of the proof is perfect.
+ 2 for explicit exhaustive case analysis
  - No credit here if the case analysis omits an infinite number of objects. (For example: all odd-length palindromes.)
  - −1 if the case analysis omits a finite number of objects. (For example: the empty string.)
  - −1 for making the reader infer the case conditions. Spell them out!
  - No penalty if cases overlap (for example:
+ 1 for cases that do not invoke the inductive hypothesis (“base cases”)
  - No credit here if one or more “base cases” are missing.
+ 2 for correctly applying the stated inductive hypothesis
  - No credit here for applying a different inductive hypothesis, even if that different inductive hypothesis would be valid.
+ 2 for other details in cases that invoke the inductive hypothesis (“inductive cases”)
  - No credit here if one or more “inductive cases” are missing.
Starting with this homework, groups of up to three people can submit joint solutions. Each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the Gradescope names and email addresses of each group member. In addition, whoever submits the homework must tell Gradescope who their other group members are.

1. For each of the following languages over the alphabet \{0, 1\}, give a regular expression that describes that language, and briefly argue why your expression is correct.

   (a) All strings that end with the suffix 01010101.
   (b) All strings except 111.
   (c) All strings that contain the substring 010.
   (d) All strings that contain the subsequence 010.
   (e) All strings that do not contain the substring 010.
   (f) All strings that do not contain the subsequence 010.

2. This problem considers two special classes of regular expressions.

   • A regular expression \(R\) is \textbf{plus-free} if and only if it never uses the + operator.
   • A regular expression \(R\) is \textbf{top-plus} if and only if either
     - \(R\) is plus-free, or
     - \(R = S + T\), where \(S\) and \(T\) are top-plus.

   For example, \(1((0^*10)^*1)^*0\) is plus-free and (therefore) top-plus; \(01^*0 + 10^*1 + \epsilon\) is top-plus but not plus-free, and \(0(0 + 1)^*(1 + \epsilon)\) is neither top-plus nor plus-free.

   Recall that two regular expressions \(R\) and \(S\) are \textbf{equivalent} if they describe exactly the same language: \(L(R) = L(S)\).

   (a) Prove that for any top-plus regular expressions \(R\) and \(S\), there is a top-plus regular expression that is equivalent to \(RS\). [Hint: Use the fact that \((A + B)(C + D)\) and \(AC + AD + BC + BD\) are equivalent, for all regular expressions \(A, B, C,\) and \(D\).]

   (b) Prove that for any top-plus regular expression \(R\), there is a \textbf{plus-free} regular expression \(S\) such that \(R^*\) and \(S^*\) are equivalent. [Hint: Use the fact that \((A + B)^*\) is equivalent to \((A^*B^*)^*\), for all regular expressions \(A\) and \(B\).]

   (c) Prove that for any regular expression, there is an equivalent top-plus regular expression.
3. Let \( L \) be the set of all strings in \( \{0, 1\}^* \) that contain exactly two occurrences of the substring \( 001 \).

(a) Describe a DFA that over the alphabet \( \Sigma = \{0, 1\} \) that accepts the language \( L \). Argue that your machine accepts every string in \( L \) and nothing else, by explaining what each state in your DFA means.

You may either draw the DFA or describe it formally, but the states \( Q \), the start state \( s \), the accepting states \( A \), and the transition function \( \delta \) must be clearly specified.

(b) Give a regular expression for \( L \), and briefly argue that why expression is correct.
Solved problem

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

- Line comments: Strings of the form $//\cdots\dagger$.
- Block comments: Strings of the form $/\ast\cdots\ast/$.

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

- /****
- $/\ast$/
- $//\ast\ast\dagger$
- $/\ast\ast//\ast\ast\ast\ast\dagger$
- $/\ast\ast//\ast\ast\dagger\ast\ast$

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

- $/*$
- $//\ast\ast\dagger$
- $/\ast\ast//\ast\ast\dagger\ast\ast$
- $/\ast\ast//\ast\ast\ast\ast\dagger$
- $/\ast\ast//\ast\ast\dagger\ast\ast$

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

(b) Describe a DFA that accepts the set of all strings composed entirely of blanks ($\ast\ast$), newlines ($\dagger$), and C comments.

**You must explain in English how your DFAs work.** Drawings or formal descriptions without English explanations will receive no credit, even if they are correct.

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

- The opening $/\ast$ or $//$ of a comment must not be inside a string literal (“\cdots”) or a (multi-)character literal (’\cdots’).
- 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 (”\cdots”’) 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 ”\ast\ast\\ast\ast\ast\ast\ast\ast\ast\ast” is a valid string literal (representing the 5-character string $/\ast\ast\ast\ast\ast\ast\ast\ast$/, 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.
Solution:

(a) 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 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 read a complete block comment.

(b) 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.
Rubric: 10 points = 5 for each part, using the standard DFA design rubric (scaled)

Rubric (DFA design): For problems worth 10 points:

- 2 points for an unambiguous description of a DFA, including the states set \( Q \), the start state \( s \), the accepting states \( A \), and the transition function \( \delta \).
  - For drawings: Use an arrow from nowhere to indicate \( s \), and doubled circles to indicate accepting states \( A \). If \( A = \emptyset \), say so explicitly. If your drawing omits a reject state, say so explicitly. Draw neatly! If we can’t read your solution, we can’t give you credit for it.
  - For text descriptions: You can describe the transition function either using a 2d array, using mathematical notation, or using an algorithm.
  - For product constructions: You must give a complete description of the states and transition functions of the DFAs you are combining (as either drawings or text), together with the accepting states of the product DFA.

- Homework only: 4 points for briefly and correctly explaining the purpose of each state in English. This is how you justify that your DFA is correct.
  - For product constructions, explaining the states in the factor DFAs is enough.
  - Deadly Sin: (“Declare your variables.”) No credit for the problem if the English description is missing, even if the DFA is correct.

- 4 points for correctness. (8 points on exams, with all penalties doubled)
  - —1 for a single mistake: a single misdirected transition, a single missing or extra accept state, rejecting exactly one string that should be accepted, or accepting exactly one string that should be accepted.
  - —2 for incorrectly accepting/rejecting more than one but a finite number of strings.
  - —4 for incorrectly accepting/rejecting an infinite number of strings.

- DFA drawings with too many states may be penalized. DFA drawings with significantly too many states may get no credit at all.

- Half credit for describing an NFA when the problem asks for a DFA.
1. 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 $\Sigma$ called the input alphabet
- A finite set $\Gamma$ called the output alphabet
- A finite set $Q$ whose elements are called states
- A start state $s \in Q$
- A transition function $\delta : Q \times \Sigma \rightarrow Q$
- An output function $\omega : Q \rightarrow \Gamma$

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 $\delta$, and then outputs the symbol $\omega(q)$, where $q$ is the new state. Formally, we recursively define a transducer function $\omega^* : Q \times \Sigma^* \rightarrow \Gamma^*$ 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** $L^*(M)$ of a Moore machine $M$ is the set of all strings that the machine can output:

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

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

(b) Let $M$ be an arbitrary Moore machine whose input alphabet $\Sigma$ and output alphabet $\Gamma$ are identical. Prove that the language

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

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

*Hint: These problems are easier than they look!*
2. Prove that the following languages are not regular.

(a) \( \{ w \in (0 + 1)^* \mid |\#(0, w) - \#(1, w)| < 5 \} \)

(b) Strings in \((0 + 1)^*\) in which the substrings 00 and 11 appear the same number of times.

(c) \( \{ 0^m 1^m \mid n/m \text{ is an integer} \} \)

3. Let \( L \) be an arbitrary regular language.

(a) Prove that the language \( \text{palin}(L) := \{ w \mid ww^R \in L \} \) is also regular.

(b) Prove that the language \( \text{drome}(L) := \{ w \mid w^Rw \in L \} \) is also regular.
Solved problem

4. Let $L$ be an arbitrary regular language. Prove that the language $\text{half}(L) := \{w \mid ww \in L\}$ 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 $\epsilon$-transitions that accepts $\text{half}(L)$, as follows:

$$Q' = (Q \times Q \times Q) \cup \{s'\}$$
$$s' \text{ is an explicit state in } Q'$$
$$A' = \{(h, h, q) \mid h \in Q \text{ and } q \in A\}$$
$$\delta'(s', \epsilon) = \{(s, h, h) \mid h \in Q\}$$
$$\delta'( (p, h, q), a ) = \{(\delta(p, a), h, \delta(q, a))\}$$

$M'$ 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.

**Rubric:** 5 points =
- + 1 for a formal, complete, and unambiguous description of a DFA or NFA
  - No points for the rest of the problem if this is missing.
- + 3 for a correct NFA
  - -1 for a single mistake in the description (for example a typo)
- + 1 for a brief English justification. We explicitly do not want a formal proof of correctness, but we do want one or two sentences explaining how the NFA works.
Groups of up to three people can submit joint solutions. Each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the Gradescope names and email addresses of each group member. In addition, whoever submits the homework must tell Gradescope who their other group members are.

1. For each of the following regular expressions, describe or draw two finite-state machines:
   - An NFA that accepts the same language, obtained using Thompson’s recursive algorithm
   - An equivalent DFA, obtained using the incremental subset construction. For each state in your DFA, identify the corresponding subset of states in your NFA. Your DFA should have no unreachable states.
   
   (a) $(00 + 11)^*(0 + 1 + \varepsilon)$
   (b) $1^* + (01)^* + (001)^*$

2. Give context-free grammars for the following languages, and clearly explain how they work and the role of each nonterminal. Grammars can be very difficult to understand; if the grader does not understand how your construction is intended to generate the language, then you will receive no credit.

   (a) In any string, a block (also called a run) is a maximal non-empty substring of identical symbols. For example, the string $011000011001$ has six blocks: three blocks of 0s of lengths 1, 4, and 2, and three blocks of 1s of lengths 3, 2, and 1.
      
      Let $L$ be the set of all strings in $\{0, 1\}$ that contain two blocks of 0s of equal length. For example, $L$ contains the strings $01101111$ and $0100101110010$ but does not contain the strings $000110011011$ and $00000001111$.

   (b) $L = \{w \in \{0, 1\}^* \mid w \text{ is not a palindrome}\}$

3. Let $L = \{0^i1^j2^k \mid k = i + j\}$.
   
   (a) Show that $L$ is context-free by describing a grammar for $L$.
   (b) Prove that your grammar $G$ is correct. As usual, you need to prove both $L \subseteq L(G)$ and $L(G) \subseteq L$. 
Solved problem

4. Let \( L \) be the set of all strings over \( \{0, 1\}^* \) with exactly twice as many 0s as 1s.

(a) Describe a CFG for the language \( L \).

[Hint: For any string \( u \) define \( \Delta(u) = \#(0, u) - 2\#(1, u) \). Introduce intermediate variables that derive strings with \( \Delta(u) = 1 \) and \( \Delta(u) = -1 \) and use them to define a non-terminal that generates \( L \).]

**Solution:** \( S \rightarrow \varepsilon \mid SS \mid 00S1 \mid 0S1S0 \mid 1S00 \) \( \square \)

(b) Prove that your grammar \( G \) is correct. As usual, you need to prove both \( L \subseteq L(G) \) and \( L(G) \subseteq L \).

[Hint: Let \( u_{\leq i} \) denote the prefix of \( u \) of length \( i \). If \( \Delta(u) = 1 \), what can you say about the smallest \( i \) for which \( \Delta(u_{\leq i}) = 1 \)? How does \( u \) split up at that position? If \( \Delta(u) = -1 \), what can you say about the smallest \( i \) such that \( \Delta(u_{\leq i}) = -1 \)?]

**Solution:** (Hopefully you recognized this as a more advanced version of HW problem 3.) We separately prove \( L \subseteq L(G) \) and \( L(G) \subseteq L \) as follows:

**Claim 1.** \( L(G) \subseteq L \), that is, every string in \( L(G) \) has exactly twice as many 0s as 1s.

**Proof:** As suggested by the hint, for any string \( u \), let \( \Delta(u) = \#(0, u) - 2\#(1, u) \). We need to prove that \( \Delta(w) = 0 \) for every string \( w \in L(G) \).

Let \( w \) be an arbitrary string in \( L(G) \), and consider an arbitrary derivation of \( w \) of length \( k \). Assume that \( \Delta(x) = 0 \) for every string \( x \in L(G) \) that can be derived with fewer than \( k \) productions.\( \square \) There are five cases to consider, depending on the first production in the derivation of \( w \).

- If \( w = \varepsilon \), then \( \#(0, w) = \#(1, w) = 0 \) by definition, so \( \Delta(w) = 0 \).
- Suppose the derivation begins \( S \rightarrow SS \rightarrow^* w \). Then \( w = xy \) for some strings \( x, y \in L(G) \), each of which can be derived with fewer than \( k \) productions. The inductive hypothesis implies \( \Delta(x) = \Delta(y) = 0 \). It immediately follows that \( \Delta(w) = 0 \).\( \square \)
- Suppose the derivation begins \( S \rightarrow 00S1 \rightarrow^* w \). Then \( w = 00x1 \) for some string \( x \in L(G) \). The inductive hypothesis implies \( \Delta(x) = 0 \). It immediately follows that \( \Delta(w) = 0 \).
- Suppose the derivation begins \( S \rightarrow 1S00 \rightarrow^* w \). Then \( w = 1x00 \) for some string \( x \in L(G) \). The inductive hypothesis implies \( \Delta(x) = 0 \). It immediately follows that \( \Delta(w) = 0 \).
- Suppose the derivation begins \( S \rightarrow 0S1S1 \rightarrow^* w \). Then \( w = 0x1y0 \) for some strings \( x, y \in L(G) \). The inductive hypothesis implies \( \Delta(x) = \Delta(y) = 0 \). It immediately follows that \( \Delta(w) = 0 \).

