• Each student must submit individual solutions for this homework. You may use any source at your disposal—paper, electronic, or human—but you must cite every source that you use. See the academic integrity policies on the course web site for more details. For all future homeworks, groups of up to three students will be allowed to submit joint solutions.

• Submit your solutions on standard printer/copier paper. At the top of each page, please clearly print your name and NetID, and indicate your registered discussion section. Use both sides of the paper. If you plan to write your solutions by hand, please print the last three pages of this homework as templates. If you plan to typeset your homework, you can find a \LaTeX template on the course web site; well-typeset homework will get a small amount of extra credit.

• Submit your solutions in the drop boxes outside 1404 Siebel. There is a separate drop box for each numbered problem. Don't staple your entire homework together. Don't give your homework to Jeff in class; he is fond of losing important pieces of paper.

• Avoid the Three Deadly Sins! There are a few dangerous writing (and thinking) habits that will trigger an automatic zero on any homework or exam problem. Yes, we are completely serious.
  – Give complete solutions, not just examples.
  – Declare all your variables.
  – Never use weak induction.

• Answering any homework or exam problem (or subproblem) in this course with “I don't know” and nothing else is worth 25% partial credit. We will accept synonyms like “No idea” or “WTF”, but you must write something.

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 Terminal Game is a two-person game played with pen and paper. The game begins by drawing a rectangle with \( n \) “terminals” protruding into the rectangles, for some positive integer \( n \), as shown in the figure below. On a player’s turn, she selects two terminals, draws a simple curve from one to the other without crossing any other curve (or itself), and finally draws a new terminal on each side of the curve. A player loses if it is her turn and no moves are possible, that is, if no two terminals may be connected without crossing at least one other curve.

![Initial setup, first turn, no more moves](image)

Analyze this game, answering the following questions (and any more that you determine the answers to): When is it better to play first, and when it is better to play second? Is there always a winning strategy? What is the smallest number of moves in which you can defeat your opponent? Prove your answers are correct.

2. Herr Professor Doktor Georg von den Dschungel has a 23-node binary tree, in which each node is labeled with a unique letter of the German alphabet, which is just like the English alphabet with four extra letters: Ä, Ö, Ü, and ß. (Don’t confuse these with A, O, U, and B!) Preorder and postorder traversals of the tree visit the nodes in the following order:

- Preorder: B K Ü E H L Z I Ö R C B T S O A Ä D F M N Ù G
- Postorder: H I Ö Z R L E C Ü S O T A ß K D M U G N F Ä B

(a) List the nodes in Professor von den Dschungel’s tree in the order visited by an inorder traversal.
(b) Draw Professor von den Dschungel’s tree.

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

- The empty string \( \epsilon \) is in \( L \).
- For any two strings \( x \) and \( y \) in \( L \), the string \( 0x1y0 \) is also in \( L \).
- These are the only strings in \( L \).

(a) Prove that the string \( 000010101010101000 \) is in \( L \).
(b) Prove by induction that every string in \( L \) has exactly twice as many 0s as 1s.
(c) Give an example of a string with exactly twice as many 0s as 1s that is not in \( L \).

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

\[
#(0,000010101010101000) = 12 \quad \text{and} \quad #(1,000010101010101000) = 6.
\]

You may assume without proof that \( #(a,xy) = #(a,x) + #(a,y) \) for any symbol \( a \) and any strings \( x \) and \( y \).
4. This is an extra credit problem. Submit your solutions in the drop box for problem 2 (but don’t staple your solutions for 2 and 4 together).

A perfect riffle shuffle, also known as a Faro shuffle, is performed by cutting a deck of cards exactly in half and then perfectly interleaving the two halves. There are two different types of perfect shuffles, depending on whether the top card of the resulting deck comes from the top half or the bottom half of the original deck. An out-shuffle leaves the top card of the deck unchanged. After an in-shuffle, the original top card becomes the second card from the top. For example:

\[
\text{OutShuffle}(A^\spadesuit 2^\spadesuit 3^\spadesuit 4^\spadesuit 5^\heartsuit 6^\heartsuit 7^\heartsuit 8^\heartsuit) = A^\spadesuit 5^\heartsuit 2^\spadesuit 6^\heartsuit 3^\spadesuit 7^\heartsuit 4^\spadesuit 8^\heartsuit
\]
\[
\text{InShuffle}(A^\spadesuit 2^\spadesuit 3^\spadesuit 4^\spadesuit 5^\heartsuit 6^\heartsuit 7^\heartsuit 8^\heartsuit) = 5^\heartsuit A^\spadesuit 6^\heartsuit 2^\spadesuit 7^\heartsuit 3^\spadesuit 8^\heartsuit 4^\spadesuit
\]