In all cases, we conclude that \( \Delta(w) = 0 \), as required. \( \square \)

\( \diamond \)Alternatively: Consider the shortest derivation of \( w \), and assume \( \Delta(x) = 0 \) for every string \( x \in L(G) \) such that \( |x| < |w| \).

\( \diamond \)Alternatively: Suppose the shortest derivation of \( w \) begins \( S \rightarrow SS \rightarrow^* w \). Then \( w = xy \) for some strings \( x, y \in L(G) \). Neither \( x \) or \( y \) can be empty, because otherwise we could shorten the derivation of \( w \). Thus, \( x \) and \( y \) are both shorter than \( w \), so the induction hypothesis implies. . . We need some way to deal with the decompositions \( w = \varepsilon \star w \) and \( w = w \star \varepsilon \), which are both consistent with the production \( S \rightarrow SS \), without falling into an infinite loop.
Claim 2. \( L \subseteq L(G) \); that is, \( G \) generates every binary string with exactly twice as many \( 0 \)s as \( 1 \)s.

Proof: As suggested by the hint, for any string \( u \), let \( \Delta(u) = \#(0, u) - 2 \#(1, u) \). For any string \( u \) and any integer \( 0 \leq i \leq |u| \), let \( u_i \) denote the \( i \)th symbol in \( u \), and let \( u_{\leq i} \) denote the prefix of \( u \) of length \( i \).

Let \( w \) be an arbitrary binary string with twice as many \( 0 \)s as \( 1 \)s. Assume that \( G \) generates every binary string \( x \) that is shorter than \( w \) and has twice as many \( 0 \)s as \( 1 \)s. There are two cases to consider:

- If \( w = \varepsilon \), then \( \varepsilon \in L(G) \) because of the production \( S \to \varepsilon \).
- Suppose \( w \) is non-empty. To simplify notation, let \( \Delta_i = \Delta(w_{\leq i}) \) for every index \( i \), and observe that \( \Delta_0 = \Delta_{|w|} = 0 \). There are several subcases to consider:
  
  - Suppose \( \Delta_i = 0 \) for some index \( 0 < i < |w| \). Then \( w \) must begin with \( \varepsilon \varepsilon \), since otherwise \( \Delta_1 = -2 \) or \( \Delta_2 = -1 \), and the last symbol in \( w \) must be \( 1 \), since otherwise \( \Delta_{|w|} = -1 \). Thus, we can write \( w = \varepsilon \varepsilon x 1 \) for some binary string \( x \). We easily observe that \( \Delta(x) = 0 \), so the induction hypothesis implies \( x \in L(G) \), and thus the production rule \( S \to \varepsilon \varepsilon S 1 \) implies \( w \in L(G) \).
  
  - Suppose \( \Delta_i > 0 \) for all \( 0 < i < |w| \). Then \( w \) must begin with \( 1 \varepsilon \varepsilon \), since otherwise \( \Delta_1 = -2 \) or \( \Delta_2 = -1 \), and the last symbol in \( w \) must be \( 0 \), since otherwise \( \Delta_{|w|} = -1 \). Thus, we can write \( w = 1 \varepsilon \varepsilon x 0 \) for some binary string \( x \) with \( \Delta(x) = 0 \). The induction hypothesis implies \( x \in L(G) \), and thus the production rule \( S \to 1 \varepsilon \varepsilon S 0 \) implies \( w \in L(G) \).
  
  - Finally, suppose none of the previous cases applies: \( \Delta_i < 0 \) and \( \Delta_j > 0 \) for some indices \( i \) and \( j \), but \( \Delta_i \neq 0 \) for all \( 0 < i < |w| \).

Let \( i \) be the smallest index such that \( \Delta_i < 0 \). Because \( \Delta_j \) either increases by 1 or decreases by 2 when we increment \( j \), for all indices \( 0 < j < |w| \), we must have \( \Delta_j > 0 \) if \( j < i \) and \( \Delta_j < 0 \) if \( j \geq i \).

In other words, there is a unique index \( i \) such that \( \Delta_{i-1} > 0 \) and \( \Delta_i < 0 \). In particular, we have \( \Delta_1 > 0 \) and \( \Delta_{|w|-1} < 0 \). Thus, we can write \( w = 0 \varepsilon 1 y 0 \) for some binary strings \( x \) and \( y \), where \( |0x1| = i \).

We easily observe that \( \Delta(x) = \Delta(y) = 0 \), so the inductive hypothesis implies \( x, y \in L(G) \), and thus the production rule \( S \to 0 S 1 S 0 \) implies \( w \in L(G) \).

In all cases, we conclude that \( G \) generates \( w \).

Together, Claim 1 and Claim 2 imply \( L = L(G) \).

Rubric: 10 points:

- part (a) = 4 points. As usual, this is not the only correct grammar.
- part (b) = 6 points = 3 points for \( \subseteq + 3 \) points for \( \supseteq \), each using the standard induction template (scaled).
1. Consider the following restricted variant of the Tower of Hanoi puzzle. The pegs are numbered 0, 1, and 2, and your task is to move a stack of \( n \) disks from peg 1 to peg 2. However, you are forbidden to move any disk directly between peg 1 and peg 2; every move must involve peg 0.

Describe an algorithm to solve this version of the puzzle in as few moves as possible. Exactly how many moves does your algorithm make?

2. Consider the following cruel and unusual sorting algorithm.

\[
\text{Cruel}(A[1..n]):
\begin{align*}
\text{if } n > 1 & \\
\text{Cruel}(A[1..n/2]) & \\
\text{Cruel}(A[n/2 + 1..n]) & \\
\text{Unusual}(A[1..n]) &
\end{align*}
\]

\[
\text{Unusual}(A[1..n]):
\begin{align*}
\text{if } n = 2 & \\
\text{else} & \\
\text{for } i \leftarrow 1 \text{ to } n/4 & \text{ (swap 2nd and 3rd quarters)} \\
\text{swap } A[i + n/4] \leftrightarrow A[i + n/2] & \\
\text{Unusual}(A[1..n/2]) & \text{ (recurse on left half)} \\
\text{Unusual}(A[n/2 + 1..n]) & \text{ (recurse on right half)} \\
\text{Unusual}(A[n/4 + 1..3n/4]) & \text{ (recurse on middle half)}
\end{align*}
\]

Notice that the comparisons performed by the algorithm do not depend at all on the values in the input array; such a sorting algorithm is called oblivious. Assume for this problem that the input size \( n \) is always a power of 2.

(a) Prove by induction that Cruel correctly sorts any input array. [Hint: Consider an array that contains \( n/4 \) 1s, \( n/4 \) 2s, \( n/4 \) 3s, and \( n/4 \) 4s. Why is this special case enough? What does Unusual actually do?]

(b) Prove that Cruel would not correctly sort if we removed the for-loop from Unusual.

(c) Prove that Cruel would not correctly sort if we swapped the last two lines of Unusual.

(d) What is the running time of Unusual? Justify your answer.

(e) What is the running time of Cruel? Justify your answer.
3. You are a visitor at a political convention (or perhaps a faculty meeting) with \( n \) delegates. Each delegate is a member of exactly one political party. It is impossible to tell which political party any delegate belongs to. In particular, you will be summarily ejected from the convention if you ask. However, you can determine whether any pair of delegates belong to the same party or not simply by introducing them to each other. Members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.

(a) Suppose more than half of the delegates belong to the same political party. Describe and analyze an efficient algorithm that identifies every member of this majority party.

(b) Now suppose precisely \( p \) political parties are present and one party has a plurality: more delegates belong to that party than to any other party. Please present a procedure to pick out the people from the plurality party as parsimoniously as possible. Do not assume that \( p = O(1) \).

\[\text{Describe and analyze an efficient algorithm that identifies every member of the plurality party.}\]
Solved Problem

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:

- Recursively count inversions in (and sort) $Q[1..\lfloor n/2 \rfloor]$.
- Recursively count inversions in (and sort) $Q[\lceil n/2 \rceil + 1..n]$.
- Count inversions $Q[i] > Q[j]$ where $i \leq \lfloor n/2 \rfloor$ and $j > \lceil n/2 \rceil$ as follows:
  - Color the elements in the Left half $Q[1..\lfloor n/2 \rfloor]$ blue.
  - Color the elements in the Right half $Q[\lceil n/2 \rceil + 1..n]$ red.
  - Merge $Q[1..\lfloor n/2 \rfloor]$ and $Q[\lceil n/2 \rceil + 1..n]$, maintaining their colors.
  - For each blue element $Q[i]$, count the number of smaller red elements $Q[j]$.

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

```python
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
```

3
In fact, we can execute the third merge-and-count step directly by modifying the \textsc{Merge} algorithm, without any need for “colors”. Here changes to the standard \textsc{Merge} algorithm are indicated in red.

\begin{verbatim}
\textsc{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
\end{verbatim}

We can further optimize this algorithm by observing that \textit{count} is always equal to \( j - m - 1 \). (Proof: Initially, \( j = m + 1 \) and \textit{count} = 0, and we always increment \( j \) and \textit{count} together.)

\begin{verbatim}
\textsc{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
\end{verbatim}

The modified \textsc{Merge} algorithm still runs in \( O(n) \) time, so the running time of the resulting modified mergesort still obeys the recurrence \( T(n) = 2T(n/2) + O(n) \). We conclude that the overall running time is \( O(n \log n) \), as required.

\textbf{Rubric:} 10 points = 2 for base case + 3 for divide (split and recurse) + 3 for conquer (merge and count) + 2 for time analysis. Max 3 points for a correct \( O(n^2) \)-time algorithm. 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.
1. For each of the following problems, the input consists of two arrays $X[1..k]$ and $Y[1..n]$ where $k \leq n$.

   (a) Describe and analyze an algorithm to determine whether $X$ occurs as two disjoint subsequences of $Y$, where “disjoint” means the two subsequences have no indices in common. For example, the string PPAP appears as two disjoint subsequences in the string PENPINEAPPLEAPPLEPEN, but the string PEEPLE does not.

   (b) Describe and analyze an algorithm to compute the number of occurrences of $X$ as a subsequence of $Y$. For example, the string PPAP appears exactly 23 times as a subsequence of the string PENPINEAPPLEAPPLEPEN. If all characters in $X$ and $Y$ are equal, your algorithm should return $\binom{n}{k}$. For purposes of analysis, assume that each arithmetic operation takes $O(1)$ time.

2. You are driving a bus along a long straight highway, full of rowdy, hyper, thirsty students and an endless supply of soda. Each minute that each student is on your bus, that student drinks one ounce of soda. Your goal is to drive all students home, so that the total volume of soda consumed by the students is as small as possible.

   Your bus begins at an exit (probably not at either end) with all students on board and moves at a constant speed of 37.4 miles per hour. Each student needs to be dropped off at a highway exit. You may reverse directions as often as you like; for example, you are allowed to drive forward to the next exit, let some students off, then turn around and drive back to the previous exit, drop more students off, then turn around again and drive to further exits. (Assume that at each exit, you can stop the bus, drop off students, and if necessary turn around, all instantaneously.)

   Describe an efficient algorithm to take the students home so that they drink as little soda as possible. Your algorithm will be given the following input:

   • A sorted array $L[1..n]$, where $L[i]$ is the Location of the $i$th exit, measured in miles from the first exit; in particular, $L[1] = 0$.
   • An array $N[1..n]$, where $N[i]$ is the Number of students you need to drop off at the $i$th exit.
   • An integer start equal to the index of the starting exit.

   Your algorithm should return the total volume of soda consumed by the students when you drive the optimal route.\[1\] ¹Non-US students are welcome to assume kilometers and liters instead of miles and ounces. Late 18th-century French students are welcome to use decimal minutes.
3. *Vankin’s Mile* is an American solitaire game played on an $n \times n$ square grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his score. The object of the game is to score as many points as possible.

For example, given the grid below, the player can score $8 - 6 + 7 - 3 + 4 = 10$ points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. (This is not the best possible score for these values.)

<table>
<thead>
<tr>
<th>7</th>
<th>10</th>
<th>-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>7</td>
<td>-8</td>
</tr>
<tr>
<td>-4</td>
<td>-9</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>-2</td>
<td>-6</td>
</tr>
<tr>
<td>-7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>-6</td>
</tr>
</tbody>
</table>

(a) Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin’s Mile, given the $n \times n$ array of values as input.

(b) In the Canadian version of this game, appropriately called *Vankin’s Kilometer*, the player can move the token either one square down, one square right, or one square left in each turn. However, to prevent infinite scores, the token cannot land on the same square more than once. Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin’s Kilometer, given the $n \times n$ array of values as input.\[\square\]

\[\footnote{If we also allowed upward movement, the resulting game (Vankin’s Fathom?) would be NP-hard.}\]
Solved Problem

4. 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 $\text{BANANAANANAS}$ is a shuffle of the strings $\text{BANANA}$ and $\text{ANANAS}$ in several different ways.