(If you are unfamiliar with playing cards, please refer to the Wikipedia article https://en.wikipedia.org/wiki/Standard_52-card_deck.)

Suppose we start with a deck of \(2^n\) distinct cards, for some non-negative integer \(n\). What is the effect of performing exactly \(n\) perfect in-shuffles on this deck? Prove your answer is correct!
1. Give regular expressions for each of the following languages over the alphabet \{0, 1\}. You do not need to prove your answers are correct.

   (a) All strings with an odd number of 1s.
   (b) All strings with at most three 0s.
   (c) All strings that do not contain the substring 010.
   (d) All strings in which every occurrence of the substring 00 occurs before every occurrence of the substring 11.

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

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

The reversal $L^R$ of a language $L$ is defined as the set of reversals of all strings in $L$:

\[
L^R := \{ w^R \mid w \in L \}
\]

(a) Prove that $(L^*)^R = (L^R)^*$ for every language $L$.

(b) Prove that the reversal of any regular language is also a regular language. (You may assume part (a) even if you haven’t proved it yet.)

You may assume the following facts without proof:

- $L^* \cdot L^* = L^*$ for every language $L$.
- $(w^R)^R = w$ for every string $w$.
- $(x \cdot y)^R = y^R \cdot x^R$ for all strings $x$ and $y$.

[Hint: Yes, all three proofs use induction, but induction on what? And yes, all three proofs.]

3. Describe context-free grammars for each of the following languages over the alphabet \{0, 1\}. Explain briefly why your grammars are correct; in particular, describe in English the language generated by each non-terminal in your grammars. (We are not looking for full formal proofs of correctness, but convincing evidence that you understand why your answers are correct.)

(a) The set of all strings with more than twice as many 0s as 1s.
(b) The set of all strings that are not palindromes.

*(c) [Extra credit] The set of all strings that are not equal to $ww$ for any string $w$.

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

   - Line comments: Strings of the form $// \ldots \text{\texttt{\textbackslash n}}$.
   - Block comments: Strings of the form $/* \ldots */$.

   Following the C99 standard, we explicitly disallow **nesting** comments of the same type. A line comment starts with $//$ and ends at the first $\text{\texttt{\textbackslash n}}$ after the opening $//$. A block comment starts with $/*$ and ends at the first $*/$ completely after the opening $/*$; in particular, every block comment has at least two $*$. For example, the following strings are all valid C comments:

   - /***
   - $//\diamond//\text{\texttt{\textbackslash n}}$
   - $/*///\diamond\text{\texttt{\textbackslash n}}\ast///*$.

   On the other hand, the following strings are not valid C comments:

   - $*/$
   - $//\diamond//\text{\texttt{\textbackslash n}}$
   - $/*//\ast///*$

   (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($\diamond$), newlines($\text{\texttt{\textbackslash n}}$), 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.
2. Construct a DFA for the following language over alphabet \{0, 1\}:

\[ L = \left\{ w \in \{0, 1\}^* \mid \text{the number represented by binary string } w \text{ is divisible by 19, but the length of } w \text{ is not a multiple of 23} \right\}. \]

You must explain in English how your DFA works. A formal description without an English explanation will receive no credit, even if it is correct. Don't even try to draw the DFA.

3. Prove that each of the following languages is not regular.
   (a) \( \{ w \in \{0\}^* \mid \text{length of } w \text{ is a perfect square; that is, } |w| = k^2 \text{ for some integer } k \} \).
   (b) \( \{ w \in \{0, 1\}^* \mid \text{the number represented by } w \text{ as a binary string is a perfect square} \} \).

*4. [Extra credit] Suppose \( L \) is a regular language which guarantees to contain at least one palindrome. Prove that if an \( n \)-state DFA \( M \) accepts \( L \), then \( L \) contains a palindrome of length polynomial in \( n \). What is the polynomial bound you get?

---

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

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

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

The C++ commenting 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 comments.
1. For each of the following regular expressions, describe or draw two finite-state machines:
   • An NFA that accepts the same language, using Thompson’s algorithm (described in class and in the notes)
   • An equivalent DFA, using the incremental subset construction described in class. For each state in your DFA, identify the corresponding subset of states in your NFA. Your DFA should have no unreachable states.
   (a) \((01 + 10)^*(0 + 1 + \varepsilon)\)
   (b) \(1^* + (10)^* + (100)^*\)

2. Prove that for any regular language \(L\), the following languages are also regular:
   (a) \(\text{SUBSTRINGS}(L) := \{x \mid wxy \in L \text{ for some } w, y \in \Sigma^*\}\)
   (b) \(\text{HALF}(L) := \{w \mid ww \in L\}\)

   \([\text{Hint: Describe how to transform a DFA for } L \text{ into NFAs for } \text{SUBSTRINGS}(L) \text{ and } \text{HALF}(L). \text{ What do your NFAs have to guess? Don’t forget to explain in English how your NFAs work.}]\)

3. Which of the following languages over the alphabet \(\Sigma = \{0, 1\}\) are regular and which are not? Prove your answers are correct. Recall that \(\Sigma^+\) denotes the set of all nonempty strings over \(\Sigma\).
   (a) \(\{wxw \mid w, x \in \Sigma^+\}\)
   (b) \(\{wx x \mid w, x \in \Sigma^+\}\)
   (c) \(\{wx y w \mid w, x, y \in \Sigma^+\}\)
   (d) \(\{wx x y \mid w, x, y \in \Sigma^+\}\)
1. For this problem, a subtree of a binary tree means any connected subgraph. A binary tree is complete if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the largest complete subtree of a given binary tree. Your algorithm should return both the root and the depth of this subtree.

The largest complete subtree of this binary tree has depth 2.

2. Consider the following cruel and unusual sorting algorithm.

\[
\begin{align*}
\text{CRUEL}(A[1..n]) : & \\
\text{if } n > 1 & \\
\text{CRUEL}(A[1..n/2]) & \\
\text{CRUEL}(A[n/2 + 1..n]) & \\
\text{UNUSUAL}(A[1..n]) & \\
\text{UNUSUAL}(A[1..n/4]) & \\
\text{UNUSUAL}(A[n/4 + 1..3n/4]) & \\
\text{UNUSUAL}(A[n/4 + 1..3n/4]) & \\
\end{align*}
\]

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?]

(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. In the early 20th century, a German mathematician developed a variant of the Towers of Hanoi game, which quickly became known in the American literature as “Liberty Towers”.¹ In this variant, there is a row of $k \geq 3$ pegs, numbered in order from 1 to $k$. In a single turn, for any index $i$, you can move the smallest disk on peg $i$ to either peg $i - 1$ or peg $i + 1$, subject to the usual restriction that you cannot place a bigger disk on a smaller disk. Your mission is to move a stack of $n$ disks from peg 1 to peg $k$.

(a) Describe and analyze a recursive algorithm for the case $k = 3$. Exactly how many moves does your algorithm perform?

(b) Describe and analyze a recursive algorithm for the case $k = n + 1$ that requires at most $O(n^3)$ moves. To simplify the algorithm, assume that $n$ is a power of 2. [Hint: Use part (a).]

(c) [Extra credit] Describe and analyze a recursive algorithm for the case $k = n + 1$ that requires at most $O(n^2)$ moves. Do not assume that $n$ is a power of 2. [Hint: Don’t use part (a).]

(d) [Extra credit] Describe and analyze a recursive algorithm for the case $k = \sqrt{n} + 1$ that requires at most a polynomial number of moves. To simplify the algorithm, assume that $n$ is a power of 4. What polynomial bound do you get? [Hint: Use part (a)!]