$\text{BANANAANANAS} \quad \text{BANANAANANAS} \quad \text{BANANAANANAS}$

Similarly, the strings $\text{PRODGYRNAMMIINCG}$ and $\text{DYPRONGARMAMMICING}$ are both shuffles of $\text{DYNAMIC}$ and $\text{PROGRAMMING}$:

$\text{PRODGYRNAMMIINCG} \quad \text{DYPRONGARMAMMICING}$

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 $\text{Shuf}(i,j)$, which is $\text{TRUE}$ if and only if the prefix $C[1..i+j]$ is a shuffle of the prefixes $A[1..i]$ and $B[1..j]$. This function satisfies the following recurrence:

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

We need to compute $\text{Shuf}(m,n)$.

We can memoize all function values into a two-dimensional array $\text{Shuf}[0..m][0..n]$. Each array entry $\text{Shuf}[i,j]$ depends only on the entries immediately below and immediately to the right: $\text{Shuf}[i-1, j]$ and $\text{Shuf}[i, j-1]$. Thus, we can fill the array in standard row-major order. The original recurrence gives us the following pseudocode:

```
SHUFFLE?(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,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]
```

The algorithm runs in $O(mn)$ time. ■
Rubric: Max 10 points: Standard dynamic programming rubric. No proofs required. Max 7 points for a slower polynomial-time algorithm; scale partial credit accordingly.

Standard dynamic programming rubric.  For problems worth 10 points:

• 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  + 1 point for a clear English description of the function you are trying to evaluate. (Otherwise, we don’t even know what you’re trying to do.) Automatic zero if the English description is missing.
  + 1 point for stating how to call your function to get the final answer.
  + 1 point for base case(s). —½ for one minor bug, like a typo or an off-by-one error.
  + 3 points for recursive case(s). —1 for each minor bug, like a typo or an off-by-one error. No credit for the rest of the problem if the recursive case(s) are incorrect.

• 4 points for details of the dynamic programming algorithm
  + 1 point for describing the memoization data structure
  + 2 points for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested loops, be sure to specify the nesting order.
  + 1 point for time analysis

• It is not necessary to state a space bound.

• For problems that ask for an algorithm that computes an optimal structure—such as a subset, partition, subsequence, or tree—an algorithm that computes only the value or cost of the optimal structure is sufficient for full credit, unless the problem says otherwise.

• Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, but iterative psuedocode is not required for full credit. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. (But you still need to describe the underlying recursive function in English.)

• Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we have included them, significantly more students turned in algorithms that meet the target time bound but didn’t work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).
1. Every year, as part of its annual meeting, the Antarctic Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to \( n \). During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.

The end of a typical Antarctican SLUG race. Snails 6 and 8 never find mates.


For every pair of snails, the Antarctic SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array \( M[1..n, 1..n] \) posted on the wall behind the Round Table, where \( M[i, j] = M[j, i] \) is the reward to be paid if snails \( i \) and \( j \) meet. Rewards may be positive, negative, or zero.

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array \( M \) as input.
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. 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, Elmo has become a significantly 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. [Hint: What is a perfect opponent?]
3. One day, Alex got tired of climbing in a gym and decided to take a very large group of climber friends outside to climb. The climbing area where they went, had a huge wide boulder, not very tall, with various marked hand and foot holds. Alex quickly determined an “allowed” set of moves that her group of friends can perform to get from one hold to another.

The overall system of holds can be described by a rooted tree $T$ with $n$ vertices, where each vertex corresponds to a hold and each edge corresponds to an allowed move between holds. The climbing paths converge as they go up the boulder, leading to a unique hold at the summit, represented by the root of $T$.  

Alex and her friends (who are all excellent climbers) decided to play a game, where as many climbers as possible are simultaneously on the boulder and each climber needs to perform a sequence of exactly $k$ moves. Each climber can choose an arbitrary hold to start from, and all moves must move away from the ground. Thus, each climber traces out a path of $k$ edges in the tree $T$, all directed toward the root. However, no two climbers are allowed to touch the same hold; the paths followed by different climbers cannot intersect at all.

Describe and analyze an efficient algorithm to compute the maximum number of climbers that can play this game. More formally, you are given a rooted tree $T$ and an integer $k$, and you want to find the largest possible number of disjoint paths in $T$, where each path has length $k$. For full credit, do not assume that $T$ is a binary tree. For example, given the tree $T$ below and $k = 3$ as input, your algorithm should return the integer 8.

---

Why do computer science professors think trees have their roots at the top?
A: Because they've never been outside!
Solved Problems

4. A string $w$ of parentheses ( and ) and brackets [ and ] is balanced if it is generated by
   the following context-free grammar:

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

   For example, the string $w = (([]))(()())(())()$ is balanced, because $w = xy$, where
   $$x = ( [ ] ) [ ] ) \) and \( y = [ ( ) ] ().$$

   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 $i$.


   For all indices $i$ and $j$, let $LBS(i, j)$ denote the length of the longest balanced subsequence of the substring $A[i..j]$. We need to compute $LBS(1, n)$. This function obeys the following recurrence:

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

   We can memoize this function into a two-dimensional array $LBS[1..n, 1..n]$. Since every entry $LBS[i, j]$ depends only on entries in later rows or earlier columns (or both), we can evaluate this array row-by-row from bottom up in the outer loop, scanning each row from left to right in the inner loop. The resulting algorithm runs in $O(n^3)$ time.

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

   **Rubric:** 10 points, standard dynamic programming rubric
5. 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 \( T \).

- \( \text{MaxFunYes}(v) \) is the maximum total “fun” of a legal party among the descendants of \( v \), where \( v \) is definitely invited.
- \( \text{MaxFunNo}(v) \) is the maximum total “fun” of a legal party among the descendants of \( v \), where \( v \) is definitely not invited.

We need to compute \( \text{MaxFunYes}(\text{root}) \). These two functions obey the following mutual recurrences:

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

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

(These recurrences do not require separate base cases, because \( \sum \emptyset = 0 \).) 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 values at its children, so we can compute all \( 2n \) values using a postorder traversal of \( T \).

\[
\text{ComputeMaxFun}(v):\]

\[
\begin{align*}
&v.yes \leftarrow v.fun \\
&v.no \leftarrow 0 \\
&\text{for all children } w \text{ of } v \\
&\quad \text{ComputeMaxFun}(w) \\
&\quad v.yes \leftarrow v.yes + w.no \\
&\quad v.no \leftarrow v.no + \max\{w.yes, w.no\}
\end{align*}
\]

(Yes, this is still dynamic programming; we’re only traversing the tree recursively because that’s the most natural way to traverse trees!\(^\dagger\) The algorithm spends \( O(1) \) time at each node, and therefore runs in \( O(n) \) time altogether.

\[^\dagger\text{A naïve recursive implementation would run in } O(\phi^n) \text{ time in the worst case, where } \phi = (1 + \sqrt{5})/2 \approx 1.618 \text{ is the golden ratio. The worst-case tree is a path—every non-leaf node has exactly one child.}\]
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

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

The function $MaxFun$ obeys the following recurrence:

$$MaxFun(v) = \max \left\{ v.\text{fun} + \sum_{\text{grandchildren } x \text{ of } v} MaxFun(x), \sum_{\text{children } w \text{ of } v} MaxFun(w) \right\}$$

(This recurrence does not require a separate base case, because $\sum \emptyset = 0$.) We can memoize this function by adding an additional field $v.\text{maxFun}$ 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 $T$.

---

**BestParty**($T$):

**ComputeMaxFun**($T.\text{root}$)

party $\leftarrow T.\text{root.fun}$

for all children $w$ of $T.\text{root}$

for all children $x$ of $w$

party $\leftarrow$ party + $x.\text{maxFun}$

return party

---

**ComputeMaxFun**($v$):

yes $\leftarrow v.\text{fun}$

no $\leftarrow 0$

for all children $w$ of $v$

**ComputeMaxFun**($w$)

no $\leftarrow$ no + $w.\text{maxFun}$

for all children $x$ of $w$

yes $\leftarrow$ yes + $x.\text{maxFun}$

$v.\text{maxFun}$ $\leftarrow \max\{$yes, no$\}$

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

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

---

**Rubric:** 10 points: standard dynamic programming rubric. These are not the only correct solutions.

---

³Like the previous solution, a direct recursive implementation would run in $O(\phi^n)$ time in the worst case, where $\phi = (1 + \sqrt{5})/2 \approx 1.618$ is the golden ratio.
If you use a greedy algorithm, you must prove that it is correct, or you will get zero points even if your algorithm is correct.

1. You’ve been hired to store a sequence of \( n \) books on shelves in a library. The order of the books is fixed by the cataloging system and cannot be changed; each shelf must store a contiguous interval of the given sequence of books. You are given two arrays \( H[1..n] \) and \( T[1..n] \), where \( H[i] \) and \( T[i] \) are respectively the height and thickness of the \( i \)th book in the sequence. All shelves in this library have the same length \( L \); the total thickness of all books on any single shelf cannot exceed \( L \).

   (a) Suppose all the books have the same height \( h \) (that is, \( H[i] = h \) for all \( i \)) and the shelves have height larger than \( h \), so each book fits on every shelf. Describe and analyze a greedy algorithm to store the books in as few shelves as possible. [Hint: The algorithm is obvious, but why is it correct?]

   (b) That was a nice warmup, but now here’s the real problem. In fact the books have different heights, but you can adjust the height of each shelf to match the tallest book on that shelf. (In particular, you can change the height of any empty shelf to zero.) Now your task is to store the books so that the sum of the heights of the shelves is as small as possible. Show that your greedy algorithm from part (a) does not always give the best solution to this problem.

   (c) Describe and analyze an algorithm to find the best assignment of books to shelves as described in part (b).

2. Consider a directed graph \( G \), where each edge is colored either red, white, or blue. A walk in \( G \) is called a French flag walk if its sequence of edge colors is red, white, blue, red, white, blue, and so on. More formally, a walk \( v_0 \to v_1 \to \cdots \to v_k \) is a French flag walk if, for every integer \( i \), the edge \( v_i \to v_{i+1} \) is red if \( i \text{ mod } 3 = 0 \), white if \( i \text{ mod } 3 = 1 \), and blue if \( i \text{ mod } 3 = 2 \).

   Describe an efficient algorithm to find all vertices in a given edge-colored directed graph \( G \) that can be reached from a given vertex \( v \) through a French flag walk.

\[\texttt{Recall that a walk in a directed graph } G \texttt{ is a sequence of vertices } v_0 \to v_1 \to \cdots \to v_k \texttt{, such that } v_{i-1} \to v_i \texttt{ is an edge in } G \texttt{ for every index } i \texttt{. A path is a walk in which no vertex appears more than once.}\]
3. **Racetrack** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade. The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

Each car has a *position* and a *velocity*, both with integer $x$- and $y$-coordinates. A subset of grid squares is marked as the *starting area*, and another subset is marked as the *finishing area*. The initial position of each car is chosen by the player somewhere in the starting area; the initial velocity of each car is always $(0, 0)$. At each step, the player optionally increments or decrements either or both coordinates of the car’s velocity; in other words, each component of the velocity can change by at most 1 in a single step. The car’s new position is then determined by adding the new velocity to the car’s previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race.

Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the “starting area” is the first column, and the “finishing area” is the last column.

Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting line to the finish line of a given racetrack. *[Hint: Build a graph. No, not that graph, a different one. What are the vertices? What are the edges? What problem is this?]*

<table>
<thead>
<tr>
<th>velocity</th>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0)</td>
<td>(1, 5)</td>
</tr>
<tr>
<td>(1, 0)</td>
<td>(2, 5)</td>
</tr>
<tr>
<td>(2, −1)</td>
<td>(4, 4)</td>
</tr>
<tr>
<td>(3, 0)</td>
<td>(7, 4)</td>
</tr>
<tr>
<td>(2, 1)</td>
<td>(9, 5)</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>(10, 7)</td>
</tr>
<tr>
<td>(0, 3)</td>
<td>(10, 10)</td>
</tr>
<tr>
<td>(−1, 4)</td>
<td>(9, 14)</td>
</tr>
<tr>
<td>(0, 3)</td>
<td>(9, 17)</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>(10, 19)</td>
</tr>
<tr>
<td>(2, 2)</td>
<td>(12, 21)</td>
</tr>
<tr>
<td>(2, 1)</td>
<td>(14, 22)</td>
</tr>
<tr>
<td>(2, 0)</td>
<td>(16, 22)</td>
</tr>
<tr>
<td>(1, −1)</td>
<td>(17, 21)</td>
</tr>
<tr>
<td>(2, −1)</td>
<td>(19, 20)</td>
</tr>
<tr>
<td>(3, 0)</td>
<td>(22, 20)</td>
</tr>
<tr>
<td>(3, 1)</td>
<td>(25, 21)</td>
</tr>
</tbody>
</table>

A 16-step Racetrack run, on a 25 × 25 track. This is not the shortest run on this track.