* (e) [Extra extra credit] Describe and analyze a recursive algorithm for arbitrary $n$ and $k$. How small must $k$ be (as a function of $n$) so that the number of moves is bounded by a polynomial in $n$? (This is actually an open research problem, a phrase which here means “Nobody knows the best answer.”)

---

¹No, not really. During World War I, many German-derived or Germany-related names were changed to more patriotic variants. For example, sauerkraut became “liberty cabbage”, hamburgers became “liberty sandwiches”, frankfurters became “Liberty sausages” or “hot dogs”, German measles became “liberty measles”, dachshunds became “liberty pups”, German shepherds became “Alsatians”, and pinochle (the card game) became “Liberty”. For more recent anti-French examples, see “freedom fries”, “freedom toast”, and “liberty lip lock”. Americans are weird.
1. *Dance Dance Revolution* is a dance video game, first introduced in Japan by Konami in 1998. Players stand on a platform marked with four arrows, pointing forward, back, left, and right, arranged in a cross pattern. During play, the game plays a song and scrolls a sequence of \( n \) arrows (\( \downarrow \), \( \uparrow \), \( \downarrow \), or \( \uparrow \)) from the bottom to the top of the screen. At the precise moment each arrow reaches the top of the screen, the player must step on the corresponding arrow on the dance platform. (The arrows are timed so that you'll step with the beat of the song.)

You are playing a variant of this game called “Vogue Vogue Revolution”, where the goal is to play perfectly but move as little as possible. When an arrow reaches the top of the screen, if one of your feet is already on the correct arrow, you are awarded one style point for maintaining your current pose. If neither foot is on the right arrow, you must move one (and only one) of your feet from its current location to the correct arrow on the platform. If you ever step on the wrong arrow, or fail to step on the correct arrow, or move more than one foot at a time, or move either foot when you are already standing on the correct arrow, or insult Beyoncé, all your style points are immediately taken away and you lose.

How should you move your feet to maximize your total number of style points? For purposes of this problem, assume you always start with your left foot on \( \downarrow \) and your right foot on \( \uparrow \), and that you've memorized the entire sequence of arrows. For example, if the sequence is \( \uparrow \uparrow \downarrow \downarrow \leftarrow \rightarrow \leftarrow \rightarrow \), you can earn 5 style points by moving your feet as shown below:

Describe and analyze an efficient algorithm to find the maximum number of style points you can earn during a given VVR routine. Your input is an array \( \text{Arrow}[1..n] \) containing the sequence of arrows.

2. Recall that a palindrome is any string that is exactly the same as its reversal, like \( I \), or \( DEED \), or \( RACECAR \), or \( AMANAPLANACATACANALPANAMA \).

Any string can be decomposed into a sequence of palindrome substrings. For example, the string \( \text{BUBBASEESABANANA} \) ("Bubba sees a banana.") can be broken into palindromes in the following ways (among many others):

\[
\begin{align*}
\text{BUB} & \bullet \text{BASEESAB} \bullet \text{ANANA} \\
\text{B} & \bullet \text{U} \bullet \text{BB} \bullet \text{A} \bullet \text{SEES} \bullet \text{ABA} \bullet \text{NAN} \bullet \text{A} \\
\text{B} & \bullet \text{U} \bullet \text{BB} \bullet \text{A} \bullet \text{SEES} \bullet \text{A} \bullet \text{B} \bullet \text{ANANA} \\
\text{B} & \bullet \text{U} \bullet \text{B} \bullet \text{A} \bullet \text{S} \bullet \text{E} \bullet \text{E} \bullet \text{S} \bullet \text{A} \bullet \text{B} \bullet \text{ANA} \bullet \text{N} \bullet \text{A}
\end{align*}
\]

Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string \( \text{BUBBASEESABANANA} \), your algorithm would return the integer 3.
3. Suppose you are given a DFA $M = (\{0, 1\}, Q, s, A, \delta)$ and a binary string $w \in \{0, 1\}^*$. 

(a) Describe and analyze an algorithm that computes the longest subsequence of $w$ that is accepted by $M$, or correctly reports that $M$ does not accept any subsequence of $w$.

*(b) [Extra credit] Describe and analyze an algorithm that computes the shortest supersequence of $w$ that is accepted by $M$, or correctly reports that $M$ does not accept any supersequence of $w$. (Recall that a string $x$ is a supersequence of $w$ if and only if $w$ is a subsequence of $x$.)