---

²The actual game is a bit more complicated than the version described here. See [http://harmmade.com/vectorracer/](http://harmmade.com/vectorracer/) for an excellent online version.

³However, it is not necessary for the line between the old position and the new position to lie entirely within the track. Sometimes Speed Racer has to push the A button.
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 (for 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 the following directed graph:

- $V = \{(a,b,c) \mid 0 \leq a \leq p$ and $0 \leq b \leq B$ and $0 \leq c \leq C\}$. 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.
- The graph has a directed edge $(a,b,c)\rightarrow(a',b',c')$ whenever it is possible to move from the first configuration to the second in one step. Specifically, there is 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
  - $\begin{cases} (0,a+b,c) & \text{if } a+b \leq B \\ (a+b-B,B,c) & \text{if } a+b \geq B \end{cases}$ — pouring from the first jar into the second
  - $\begin{cases} (0,b,a+c) & \text{if } a+c \leq C \\ (a+c-C,b,C) & \text{if } a+c \geq C \end{cases}$ — pouring from the first jar into the third
  - $\begin{cases} (a+b,0,c) & \text{if } a+b \leq A \\ (A,a+b-A,c) & \text{if } a+b \geq A \end{cases}$ — pouring from the second jar into the first
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 $O(V + E) = O(ABC)$ time.

We can make this algorithm faster by observing that every move either leaves at least one jar empty or leaves at least one jar 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 $O(AB + BC + AC)$ time.

**Rubric (for graph reduction problems):** 10 points:

- 2 for correct vertices
- 2 for correct edges
  - ½ for forgetting “directed”
- 2 for stating the correct problem (shortest paths)
  - “Breadth-first search” is not a problem; it’s an algorithm.
- 2 points for correctly applying the correct algorithm (breadth-first search)
  - 1 for using Dijkstra instead of BFS
- 2 points for time analysis in terms of the input parameters.
- Max 8 points for $O(ABC)$ time; scale partial credit
1. After a grueling algorithms midterm, 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 Champaign-Urbana. Champaign-Urbana is currently suffering from a plague of zombies, so even though the bus stops have fences that supposedly keep the zombies out, you’d still like to spend as little time waiting at bus stops as possible. Unfortunately, there isn’t a single bus that visits both your exam building and your home; you must transfer between buses at least once.

Describe and analyze an algorithm to determine a sequence of bus rides from Siebel to your home, that minimizes the total time you spend waiting at bus stops. You can assume that there are \( b \) different bus lines, and each bus stops \( n \) times per day. Assume that the buses run exactly on schedule, that you have an accurate watch, and that walking between bus stops is too dangerous to even contemplate.

2. Kris is a professional rock climber (friends with Alex and the rest of the climbing crew from HW6) who is competing in the U.S. climbing nationals. The competition requires Kris to use as many holds on the climbing wall as possible, using only transitions that have been explicitly allowed by the route-setter.

The climbing wall has \( n \) holds. Kris is given a list of \( m \) pairs \((x, y)\) of holds, each indicating that moving directly from hold \( x \) to hold \( y \) is allowed; however, moving directly from \( y \) to \( x \) is not allowed unless the list also includes the pair \((y, x)\). Kris needs to figure out a sequence of allowed transitions that uses as many holds as possible, since each new hold increases his score by one point. The rules allow Kris to choose the first and last hold in his climbing route. The rules also allow him to use each hold as many times as he likes; however, only the first use of each hold increases Kris’s score.

(a) Define the natural graph representing the input. Describe and analyze an algorithm to solve Kris’s climbing problem if you are guaranteed that the input graph is a dag.

(b) Describe and analyze an algorithm to solve Kris’s climbing problem with no restrictions on the input graph.

Both of your algorithms should output the maximum possible score that Kris can earn.
3. Many years later, in a land far, far away, after winning all the U.S. national competitions for 10 years in a row, Kris retired from competitive climbing and became a route setter for competitions. However, as the years passed, the rules changed. Climbers are now required to climb along the shortest sequence of legal moves from one specific node to another, where the distance between two holds is specified by the route setter. In addition to the usual set of $n$ holds and $m$ valid moves between them (as in the previous problem), climbers are now told their start hold $s$, their finish hold $t$, and the distance from $x$ to $y$ for every allowed move $(x, y)$.

Rather than make up this year’s new route completely from scratch, Kris decides to make one small change to last year’s input. The previous route setter suggested a list of $k$ new allowed moves and their distances. Kris needs to choose the single edge from this list of suggestions that decreases the distance from $s$ to $t$ as much as possible.

Describe and analyze an algorithm to solve Kris’s problem. Your input consists of the following information:

- A directed graph $G = (V, E)$.
- Two vertices $s, t \in V$.
- A set of $k$ new edges $E'$, such that $E \cap E' = \emptyset$.
- A length $\ell(e) \geq 0$ for every edge $e \in E \cup E'$.

Your algorithm should return the edge $e \in E'$ whose addition to the graph yields the smallest shortest path distance from $s$ to $t$.

For full credit, your algorithm should run in $O(m \log n + k)$ time, but as always, a slower correct algorithm is worth more than a faster incorrect algorithm.
Solved Problem

4. Although we typically speak of “the” shortest path between two nodes, a single graph could contain several minimum-length paths with the same endpoints.

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

[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 $u \rightarrow v$ tight if $dist(u) + w(u \rightarrow v) = dist(v)$. 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 $PathsToT(v)$ denote the number of paths in $H$ from $v$ to $t$; we need to compute $PathsToT(s)$. This function satisfies the following simple recurrence:

$$PathsToT(v) = \begin{cases} 
1 & \text{if } v = t \\
\sum_{v \rightarrow w} PathsToT(w) & \text{otherwise}
\end{cases}$$

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 $PathsToT(v) = \sum \emptyset = 0$.

We can memoize this function into the graph itself, storing each value $PathsToT(v)$ at the corresponding vertex $v$. Since each subproblem depends only on its successors in $H$, we can compute $PathsToT(v)$ 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 $O(E \log V)$ time. ■

Rubric: 10 points = 5 points for reduction to counting paths in a dag + 5 points for the path-counting algorithm (standard dynamic programming rubric)
1. Consider the following problem, called BoxDepth: Given a set of \( n \) axis-aligned rectangles in the plane, how big is the largest subset of these rectangles that contain a common point?

   (a) Describe a polynomial-time reduction from BoxDepth to MaxClique, and prove that your reduction is correct.

   (b) Describe and analyze a polynomial-time algorithm for BoxDepth. \([Hint: \text{Don’t try to optimize the running time; } O(n^3) \text{ is good enough.}]\)

   (c) Why don’t these two results imply that P=NP?

2. 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 prove that your reductions are correct.

3. Suppose you are given a graph \( G = (V,E) \) where \( V \) represents a collection of people and an edge between two people indicates that they are friends. You wish to partition \( V \) into at most \( k \) non-overlapping groups \( V_1, V_2, \ldots, V_k \) such that each group is very cohesive. One way to model cohesiveness is to insist that each pair of people in the same group should be friends; in other words, they should form a clique.

   Prove that the following problem is NP-hard: Given an undirected graph \( G \) and an integer \( k \), decide whether the vertices of \( G \) can be partitioned into \( k \) cliques.
Solved Problem

4. 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, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.

![A solvable puzzle and one of its many solutions. An unsolvable puzzle.](image)

Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether this puzzle can be solved.

Solution: We show that this puzzle is NP-hard by describing a reduction from \( 3\text{SAT} \).

Let \( \Phi \) be a \( 3\text{CNF} \) boolean formula with \( m \) variables and \( n \) clauses. We transform this formula into a puzzle configuration 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 \( i \)th clause of \( \Phi \), we place a blue stone at \((i, j)\).
- If the negated variable \( \overline{x_j} \) appears in the \( i \)th clause of \( \Phi \), we place a red stone at \((i, j)\).
- Otherwise, we leave cell \((i, j)\) blank.

**We claim that this puzzle has a solution if and only if \( \Phi \) is satisfiable.** This claim immediately implies that solving the puzzle is NP-hard. We prove our claim as follows:

\( \Rightarrow \) First, suppose \( \Phi \) 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 \( \text{FALSE} \) 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 \( \Phi \) must contain at least one \( \text{TRUE} \) literal, and thus each row still contains at least one stone. We conclude that the puzzle is satisfiable.

\( \Leftarrow \) On the other hand, suppose the puzzle 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 \textsc{true}. Each row still has at least one stone, so each clause of $\Phi$ contains at least one \textsc{true} literal, so this assignment makes $\Phi = \textsc{true}$. We conclude that $\Phi$ is satisfiable.

This reduction clearly requires only polynomial time. \hfill \blacksquare

**Rubric (for all polynomial-time reductions):** 10 points =

+ 3 points for the reduction itself
  – For an NP-hardness proof, the reduction must be from a known NP-hard problem. You can use any of the NP-hard problems listed in the lecture notes (except the one you are trying to prove NP-hard, of course).

+ 3 points for the “if” proof of correctness
+ 3 points for the “only if” proof of correctness
+ 1 point for writing “polynomial time”

• An incorrect polynomial-time reduction that still satisfies half of the correctness proof is worth at most 4/10.
• A reduction in the wrong direction is worth 0/10.
1. A subset $S$ of vertices in an undirected graph $G$ is called \textit{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.

2. A subset $S$ of vertices in an undirected graph $G$ is called \textit{triangle-free} if, for every triple of vertices $u, v, w \in S$, at least one of the three edges $uv, uw, vw$ is \textit{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.
This is \textit{not} the largest triangle-free subset in this graph.

3. Charon needs to ferry $n$ recently deceased people across the river Acheron into Hades. Certain pairs of these people are sworn enemies, who cannot be together on either side of the river unless Charon is also present. (If two enemies are left alone, one will steal the obol from the other's mouth, leaving them to wander the banks of the Acheron as a ghost for all eternity. Let's just say this is a Very Bad Thing.) The ferry can hold at most $k$ passengers at a time, including Charon, and only Charon can pilot the ferry.

Prove that it is NP-hard to decide whether Charon can ferry all $n$ people across the Acheron unharmed.\footnote{Aside from being, you know, dead.} The input for Charon's problem consists of the integers $k$ and $n$ and an $n$-vertex graph $G$ describing the pairs of enemies. The output is either \textsc{true} or \textsc{false}. 

\footnote{Aside from being, you know, dead.}
Problem 3 is a generalization of the following extremely well-known puzzle, whose first known appearance is in the treatise *Propositiones ad Acuendos Juvenes* [*Problems to Sharpen the Young*] by the 8th-century English scholar Alcuin of York.

XVIII. Propositio De Homine et Capra et Lvpo.

Homo quidam debebat ultra fluuium transferre lupum, capram, et fasciculum cauli. Et non potuit aliam nauem inuenire, nisi quae duos tantum ex ipsis ferre ualebat. Praeceptum itaque ei fuerat, ut omnia haec ultra illaes a omnino transferret. Dicat, qui potest, quomodo eis illaes is transire potuit?

**Solutio.** Simili namque tenore ducerem prius capram et dimitterem foris lupum et caulum. Tum deinde uenirem, lupumque transferrem: lupoque foris misso capram naui receptam ultra reducerem; capramque foris missam caulum transueherem ultra; atque iterum remigassem, capramque assumptam ultra duxissem. Sicque faciendo facta erit remigatio salubris, absque uoragine lacerationis.

In case your classical Latin is rusty, here is an English translation:

XVIII. The Problem of the Man, the Goat, and the Wolf.

A man needed to transfer a wolf, a goat, and a bundle of cabbage across a river. However, he found that his boat could only bear the weight of two [objects at a time, including the man]. And he had to get everything across unharmed. Tell me if you can: How they were able to cross unharmed?

**Solution.** In a similar fashion [as an earlier problem], I would first take the goat across and leave the wolf and cabbage on the opposite bank. Then I would take the wolf across; leaving the wolf on shore, I would retrieve the goat and bring it back again. Then I would leave the goat and take the cabbage across. And then I would row across again and get the goat. In this way the crossing would go well, without any threat of slaughter.

Please do not write your solution to problem 3 in classical Latin.

---

²At least, we think that’s who wrote it; the evidence for his authorship is rather circumstantial, although we do know from his correspondence with Charlemagne that he sent the emperor some “simple arithmetical problems for fun”. Most scholars believe that even if Alcuin is the actual author of the *Propositiones*, he didn’t come up with the problems himself, but just collected his problems from other sources. Some things never change.
Solved Problem

4. 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 $a \rightarrow b \rightarrow d \rightarrow g \rightarrow e \rightarrow b \rightarrow d \rightarrow c \rightarrow f \rightarrow a \rightarrow c \rightarrow f \rightarrow g \rightarrow e \rightarrow a$.

**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 $v^\flat$ and $v^\sharp$, along with three edges $vv^\flat$, $vv^\sharp$, and $v^\flat v^\sharp$.

I claim that $G$ has a Hamiltonian cycle if and only if $H$ has a double-Hamiltonian tour.

$\Rightarrow$ Suppose $G$ has a Hamiltonian cycle $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n \rightarrow v_1$. We can construct a double-Hamiltonian tour of $H$ by replacing each vertex $v_i$ with the following walk:

$$\cdots \rightarrow v_i \rightarrow v_i^\flat \rightarrow v_i^\sharp \rightarrow v_i^\flat \rightarrow v_i^\sharp \rightarrow v_i \rightarrow \cdots$$

$\Leftarrow$ 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 simply remove the vertices in $H \setminus G$ from $D$, we obtain a closed walk in $G$ that visits every vertex in $G$ once.

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

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 $H$.

**Common incorrect 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.](image)

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

$$v_1 \rightarrow v_1 \rightarrow v_2 \rightarrow v_2 \rightarrow v_3 \rightarrow \cdots \rightarrow v_n \rightarrow v_n \rightarrow v_1.$$

On the other hand, 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.](image)

**Rubric (for all polynomial-time reductions):**

10 points =
+ 3 points for the reduction itself
  - For an NP-hardness proof, the reduction must be from a known NP-hard problem. You can use any of the NP-hard problems listed in the lecture notes (except the one you are trying to prove NP-hard, of course).
+ 3 points for the “if” proof of correctness
+ 3 points for the “only if” proof of correctness
+ 1 point for writing “polynomial time”

• An incorrect polynomial-time reduction that still satisfies half of the correctness proof is worth at most 4/10.
• A reduction in the wrong direction is worth 0/10.
This homework is only for practice; it will not be graded. However, similar questions may appear on the final exam, so we still strongly recommend treating this as a regular homework. Solutions will be released next Tuesday as usual.

1. Recall that \( w^R \) denotes the reversal of string \( w \); for example, \( \text{TURING}^R = \text{GNIRUT} \). Prove that the following language is undecidable.

\[
\text{REVACCEPT} := \{ \langle M \rangle \mid M \text{ accepts } \langle M \rangle^R \}
\]

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

2. Let \( M \) be a Turing machine, let \( w \) be an arbitrary input string, and let \( s \) be an integer. We say that \( M \) accepts \( w \) in space \( |w|^2 \) if, given \( w \) as input, \( M \) accesses only the first \( s \) (or fewer) cells on its tape and eventually accepts.

(a) Sketch a Turing machine/algorithm that correctly decides the following language:

\[
\{ \langle M, w \rangle \mid M \text{ accepts } w \text{ in space } |w|^2 \}
\]

(b) Prove that the following language is undecidable:

\[
\{ \langle M \rangle \mid M \text{ accepts at least one string } w \text{ in space } |w|^2 \}
\]

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

(a) Prove that \( \text{SOMETIMESHALT} \) is undecidable.

(b) Sketch a Turing machine/algorithm that accepts \( \text{SOMETIMESHALT} \).
Solved Problem

4. For each of the following languages, either prove that the language is decidable, or prove that the language is undecidable.

(a) \( L_0 = \{ \langle M \rangle \mid \text{given any input string, } M \text{ eventually leaves its } \text{start state}\} \)

Solution: We can determine whether a given Turing machine \( M \) always leaves its start state by careful analysis of its transition function \( \delta \). As a technical point, I will assume that crashing on the first transition does not count as leaving the start state.

- If \( \delta(\text{start}, a) = (\cdot, \cdot, -1) \) for any input symbol \( a \in \Sigma \), then \( M \) crashes on input \( a \) without leaving the start state.
- If \( \delta(\text{start}, \square) = (\cdot, \cdot, -1) \), then \( M \) crashes on the empty input without leaving the start state.
- Otherwise, \( M \) moves to the right until it leaves the start state. There are two subcases to consider:
  - If \( \delta(\text{start}, \square) = (\text{start}, \cdot, +1) \), then \( M \) loops forever on the empty input without leaving the start state.
  - Otherwise, for any input string, \( M \) must eventually leave the start state, either when reading some input symbol or when reading the first blank.

It is straightforward (but tedious) to perform this case analysis with a Turing machine that receives the encoding \( \langle M \rangle \) as input. We conclude that \( L_0 \) is decidable. ■

(b) \( L_1 = \{ \langle M \rangle \mid M \text{ decides } L_0 \} \)

Solution:

- By part (a), there is a Turing machine that decides \( L_0 \).
- Let \( M_{\text{reject}} \) be a Turing machine that immediately rejects its input, by defining \( \delta(\text{start}, a) = \text{reject} \) for all \( a \in \Sigma \cup \{\square\} \). Then \( M_{\text{reject}} \) decides the language \( \emptyset \neq L_0 \).

Thus, Rice’s Decision Theorem implies that \( L_1 \) is undecidable.

(c) \( L_2 = \{ \langle M \rangle \mid M \text{ decides } L_1 \} \)

Solution: By part (b), no Turing machine decides \( L_1 \), which implies that \( L_2 = \emptyset \). Thus, \( M_{\text{reject}} \) correctly decides \( L_2 \). We conclude that \( L_2 \) is decidable. ■

(d) \( L_3 = \{ \langle M \rangle \mid M \text{ decides } L_2 \} \)

Solution: Because \( L_2 = \emptyset \), we have

\[
L_3 = \{ \langle M \rangle \mid M \text{ decides } \emptyset \} = \{ \langle M \rangle \mid \text{REJECT}(M) = \Sigma^* \}
\]

- We have already seen a Turing machine \( M_{\text{reject}} \) such that \( \text{REJECT}(M_{\text{reject}}) = \Sigma^* \).
- Let \( M_{\text{accept}} \) be a Turing machine that immediately accepts its input, by defining \( \delta(\text{start}, a) = \text{accept} \) for all \( a \in \Sigma \cup \{\square\} \). Then \( \text{REJECT}(M_{\text{accept}}) = \emptyset \neq \Sigma^* \).

Thus, Rice’s Rejection Theorem implies that \( L_1 \) is undecidable.
(e) $L_4 = \{ \langle M \rangle \mid M \text{ decides } L_3 \}$

**Solution:** By part (b), no Turing machine decides $L_3$, which implies that $L_4 = \emptyset$. Thus, $M_{\text{reject}}$ correctly decides $L_4$. We conclude that $L_4$ is **decidable**.

At this point, we have fallen into a loop. For any $k > 4$, define

$$L_k = \{ \langle M \rangle \mid M \text{ decides } L_{k-1} \}.$$ 

Then $L_k$ is decidable (because $L_k = \emptyset$) if and only if $k$ is even.

\[\Box\]

**Rubric:** 10 points: 4 for part (a) + 1½ for each other part.

**Rubric (for all undecidability proofs, out of 10 points):**

**Diagonalization:**
+ 4 for correct wrapper Turing machine
+ 6 for self-contradiction proof (= 3 for $\Leftarrow$ + 3 for $\Rightarrow$)

**Reduction:**
+ 4 for correct reduction
+ 3 for “if” proof
+ 3 for “only if” proof

**Rice’s Theorem:**
+ 4 for positive Turing machine
+ 4 for negative Turing machine
+ 2 for other details (including using the correct variant of Rice’s Theorem)
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:

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

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

You may freely use the following results, which are proved in the lecture notes:

**Lemma 1:** \(w \cdot \varepsilon = w\) for all strings \(w\).

**Lemma 2:** \(|w \cdot x| = |w| + |x|\) for all strings \(w\) and \(x\).

**Lemma 3:** \((w \cdot x) \cdot y = w \cdot (x \cdot y)\) for all strings \(w, x,\) and \(y\).

The **reversal** \(w^R\) of a string \(w\) is defined recursively as follows:

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

For example, \(\text{STRESSED}^R = \text{DESSERTS}\) and \(\text{WTF374}^R = 473\text{FTW}\).

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

[Hint: You need #2 to prove #3, but you may find it easier to solve #3 first.]

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

4. Give a formal recursive definition of \(#(a, w)\).
5. Prove that \(#(a, w \cdot z) = #(a, w) + #(a, z)\) 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 \(w\).
Give regular expressions for each of the following languages over the alphabet \{0, 1\}.

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

Work on these later:

7. All strings \( w \) such that in every prefix of \( w \), the number of 0s and 1s differ by at most 1.
8. All strings containing at least two 0s and at least one 1.
9. All strings \( w \) such that in every prefix of \( w \), the number of 0s and 1s differ by at most 2.
10. All strings in which the substring 000 appears an even number of times.
    (For example, 0001000 and 0000 are in this language, but 000000 is not.)
Describe deterministic finite-state automata that accept each of the following languages over the alphabet \( \Sigma = \{0, 1\} \). Describe briefly what each state in your DFAs means.

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 \( \delta \) are all be clear. Try to keep the number of states small.

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 no substring \( 000 \) appears before a \( 1 \).
5. All strings containing at least three \( 0 \)s.
6. Every string except \( 000 \). [Hint: Don’t try to be clever.]

Work on these 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 the substring \( 000 \) appears an even number of times.
    (For example, \( 0001000 \) and \( 0000 \) are in this language, but \( 000000 \) is 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 $\delta$ are all be clear. Try to keep the number of states small.

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

2. 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.

Work on these later:

3. All strings $w$ such that $\binom{|w|}{2} \pmod{6} = 4$. [Hint: Maintain both $\binom{|w|}{2} \pmod{6}$ and $|w| \pmod{6}$.]

4. All strings $w$ such that $\#_{10}(w) \pmod{10} = 4$, where $\#_{10}(w)$ denotes the number of times $10$ appears as a substring of $w$, and $F_n$ is the $n$th 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.

1. \( \{ \theta^{2^n} \mid n \geq 0 \} \)

2. \( \{ \theta^{2n}1^n \mid n \geq 0 \} \)

3. \( \{ \theta^m1^n \mid m \neq 2n \} \)

4. Strings over \( \{ \theta, 1 \} \) where the number of \( \theta \)'s is exactly twice the number of \( 1 \)'s.

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.

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 \( \{ \theta, 1 \}^* \), and some pair of substrings \( w_i \) and \( w_j \) are equal.

Work on these later:

7. \( \{ \theta^{n^2} \mid n \geq 0 \} \)

8. \( \{ w \in (\theta + 1)^* \mid w \text{ is the binary representation of a perfect square} \} \)
Let $L$ be an arbitrary regular language.

1. Prove that the language $\text{insert}_1(L) := \{x1y \mid xy \in L\}$ is regular.

   Intuitively, $\text{insert}_1(L)$ is the set of all strings that can be obtained from strings in $L$ by inserting exactly one $1$. For example, if $L = \{\epsilon, 00K!\}$, then $\text{insert}_1(L) = \{1, 100K!, 010K!, 001K!, 00K1!, 00K!1\}$.

2. Prove that the language $\text{delete}_1(L) := \{xy \mid x1y \in L\}$ is regular.

   Intuitively, $\text{delete}_1(L)$ is the set of all strings that can be obtained from strings in $L$ by deleting exactly one $1$. For example, if $L = \{101101, 00, \epsilon\}$, then $\text{delete}_1(L) = \{01101, 10101, 10110\}$.

---

Work on these later: (In fact, these might be easier than problems 1 and 2.)

3. Consider the following recursively defined function on strings:

   \[
   \text{stutter}(w) := \begin{cases} 
   \epsilon & \text{if } w = \epsilon \\
   aa \cdot \text{stutter}(x) & \text{if } w = ax \text{ for some symbol } a \text{ and some string } x
   \end{cases}
   \]

   Intuitively, $\text{stutter}(w)$ doubles every symbol in $w$. For example:

   - $\text{stutter(PRESTO)} = \text{PPRREESSTTOO}$
   - $\text{stutter(HOCUS} \diamond \text{POCUS) = HHOCCUUSS} \diamond \diamond \text{PPOCCCUUSS}$

   Let $L$ be an arbitrary regular language.

   (a) Prove that the language $\text{stutter}^{-1}(L) := \{w \mid \text{stutter}(w) \in L\}$ is regular.
   (b) Prove that the language $\text{stutter}(L) := \{\text{stutter}(w) \mid w \in L\}$ is regular.

4. Consider the following recursively defined function on strings:

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

   Intuitively, $\text{evens}(w)$ skips over every other symbol in $w$. For example:

   - $\text{evens(EXPELLIARMUS)} = \text{XELAMS}$
   - $\text{evens(AVADA} \diamond \text{KEDAVRA) = VD} \diamond \text{EAR}$.

   Once again, let $L$ be an arbitrary regular language.

   (a) Prove that the language $\text{evens}^{-1}(L) := \{w \mid \text{evens}(w) \in L\}$ is regular.
   (b) Prove that the language $\text{evens}(L) := \{\text{evens}(w) \mid w \in L\}$ is regular.
Alex showed the following context-free grammars in class on Tuesday; 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.
  
  \[
  S \to \epsilon \mid S(S) \quad \text{properly nested parentheses}
  \]

Here is a different grammar for the same language:

\[
S \to \epsilon \mid (S) \mid SS \quad \text{properly nested parentheses}
\]

- \(\{0^n1^m \mid m \neq n\}\). This is the set of all binary strings composed of some number of 0s followed by a different number of 1s.

\[
\begin{align*}
S & \to A \mid B & \{0^n1^m \mid m \neq n\} \\
A & \to 0A \mid 0C & \{0^n1^m \mid m > n\} \\
B & \to B1 \mid C1 & \{0^n1^m \mid m < n\} \\
C & \to \epsilon \mid 0C1 & \{0^n1^m \mid m = n\}
\end{align*}
\]

Give context-free grammars for each of the following languages. For each grammar, describe in English the language for each non-terminal, and in the examples above. As usual, we won’t get to all of these in section.