Analyze both of your algorithms in terms of the parameters $n = |w|$ and $k = |Q|$.

---

**Rubric (for all dynamic programming problems):** As usual, a score of $x$ on the following 10-point scale corresponds to a score of $\lceil x/3 \rceil$ on the 4-point homework scale.

- **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).** $-\frac{1}{2}$ 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 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.

- **Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, but this is not required for full credit.** If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, data 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 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 do include them, significantly more students turn in algorithms that meet the target time bound but don’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 Antarctican 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.


For every pair of snails, the Antarctican 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. Consider a weighted version of the class scheduling problem, where different classes offer different number of credit hours, which are of course totally unrelated to the duration of the class lectures. Given arrays \( S[1..n] \) of start times, an array \( F[1..n] \) of finishing times, and an array \( H[1..n] \) of credit hours as input, your goal is to choose a set of non-overlapping classes with the largest possible number of credit hours.

(a) Prove that the greedy algorithm described in class — Choose the class that ends first and recurse — does not always return the best schedule.

(b) Describe an efficient algorithm to compute the best schedule.

In addition to submitting a solution on paper as usual, please individually submit an electronic solution for this problem on CrowdGrader. Please see the course web page for detailed instructions.
3. Suppose you have just purchased a new type of hybrid car that uses fuel extremely efficiently, but can only travel 100 miles on a single battery. The car's fuel is stored in a single-use battery, which must be replaced after at most 100 miles. The actual fuel is virtually free, but the batteries are expensive and can only be installed by licensed battery-replacement technicians. Thus, even if you decide to replace your battery early, you must still pay full price for the new battery to be installed. Moreover, because these batteries are in high demand, no one can afford to own more than one battery at a time.

Suppose you are trying to get from San Francisco to New York City on the new Inter-Continental Super-Highway, which runs in a direct line between these two cities. There are several fueling stations along the way; each station charges a different price for installing a new battery. Before you start your trip, you carefully print the Wikipedia page listing the locations and prices of every fueling station on the ICSH. Given this information, how do you decide the best places to stop for fuel?

More formally, suppose you are given two arrays $D[1..n]$ and $C[1..n]$, where $D[i]$ is the distance from the start of the highway to the $i$th station, and $C[i]$ is the cost to replace your battery at the $i$th station. Assume that your trip starts and ends at fueling stations (so $D[1] = 0$ and $D[n]$ is the total length of your trip), and that your car starts with an empty battery (so you must install a new battery at station 1).

(a) Describe and analyze a greedy algorithm to find the minimum number of refueling stops needed to complete your trip. Don't forget to prove that your algorithm is correct.

(b) But what you really want to minimize is the total cost of travel. Show that your greedy algorithm in part (a) does not produce an optimal solution when extended to this setting.

(c) Describe an efficient algorithm to compute the locations of the fuel stations you should stop at to minimize the total cost of travel.
1. You are standing next to a water pond, and you have three empty jars. Each jar holds a positive integer number of gallons; the capacities of the three jars may or may not be different. You want one of the jars (which one doesn’t matter) to contain exactly $k$ gallons of water, for some integer $k$. You are only allowed to perform the following operations:

(a) Fill a jar with water from the pond until the jar is full.
(b) Empty a jar of water by pouring water into the pond.
(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 pond.
- Fill the first jar from the third jar. (Now the third jar holds 9 gallons.)
- Empty the first jar into the pond.
- Fill the second jar from the pond.
- Fill the first jar from the second jar. (Now the second jar holds 4 gallons.)
- Empty the second jar into the third jar.

Describe an efficient algorithm that finds the minimum number of operations required to obtain a jar containing exactly $k$ gallons of water, or reports correctly that obtaining exactly $k$ gallons of water is impossible, given the capacities of the three jars and a positive integer $k$ as input. 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).

2. Consider a directed graph $G$, where each edge is colored either red, white, or blue. A walk$^1$ 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 $v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k$ is a French flag path if, for every integer $i$, the edge $v_i \rightarrow v_{i+1}$ is red if $i \mod 3 = 0$, white if $i \mod 3 = 1$, and blue if $i \mod 3 = 2$.