1. \(\{0^{2n}1^n \mid n \geq 0\}\)

2. \(\{0^n1^n \mid m \neq 2n\}\)
   
   [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 \(01\).]

3. \(\{0,1\}^* \setminus \{0^n1^n \mid n \geq 0\}\)
   
   [Hint: Extend the previous grammar. What’s missing?]

Work on these later:

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

5. \(\{0,1\}^* \setminus \{ww \mid w \in \{0,1\}^*\}\).
   
   [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!]
For each of the following languages over the alphabet $\Sigma = \{0, 1\}$, either prove the language is regular (by giving an equivalent regular expression, DFA, or NFA) or prove that the language is not regular (using a fooling set argument). Exactly half of these languages are regular.

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

2. $\{0^n 1^m w \mid n \geq 0 \text{ and } w \in \Sigma^*\}$

3. $\{w0^n 1^n x \mid w \in \Sigma^* \text{ and } n \geq 0 \text{ and } x \in \Sigma^*\}$

4. Strings in which the number of 0s and the number of 1s differ by at most 2.

5. Strings such that in every prefix, the number of 0s and the number of 1s differ by at most 2.

6. Strings such that in every substring, the number of 0s and the number of 1s differ by at most 2.
Design Turing machines $M = (Q, \Sigma, \Gamma, \delta, \text{start}, \text{accept}, \text{reject})$ for each of the following tasks, either by listing the states $Q$, the tape alphabet $\Gamma$, and the transition function $\delta$ (in a table), or by drawing the corresponding labeled graph.

Each of these machines uses the input alphabet $\Sigma = \{1, \#\}$; the tape alphabet $\Gamma$ can be any superset of $\{1, \#, \square, \triangleright\}$ where $\square$ is the blank symbol and $\triangleright$ is a special symbol marking the left end of the tape. Each machine should reject any input not in the form specified below.

1. On input $1^n$, for any non-negative integer $n$, write $1^n1^n$ on the tape and accept.

2. On input $\#^n1^m$, for any non-negative integers $m$ and $n$, write $1^m$ on the tape and accept. In other words, delete all the $\#$s and shift the $1$s to the start of the tape.

3. On input $\#1^n$, for any non-negative integer $n$, write $\#1^{2n}$ on the tape and accept. [Hint: Modify the Turing machine from problem 1.]

4. On input $1^n$, for any non-negative integer $n$, write $1^{2^n}$ on the tape and accept. [Hint: Use the three previous Turing machines as subroutines.]
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.


   (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:

   $\begin{array}{cccccccc}
   9 & 7 & 7 & 2 & 1 & 3 & 7 & 5
   \end{array}$

   Describe and analyze a fast algorithm that returns the index of one local minimum. For example, given the input 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 $n$th smallest element) of the union $A \cup B$. For example, given the input

   $\begin{array}{cccccccc}
   A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \\
   B[1..8] = [2, 4, 5, 8, 17, 19, 21, 23]
   \end{array}$

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

   **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 $k$th smallest element in the union $A \cup B$. For example, given the input

   $\begin{array}{cccccccc}
   A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \\
   B[1..5] = [2, 5, 7, 17, 19]
   \end{array}$

   $k = 6$

   your algorithm should return the integer 7.
In lecture, Alex described an algorithm of Karatsuba that multiplies two $n$-digit integers using $O(n^{\log_2 3})$ single-digit additions, subtractions, and multiplications. In this lab we’ll look at some extensions and applications of this algorithm.

1. Describe an algorithm to compute the product of an $n$-digit number and an $m$-digit number, where $m < n$, in $O(m^{\log_2 3} - 1) n$ time.

2. Describe an algorithm to compute the decimal representation of $2^n$ in $O(n^{\log_2 3})$ time. (The standard algorithm that computes one digit at a time requires $\Theta(n^2)$ time.)

3. Describe a divide-and-conquer algorithm to compute the decimal representation of an arbitrary $n$-bit binary number in $O(n^{\log_2 3})$ time. [Hint: Let $x = a \cdot 2^{n/2} + b$. Watch out for an extra log factor in the running time.]

Think about later:

4. Suppose we can multiply two $n$-digit numbers in $O(M(n))$ time. Describe an algorithm to compute the decimal representation of an arbitrary $n$-bit binary number in $O(M(n) \log n)$ time.
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 $\epsilon$ 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 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
   
   $\langle 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7 \rangle$

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

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

   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).
To think about later:


   For example, given the array
   
   $$(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7)$$
   
   your algorithm should return the integer 6, because $(3, 1, 2, 5, 9)$ is a longest convex subsequence (one of many).

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

   For example, given the array
   
   $$(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7)$$
   
   your algorithm should return the integer 7, because $(4, 9, 5, 3, 5, 9, 4)$ 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 \( \varepsilon \) are all substrings of the string 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 problems. For the first three, use the backtracking algorithms 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 \( i \geq 3 \).


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 is an automatic zero.

   (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.

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. This stage can be broken down into several smaller, relatively mechanical steps:

   (a) **Identify the subproblems.** What are all the different ways your recursive algorithm can 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 depend 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.
Lenny Rutenbar, the founding dean of the new Maximilian Q. Levchin College of Computer Science, has commissioned a series of snow ramps on the south slope of the Orchard Downs sledding hill¹ and challenged Bill Kudeki, head of the Department of Electrical and Computer Engineering, to a sledding contest. Bill and Lenny will both sled down the hill, each trying to maximize their air time. The winner gets to expand their department/college into both Siebel Center and the new ECE Building; the loser has to move their entire department/college under the Boneyard bridge next to Everitt Lab.

Whenever Lenny or Bill 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 i-th ramp, and Length[i] is the distance that any sledder who takes the i-th ramp will travel through the air.

   Describe and analyze an algorithm to determine the maximum total distance that Lenny or Bill can spend in the air.

2. Uh-oh. The university lawyers heard about Lenny and Bill's little bet and immediately objected. To protect the university from either lawsuits or 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 Lenny or Bill 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.

3. To think about later: When the lawyers realized that imposing their restriction didn’t immediately shut down the contest, they added a new restriction: No ramp can be used more than once! Disgusted by the legal interference, Lenny and Bill 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 Lenny and Bill 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.

¹The 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 \( 14 \):

\[
\begin{align*}
&1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\
&((1 + 1) \times (1 + 1 + 1 + 1)) + ((1 + 1) \times (1 + 1)) \\
&(1 + 1) \times (1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1) \\
&(1 + 1) \times (((1 + 1 + 1) \times (1 + 1)) + 1)
\end{align*}
\]

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 \( n \).

2. To think about later: 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:

\[
\begin{align*}
&1 + 3 - 2 - 5 + 1 - 6 + 7 = -1 \\
&(1 + 3 - (2 - 5)) + (1 - 6) + 7 = 9 \\
&(1 + (3 - 2)) - (5 + 1) - (6 + 7) = -17
\end{align*}
\]

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 \( 1 + 3(-2)(-5) + 1 - 6 + 7 = 33 \).
Recall the class scheduling problem described in lecture on Tuesday. We are given two arrays \( S[1..n] \) and \( F[1..n] \), where \( S[i] < F[i] \) for each \( i \), representing the start and finish times of \( n \) classes. Your goal is to find the largest number of classes you can take without ever taking two classes simultaneously. We showed in class that the following greedy algorithm constructs an optimal schedule:

Choose the course that \textit{ends first}, discard all conflicting classes, and recurse.

But this is not the only greedy strategy we could have tried. For each of the following alternative greedy algorithms, either prove that the algorithm always constructs an optimal schedule, or describe a small input example for which the algorithm does not produce an optimal schedule. Assume that all algorithms break ties arbitrarily (that is, in a manner that is completely out of your control). \textbf{Exactly three of these greedy strategies actually work.}

1. Choose the course \( x \) that \textit{ends last}, discard classes that conflict with \( x \), and recurse.
2. Choose the course \( x \) that \textit{starts first}, discard all classes that conflict with \( x \), and recurse.
3. Choose the course \( x \) that \textit{starts last}, discard all classes that conflict with \( x \), and recurse.
4. Choose the course \( x \) with \textit{shortest duration}, discard all classes that conflict with \( x \), and recurse.
5. Choose a course \( x \) that \textit{conflicts with the fewest other courses}, discard all classes that conflict with \( x \), and recurse.
6. If no classes conflict, choose them all. Otherwise, discard the course with \textit{longest duration} and recurse.
7. If no classes conflict, choose them all. Otherwise, discard a course that \textit{conflicts with the most other courses} and recurse.
8. Let \( x \) be the class with the \textit{earliest start time}, and let \( y \) be the class with the \textit{second earliest start time}.
   - If \( x \) and \( y \) are disjoint, choose \( x \) and recurse on everything but \( x \).
   - If \( x \) completely contains \( y \), discard \( x \) and recurse.
   - Otherwise, discard \( y \) and recurse.
9. If any course \( x \) completely contains another course, discard \( x \) and recurse. Otherwise, choose the course \( y \) that \textit{ends last}, discard all classes that conflict with \( y \), and recurse.
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 are the edges? Are they directed or undirected?
- If the vertices and/or edges have associated values, what are they?
- What problem do you need to solve on this graph?
- What standard algorithm are you using to solve that 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?

1. **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.](image)

   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.

2. Let \( G \) be a connected undirected graph. Suppose we start with two coins on two arbitrarily chosen vertices of \( G \). At every step, each coin must move to an adjacent vertex. 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 a graph \( G = (V, E) \) and two vertices \( u, v \in V \) (which may or may not be distinct).
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 are the edges? Are they directed or undirected?
- If the vertices and/or edges have associated values, what are they?
- What problem do you need to solve on this graph?
- What standard algorithm are you using to solve that 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?

1. Inspired by the previous lab, 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 any game, then A must rank better than B in the overall ranking.

   You are given the list of players and their ranking 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 ≤ s ≤ n identifying Judy’s home galaxy, and an array D[1 .. n] containing the distances of each galaxy from s.

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 a new algorithm to compute the maximum number of distinct galaxies Judy can visit. She can 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 $u \rightarrow v$ has negative weight. Assume the graph has no negative cycles. [Hint: Modify the input graph and run Dijkstra’s algorithm. Alternatively, don’t modify the input graph, but run Dijkstra’s algorithm anyway.]

2. 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 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.

To 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) 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 $\text{dist}[1..V, 1..V]$ 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 $u \rightarrow v$, 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 $w(u \rightarrow v)$ denote the weight that you used in your distance computation, and let $w'(u \rightarrow v)$ denote the correct weight of $u \rightarrow v$.

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

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

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

(d) To think about later: Argue that when $w(u \rightarrow v) < w'(u \rightarrow v)$, 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! 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 \$ \rightarrow ¥ \rightarrow € \rightarrow \$ 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 the matrix $R[1..n]$ of 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.

---

†′No, you can’t.
1. 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.*

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.

   (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 $G$.

   *Hint: You’ve seen this problem before.*

   (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.
To think about later:

3. Formally, a **proper coloring** of a graph $G = (V,E)$ is a function $c : V \rightarrow \{1, 2, \ldots, k\}$, 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.

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.*
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.

1. Recall the following $k$Color 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 $k$Color 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 node has degree at most 374?

3. Given an undirected graph $G$, does $G$ have a spanning tree with at most 374 leaves?
1. Recall that a 5-coloring of a graph $G$ is a function that assigns each vertex of $G$ a “color” from the set \{0, 1, 2, 3, 4\}, such that for any edge $uv$, vertices $u$ and $v$ are assigned different “colors”. 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 $5$Color problem.]

![A careful 5-coloring.](image)

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.

3. 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 finding the minimum number of colors in a weak bicoloring of a given graph is NP-hard.

   (b) Prove that finding the minimum number of colors in a strong bicoloring of a given graph is NP-hard.

![Left: A weak bicoloring of a 5-clique with four colors. Right: A strong bicoloring of a 5-cycle with five colors.](image)
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

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

- 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 $L = \text{HALT}$, 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 refer to “the Turing machine” or “the algorithm” or “the input string” or (god 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.

Prove that the following languages are undecidable.

1. $\text{ACCEPTILLINI} := \{ \langle M \rangle \mid M \text{ accepts the string } \text{ILLINI} \}$
2. $\text{ACCEPTTHREE} := \{ \langle M \rangle \mid M \text{ accepts exactly three strings} \}$
3. $\text{ACCEPTPALINDROME} := \{ \langle M \rangle \mid M \text{ accepts at least one palindrome} \}$
**Solution (for problem 1):** For the sake of argument, suppose there is an algorithm `DECIDE-ACCEPTILLINI` that correctly decides the language `ACCEPTILLINI`. Then we can solve the halting problem as follows:

\[
\text{DECIDEHALT}(\langle M, w \rangle):
\]

1. Encode the following Turing machine `M'`:
   \[
   \begin{align*}
   M'(x) : & \quad \text{run } M \text{ on input } w \\
   & \quad \text{return True}
   \end{align*}
   \]
   if `DECIDEACCEPTILLINI(\langle M' \rangle)`
   return True
   else
   return False

We prove this reduction correct as follows:

\[\Rightarrow\]
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`.

\[\Leftarrow\]
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 undecidability 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`. 
Rice's Theorem. Let $L$ be any set of languages that satisfies the following conditions:

- There is a Turing machine $Y$ such that $\text{Accept}(Y) \in L$.
- There is a Turing machine $N$ such that $\text{Accept}(N) \not\in L$.

The language $\text{AcceptIn}(L) := \{\langle M \rangle \mid \text{Accept}(M) \in L\}$ is undecidable.

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

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

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

1. $\text{Accept}\{\varepsilon\} := \{\langle M \rangle \mid M \text{ accepts only the string } \varepsilon; \text{ that is, } \text{Accept}(M) = \{\varepsilon\}\}$
2. $\text{Accept}\emptyset := \{\langle M \rangle \mid M \text{ does not accept any strings; that is, } \text{Accept}(M) = \emptyset\}$
3. $\text{Accept}\emptyset := \{\langle M \rangle \mid \text{Accept}(M) \text{ is not an acceptable language}\}$
4. $\text{Accept=Reject} := \{\langle M \rangle \mid \text{Accept}(M) = \text{Reject}(M)\}$
5. $\text{Accept}\neq\text{Reject} := \{\langle M \rangle \mid \text{Accept}(M) \neq \text{Reject}(M)\}$
6. $\text{Accept}\cup\text{Reject} := \{\langle M \rangle \mid \text{Accept}(M) \cup \text{Reject}(M) = \Sigma^*\}$
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, check “True” if the statement is always true and “False” otherwise. Each correct answer is worth +1 point; each incorrect answer is worth $-\frac{1}{2}$ point; checking “I don’t know” is worth +\frac{1}{4} point; and flipping a coin is (on average) worth +\frac{1}{4} point. You do not need to prove your answer is correct.

   Read each statement very carefully. Some of these are deliberately subtle.

   (a) If the moon is made of cheese, then Jeff is the Queen of England.
   (b) The language $\{0^m1^n \mid m, n \geq 0\}$ is not regular.
   (c) For all languages $L$, the language $L^*$ is regular.
   (d) For all languages $L \subset \Sigma^*$, if $L$ is recognized by a DFA, then $\Sigma^* \setminus L$ can be represented by a regular expression.
   (e) For all languages $L$ and $L'$, if $L \cap L' = \emptyset$ and $L'$ is not regular, then $L$ is regular.
   (f) For all languages $L$, if $L$ is not regular, then $L$ does not have a finite fooling set.
   (g) Let $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q, s, Q \setminus A, \delta)$ be arbitrary DFAs with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then $L(M) \cap L(M') = \emptyset$.
   (h) Let $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q, s, Q \setminus A, \delta)$ be arbitrary NFAs with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then $L(M) \cap L(M') = \emptyset$.
   (i) For all context-free languages $L$ and $L'$, the language $L \cdot L'$ is also context-free.
   (j) Every non-context-free language is non-regular.

2. For each of the following languages over the alphabet $\Sigma = \{0, 1\}$, either prove that the language is regular or prove that the language is not regular. Exactly one of these two languages is regular.

   (a) $\{0^n w 0^n \mid w \in \Sigma^+ \text{ and } n > 0\}$
   (b) $\{w 0^n w \mid w \in \Sigma^+ \text{ and } n > 0\}$

   For example, both of these languages contain the string 00110100000110100.
3. Let \( L = \{0^{2i}1^{i+2j}0^j \mid i, j \geq 0\} \) and let \( G \) be the following context free-grammar:

\[
S \rightarrow AB \\
A \rightarrow \varepsilon \mid 0A1 \\
B \rightarrow \varepsilon \mid 1B0
\]

(a) **Prove** that \( L(G) \subseteq L \).
(b) **Prove** that \( L \subseteq L(G) \).

[Hint: What are \( L(A) \) and \( L(B) \)? Prove it!]

4. For any language \( L \), let \( \text{Suffixes}(L) := \{x \mid \text{there exists } y \in \Sigma^* \text{ such that } xy \in L\} \) be the language containing all suffixes of all strings in \( L \). For example, if \( L = \{010, 101, 110\} \), then \( \text{Suffixes}(L) = \{\varepsilon, 0, 1, 01, 10, 010, 101, 110\} \).

**Prove** that for any regular language \( L \), the language \( \text{Suffixes}(L) \) is also regular.

5. For each of the following languages \( L \), give a regular expression that represents \( L \) and describe a DFA that recognizes \( L \).

(a) The set of all strings in \( \{0, 1\}^* \) that do not contain the substring \( 0110 \).
(b) The set of all strings in \( \{0, 1\}^* \) that contain exactly one of the substrings \( 01 \) or \( 10 \).

You do **not** need to prove that your answers are correct.
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, check “True” if the statement is always true and “False” otherwise. Each correct answer is worth +1 point; each incorrect answer is worth −\(\frac{1}{2}\) point; checking “I don’t know” is worth +\(\frac{1}{4}\) point; and flipping a coin is (on average) worth +\(\frac{1}{4}\) point. You do not need to prove your answer is correct.

Read each statement very carefully. Some of these are deliberately subtle.

(a) If \(2 + 2 = 5\), then Jeff is the Queen of England.
(b) The language \(\{0^m \# 0^n \mid m, n \geq 0\}\) is regular.
(c) For all languages \(L\), the language \(L^*\) is regular.
(d) For all languages \(L \subseteq \Sigma^*\), if \(L\) cannot be recognized by a DFA, then \(\Sigma^* \setminus L\) cannot be represented by a regular expression.
(e) For all languages \(L\) and \(L'\), if \(L \cap L' = \emptyset\) and \(L'\) is regular, then \(L\) is regular.
(f) For all languages \(L\), if \(L\) has a finite fooling set, then \(L\) is regular.
(g) Let \(M = (\Sigma, Q, s, A, \delta)\) and \(M' = (\Sigma, Q, s, Q \setminus A, \delta)\) be arbitrary NFAs (not necessarily different) with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then \(L(M) \cup L(M') = \Sigma^*\).
(h) Let \(M = (\Sigma, Q, s, A, \delta)\) and \(M' = (\Sigma, Q, s, Q \setminus A, \delta)\) be arbitrary NFAs (not necessarily different) with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then \(L(M) \cap L(M') = \emptyset\).
(i) For all context-free languages \(L\) and \(L'\), the language \(L \cdot L'\) is also context-free.
(j) Every non-regular language is context-free.

2. For each of the following languages over the alphabet \(\Sigma = \{0, 1\}\), either prove that the language is regular or prove that the language is not regular. Exactly one of these two languages is regular.

(a) \(\{w0^nw \mid w \in \Sigma^+ \text{ and } n > 0\}\)
(b) \(\{0^nw0^n \mid w \in \Sigma^+ \text{ and } n > 0\}\)

For example, both of these languages contain the string 00110100000110100.
3. Let \( L = \{1^m0^n \mid m \leq n \leq 2m\} \) and let \( G \) be the following context free-grammar:

\[
S \rightarrow 1S0 \mid 1S00 \mid \epsilon
\]

(a) **Prove** that \( L(G) \subseteq L \).
(b) **Prove** that \( L \subseteq L(G) \).

4. For any language \( L \), let \( \text{Prefixes}(L) := \{ x \mid xy \in L \text{ for some } y \in \Sigma^* \} \) be the language containing all prefixes of all strings in \( L \). For example, if \( L = \{000, 100, 110, 111\} \), then \( \text{Prefixes}(L) = \{\epsilon, 0, 1, 00, 10, 11, 000, 100, 110, 111\} \).

**Prove** that for any regular language \( L \), the language \( \text{Prefixes}(L) \) is also regular.

5. For each of the following languages \( L \), give a regular expression that represents \( L \) and describe a DFA that recognizes \( L \).

(a) The set of all strings in \( \{0, 1\}^* \) that contain either both or neither of the substrings \( 01 \) and \( 10 \).
(b) The set of all strings in \( \{0, 1\}^* \) that do not contain the substring \( 1001 \).

You do **not** need to prove that your answers are correct.
1. For each statement below, check “True” if the statement is always true and “False” otherwise. Each correct answer is worth +1 point; each incorrect answer is worth −½ point; checking “I don’t know” is worth +¼ point; and flipping a coin is (on average) worth +¼ point. You do not need to prove your answer is correct.

Read each statement very carefully. Some of these are deliberately subtle.

(a) If 100 is a prime number, then Jeff is the Queen of England.
(b) The language \( \{0^m 0^{n+m} 0^n \mid m, n \geq 0 \} \) is regular.
(c) For all languages \( L \), the language \( L^* \) is regular.
(d) For all languages \( L \subset \Sigma^* \), if \( L \) can be recognized by a DFA, then \( \Sigma^* \setminus L \) cannot be represented by a regular expression.
(e) For all languages \( L \) and \( L' \), if \( L \subseteq L' \) and \( L' \) is regular, then \( L \) is regular.
(f) For all languages \( L \), if \( L \) has a finite fooling set, then \( L \) is not regular.
(g) Let \( M = (\Sigma, Q, s, A, \delta) \) and \( M' = (\Sigma, Q, s, Q \setminus A, \delta) \) be arbitrary NFAs with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then \( L(M) \cup L(M') = \Sigma^* \).
(h) Let \( M = (\Sigma, Q, s, A, \delta) \) and \( M' = (\Sigma, Q, s, Q \setminus A, \delta) \) be arbitrary NFAs with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then \( L(M) \cap L(M') = \emptyset \).
(i) For all context-free languages \( L \) and \( L' \), the language \( L \cdot L' \) is also context-free.
(j) Every regular language is context-free.

2. For each of the following languages over the alphabet \( \Sigma = \{0, 1\} \), either prove that the language is regular or prove that the language is not regular. Exactly one of these two languages is regular.

(a) \( \{w0^n w \mid w \in \Sigma^+ \text{ and } n > 0\} \)
(b) \( \{0^n w0^n \mid w \in \Sigma^+ \text{ and } n > 0\} \)

For example, both of these languages contain the string 001101000001101000.
3. Let \( L = \{1^m \theta^n \mid n \leq m \leq 2n\} \) and let \( G \) be the following context free-grammar:

\[
S \rightarrow 1S \theta | 11S \theta | \epsilon
\]

(a) **Prove** that \( L(G) \subseteq L \).

(b) **Prove** that \( L \subseteq L(G) \).

4. For any language \( L \), let \( \text{Prefixes}(L) := \{x \mid xy \in L \text{ for some } y \in \Sigma^*\} \) be the language containing all prefixes of all strings in \( L \). For example, if \( L = \{000, 100, 110, 111\} \), then \( \text{Prefixes}(L) = \{\epsilon, 0, 1, 00, 10, 11, 000, 100, 110, 111\} \).

**Prove** that for any regular language \( L \), the language \( \text{Prefixes}(L) \) is also regular.

5. For each of the following languages \( L \), give a regular expression that represents \( L \) and describe a DFA that recognizes \( L \).

(a) The set of all strings in \( \{0, 1\}^* \) that contain either both or neither of the substrings \( 01 \) and \( 10 \).

(b) The set of all strings in \( \{0, 1\}^* \) that do not contain the substring \( 1010 \).

You do **not** need to prove that your answers are correct.
1. **Clearly** indicate the following structures in the directed graph below, or write NONE if the indicated structure does not exist. (There are several copies of the graph in the answer booklet.)

   (a) A depth-first spanning tree rooted at \( r \).
   (b) A breadth-first spanning tree rooted at \( r \).
   (c) A topological order.
   (d) The strongly connected components.

2. The following puzzles appear in my younger daughter’s math workbook. (I’ve put the solutions on the right so you don’t waste time solving them during the exam.)

   Describe and analyze an algorithm to solve arbitrary acute-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 first maze above has 13 vertices and 15 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 consecutive edges \( u \rightarrow v \rightarrow w \) in the walk, either \( \angle uvw = 180^\circ \) or \( 0 < \angle uvw < 90^\circ \). Assume you have a subroutine that can compute the angle between any two segments in \( O(1) \) time. Do **not** assume that angles are multiples of \( 1^\circ \).

---

3. 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 $k = 10$):

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>13</td>
<td>16</td>
<td>18</td>
<td>19</td>
<td>23</td>
<td>28</td>
<td>31</td>
<td>37</td>
<td>42</td>
<td>−4</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Describe and analyze an efficient 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 $k$.

4. 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 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.

5. 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 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 $k$th song on the schedule, you will be awarded exactly $\text{Score}[k]$ points, but then you will be physically unable to dance for the next $\text{Wait}[k]$ songs (that is, you cannot dance to songs $k + 1$ through $k + \text{Wait}[k]$). 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 $\text{Score}[1..n]$ and $\text{Wait}[1..n]$. 
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. (There are several copies of the graph in the answer booklet.)

   (a) A depth-first spanning tree rooted at r.
   (b) A breadth-first spanning tree rooted at r.
   (c) A topological order.
   (d) The strongly connected components.

2. The following puzzle appears in my younger daughter’s math workbook. (I’ve put the solution on the right so you don’t waste time solving it during the exam.)

   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 \(90^\circ < \angle uvw \leq 180^\circ\) for every pair of consecutive edges \(u \rightarrow v \rightarrow w\) in the walk. Assume you have a subroutine that can compute the angle between any two segments in \(O(1)\) time. Do not assume that angles are multiples of \(1^\circ\).

---


4. 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 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 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.

5. 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 \( \text{BANANAANANAS} \) is a shuffle of the strings \( \text{BANANA} \) and \( \text{ANANAS} \) in several different ways.

\[
\begin{align*}
\text{BANANAANANAS} & \quad \text{BANANANANAS} & \quad \text{BANANANAANAS}
\end{align*}
\]

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 \( B \).
1. **Clearly** indicate the following structures in the directed graph below, or write **NONE** if the indicated structure does not exist. (There are several copies of the graph in the answer booklet.)

   - (a) A depth-first spanning tree rooted at \(r\).
   - (b) A breadth-first spanning tree rooted at \(r\).
   - (c) A topological order.
   - (d) The strongly connected components.

2. The following puzzle appears in my younger daughter’s math workbook.¹ (I’ve put the solution on the right so you don’t waste time solving it during the exam.)

   ![Maze Puzzle](image)

   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 \(\text{Start}\) and \(\text{Finish}\).

   Your algorithm should return **True** if \(G\) contains a walk from \(\text{Start}\) to \(\text{Finish}\) that has only obtuse angles, and **False** otherwise. Formally, a walk through \(G\) is valid if \(90^\circ < \angle uvw \leq 180^\circ\) for every pair of consecutive edges \(u \rightarrow v \rightarrow w\) in the walk. Assume you have a subroutine that can compute the angle between any two segments in \(O(1)\) time. Do **not** assume that angles are multiples of 1°.

---


4. 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 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 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.

5. A palindrome is any string that is exactly the same as its reversal, like I, or DEED, or RACECAR, or AMANAPLANACATACANALPANAMA.

Describe and analyze an algorithm to find the length of the longest subsequence of a given string that is also a palindrome. For example, the longest palindrome subsequence of MAH dynamic PROGRAM ZLET MESH OW YOUTH EM is MHYMR ORMYHM, so given that string as input, your algorithm should output the number 11.
• Don’t panic!

• Please print your name and your NetID and circle your discussion section in the boxes above.

• This is a closed-book, closed-notes, closed-electronics exam. If you brought anything except your writing implements and your two double-sided 8½” × 11” cheat sheets, please put it away for the duration of the exam. In particular, you may not use any electronic devices.

• Please read the entire exam before writing anything. Please ask for clarification if any question is unclear.

• The exam lasts 180 minutes.

• If you run out of space for an answer, continue on the back of the page, or on the blank pages at the end of this booklet, but please tell us where to look. Alternatively, feel free to tear out the blank pages and use them as scratch paper.

• As usual, answering any (sub)problem with “I don’t know” (and nothing else) is worth 25% partial credit. Yes, even for problem 1. Correct, complete, but suboptimal solutions are always worth more than 25%. A blank answer is not the same as “I don’t know”.

• Beware the Three Deadly Sins. Give complete solutions, not examples. Don’t use weak induction. Declare all your variables.

• If you use a greedy algorithm, you must prove that it is correct to receive any credit. Otherwise, proofs are required only when we explicitly ask for them.

• Please return your cheat sheets and all scratch paper with your answer booklet.

• Good luck! And have a great winter break!
1. For each of the following questions, indicate every correct answer by marking the “Yes” box, and indicate every incorrect answer by marking the “No” box. Assume $P \neq NP$. If there is any other ambiguity or uncertainty, mark the “No” box. For example:

- $2 + 2 = 4$
- $x + y = 5$
- 3Sat can be solved in polynomial time.
- Jeff is not the Queen of England.

There are 40 yes/no choices altogether.

- Each correct choice is worth $+\frac{1}{2}$ point.
- Each incorrect choice is worth $-\frac{1}{4}$ point.
- To indicate “I don’t know”, write IDK next to the boxes; each IDK is worth $+\frac{1}{8}$ point.

(a) Which of the following statements is true for every language $L \subseteq \{\emptyset, 1\}^*$?

- $L$ contains the empty string $\varepsilon$.
- $L^*$ is infinite.
- $L^*$ is regular.
- $L$ is infinite or $L$ is decidable (or both).
- If $L$ is the union of two regular languages, then $L$ is regular.
- If $L$ is the union of two decidable languages, then $L$ is decidable.
- If $L$ is the union of two undecidable languages, then $L$ is undecidable.
- $L$ is accepted by some NFA with 374 states if and only if $L$ is accepted by some DFA with 374 states.
- If $L \notin P$, then $L$ is not regular.
(b) Which of the following languages over the alphabet \{0, 1\} are regular?

- Yes  No
  \{0^m 1^n \mid m \geq 0 \text{ and } n \geq 0\}
  All strings with the same number of 0s and 1s
- Yes  No
  Binary representations of all prime numbers less than 10^{100}
- Yes  No
  \{ww \mid w \text{ is a palindrome}\}
- Yes  No
  \{wxw \mid w \text{ is a palindrome and } x \in \{0, 1\}^*\}
- Yes  No
  \{\langle M \rangle \mid M \text{ accepts a finite number of non-palindromes}\}

(c) Which of the following languages over the alphabet \{0, 1\} are decidable?

- Yes  No
  \emptyset
- Yes  No
  \{0^n 1^{2n} 0^n 1^{2n} \mid n \geq 0\}
- Yes  No
  \{ww \mid w \text{ is a palindrome}\}
- Yes  No
  \{\langle M \rangle \mid M \text{ accepts a finite number of non-palindromes}\}
- Yes  No
  \{\langle M, w \rangle \mid M \text{ accepts } ww\}
- Yes  No
  \{\langle M, w \rangle \mid M \text{ accepts } ww \text{ after at most } |w|^2 \text{ transitions}\}

(d) Which of the following languages can be proved undecidable using Rice’s Theorem?

- Yes  No
  \{\langle M \rangle \mid M \text{ accepts an infinite number of strings}\}
- Yes  No
  \{\langle M \rangle \mid M \text{ accepts either } \langle M \rangle \text{ or } \langle M \rangle^R\}
- Yes  No
  \{\langle M \rangle \mid M \text{ does not accept exactly 374 palindromes}\}
- Yes  No
  \{\langle M \rangle \mid M \text{ accepts some string } w \text{ after at most } |w|^2 \text{ transitions}\}
(e) Which of the following is a good English specification of a recursive function that can be used to compute the edit distance between two strings $A[1..n]$ and $B[1..n]$?

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit($i, j$) is the answer for $i$ and $j$.</td>
<td></td>
</tr>
<tr>
<td>Edit($i, j$) is the edit distance between $A[i]$ and $B[j]$.</td>
<td></td>
</tr>
<tr>
<td>Edit[1..n, 1..n] stores the edit distances for all prefixes.</td>
<td></td>
</tr>
<tr>
<td>Edit($i, j$) is the edit distance between $A[i..n]$ and $B[j..n]$.</td>
<td></td>
</tr>
<tr>
<td>Edit[$i, j$] is the value stored at row $i$ and column $j$ of the table.</td>
<td></td>
</tr>
</tbody>
</table>

(f) Suppose we want to prove that the following language is undecidable.

$$MUGGLE := \{ \langle M \rangle \mid M \text{ accepts } SCIENCE \text{ but rejects } MAGIC \}$$

Professor Potter, your instructor in Defense Against Models of Computation and Other Dark Arts, suggests a reduction from the standard halting language

$$HALT := \{ \langle M, w \rangle \mid M \text{ halts on inputs } w \}.$$ 

Specifically, suppose there is a Turing machine DETECTOMUGGLETUM that decides MUGGLE. Professor Potter claims that the following algorithm decides HALT.

Define

\[
\text{DECEDEHALT}((M, w)):\]

Encode the following Turing machine:

\[
\text{RUBBERDUCK}(x):\]

- run $M$ on input $w$
- if $x = MAGIC$
  - return False
- else
  - return True

return DETECTOMUGGLETUM((RUBBERDUCK))

Which of the following statements is true for all inputs $\langle M, w \rangle$?

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>If $M$ rejects $w$, then RUBBERDUCK rejects MAGIC.</td>
<td></td>
</tr>
<tr>
<td>If $M$ accepts $w$, then DETECTOMUGGLETUM accepts $\langle RUBBERDUCK \rangle$.</td>
<td></td>
</tr>
<tr>
<td>If $M$ rejects $w$, then DECEDEHALT rejects $\langle M, w \rangle$.</td>
<td></td>
</tr>
<tr>
<td>DECEDEHALT decides the language HALT. (That is, Professor Potter's reduction is actually correct.)</td>
<td></td>
</tr>
<tr>
<td>DECEDEHALT actually runs (or simulates) RUBBERDUCK.</td>
<td></td>
</tr>
</tbody>
</table>
(g) Consider the following pair of languages:

- \text{HAMPath} := \{ G \mid \text{G is a directed graph with a Hamiltonian path}\}
- \text{ACYCLIC} := \{ G \mid \text{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 \textbf{must} be true, assuming P\neq\text{NP}?

\begin{tabular}{|c|c|}
\hline
\text{ACYCLIC} \in \text{NP} & Yes \hspace{1cm} No \\
\hline
\text{ACYCLIC} \cap \text{HAMPath} \in \text{P} & Yes \hspace{1cm} No \\
\hline
\text{HAMPath} \text{ is decidable}. & Yes \hspace{1cm} No \\
\hline
\text{There is no polynomial-time reduction from HAMPath to ACYCLIC}. & Yes \hspace{1cm} No \\
\hline
\text{There is no polynomial-time reduction from ACYCLIC to HAMPath}. & Yes \hspace{1cm} No \\
\hline
\end{tabular}
2. A \textit{quasi-satisfying assignment} for a \textit{3CNF} boolean formula $\Phi$ is an assignment of truth values to the variables such that \textit{at most one} clause in $\Phi$ does not contain a true literal. \textbf{Prove} that it is NP-hard to determine whether a given 3CNF boolean formula has a quasi-satisfying assignment.
3. Recall that a **run** in a string is a maximal non-empty substring in which all symbols are equal. For example, the string **0001111000** consists of three runs: a run of three **0**s, a run of four **1**s, and another run of three **0**s.

(a) Let \( L \) be the set of all strings in \( \{0, 1\}^* \) in which every run of **0**s has odd length and every run of **1**s has even length. For example, \( L \) contains \( \epsilon \) and **0000** and **0001111000**, but \( L \) does not contain **1** or **0000** or **110111000**.

Describe both a regular expression for \( L \) and a DFA that accepts \( L \).

(b) Let \( L' \) be the set of all non-empty strings in \( \{0, 1\}^* \) in which the number of runs is equal to the length of the first run. For example, \( L' \) contains **0** and **1100** and **000101**, but \( L' \) does not contain **0000** or **110111000** or \( \epsilon \).

**Prove** that \( L' \) is not a regular language.
4. Your cousin 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 \( i \)th 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.
5. **Prove** that each of the following languages is undecidable:

(a) \( \{ \langle M \rangle \mid M \text{ accepts } \text{RICESTHEOREM} \} \)

(b) \( \{ \langle M \rangle \mid M \text{ rejects } \text{RICESTHEOREM} \} \)  
    \( [\text{Hint: Use part (a), not Rice’s theorem}] \)
6. 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 \( c(uv) \).

\[ \text{[Hint: This is not the same Intergalactic Judy problem that you saw in lab.]} \]
(scratch paper)
(scratch paper)
You may assume the following problems are 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 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 $G$?

**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 an undirected graph $G$, is there a path in $G$ that visits every vertex exactly once?

**HamiltonianCycle**: Given an undirected graph $G$, is there a cycle in $G$ that visits every vertex exactly once?

**DirectedHamiltonianCycle**: Given an directed graph $G$, is there a directed 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 $G$?

**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?

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

You may assume the following languages are undecidable:

\[
\begin{align*}
\text{SelfReject} &:= \{ \langle M \rangle \mid M \text{ rejects } \langle M \rangle \} \\
\text{SelfAccept} &:= \{ \langle M \rangle \mid M \text{ accepts } \langle M \rangle \} \\
\text{SelfHalt} &:= \{ \langle M \rangle \mid M \text{ halts on } \langle M \rangle \} \\
\text{SelfDiverge} &:= \{ \langle M \rangle \mid M \text{ does not halt on } \langle M \rangle \} \\
\text{Reject} &:= \{ \langle M, w \rangle \mid M \text{ rejects } w \} \\
\text{Accept} &:= \{ \langle M, w \rangle \mid M \text{ accepts } w \} \\
\text{Halt} &:= \{ \langle M, w \rangle \mid M \text{ halts on } w \} \\
\text{Diverge} &:= \{ \langle M, w \rangle \mid M \text{ does not halt on } w \} \\
\text{NeverReject} &:= \{ \langle M \rangle \mid \text{Reject}(M) = \emptyset \} \\
\text{NeverAccept} &:= \{ \langle M \rangle \mid \text{Accept}(M) = \emptyset \} \\
\text{NeverHalt} &:= \{ \langle M \rangle \mid \text{Halt}(M) = \emptyset \} \\
\text{NeverDiverge} &:= \{ \langle M \rangle \mid \text{Diverge}(M) = \emptyset \}
\end{align*}
\]