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.

3. Suppose we are given a directed acyclic graph $G$ where every edge $e$ has a positive integer weight $w(e)$, along with two specific vertices $s$ and $t$ and a positive integer $W$.

(a) Describe an efficient algorithm to find the longest path (meaning the largest number of edges) from $s$ to $t$ in $G$ with total weight at most $W$. [Hint: Use dynamic programming.]

(b) [Extra credit] Solve part (a) with a running time that does not depend on $W$.

$^1$Recall that a walk in $G$ is a sequence of vertices $v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k$, such that $v_{i-1} \rightarrow v_i$ is an edge in $G$ for every index $i$. A path is a walk in which no vertex appears more than once.
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. It is well known that the global economic collapse of 2017 was caused by computer scientists indiscriminately abusing weaknesses in the currency exchange market. *Arbitrage* was a money-making scheme that takes advantage of inconsistencies in currency exchange rates. Suppose a currency trader with $1,000,000 discovered that 1 US dollar could be traded for 120 Japanese yen, 1 yen could be traded for 0.01 euros, and 1 euro could be traded for 1.2 US dollars. Then by converting his money from dollars to yen, then from yen to euros, and finally from euros back to dollars, the trader could instantly turn his $1,000,000 into $1,440,000! The cycle of currencies $\rightarrow ¥ \rightarrow € \rightarrow$ was called an *arbitrage cycle*. Finding and exploiting arbitrage cycles before the prices were corrected required extremely fast algorithms. Of course, now that the entire world uses plastic bags as currency, such abuse is impossible.

Suppose $n$ different currencies are traded in the global currency market. You are given a two-dimensional array $Exch[1..n, 1..n]$ of exchange rates between every pair of currencies; for all indices $i$ and $j$, one unit of currency $i$ buys $Exch[i, j]$ units of currency $j$. (Do not assume that $Exch[i, j] \cdot Exch[j, i] = 1$.)

(a) Describe an algorithm that computes an array $Most[1..n]$, where $Most[i]$ is the largest 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 array of currency exchange rates creates an arbitrage cycle.

3. Describe and analyze an algorithm to find the *second smallest spanning tree* of a given undirected graph $G$ with weighted edges, that is, the spanning tree of $G$ with smallest total weight except for the minimum spanning tree. Because the minimum spanning tree is haunted, or something.
The following questions ask you to describe various Turing machines. In each problem, give both a formal description of your Turing machine in terms of specific states, tape symbols, and transition functions and explain in English how your Turing machine works. In particular:

- Clearly specify what variant of Turing machine you are using: Number of tapes, number of heads, allowed head motions, halting conditions, and so on.
- Include the type signature of your machine’s transition function. The standard model uses a transition function whose signature is $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{-1, +1\}$.
- If necessary, break your Turing machine into smaller functional pieces, and describe those pieces separately (both formally and in English).
- Use state names that convey their meaning/purpose.

1. Describe a Turing machine that computes the function $\lceil \log_2 n \rceil$. Given the string $1^n$ as input, for any positive integer $n$, your machine should return the string $1^{\lceil \log_2 n \rceil}$ as output. For example, given the input string $1111111111111$ (thirteen 1s), your machine should output the string $1111$, because $2^3 < 13 \leq 2^4$.

2. A binary-tree Turing machine uses an infinite binary tree as its tape; that is, every cell in the tape has a left child and a right child. At each step, the head moves from its current cell to its Parent, its Left child, or to its Right child. Thus, the transition function of such a machine has the form $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{P, L, R\}$. The input string is initially given along the left spine of the tape.

Prove that any binary-tree Turing machine can be simulated by a standard Turing machine. That is, given any binary-tree Turing machine $M = (\Gamma, \Box, \Sigma, Q, \text{start, accept, reject, } \delta)$, describe a standard Turing machine $M' = (\Gamma', \Box', \Sigma, Q', \text{start', accept', reject', } \delta')$ that accepts and rejects exactly the same input strings as $M$. Be sure to describe how a single transition of $M$ is simulated by $M'$.

In addition to submitting paper solutions, please also electronically submit your solution to this problem on CrowdGrader.

3. [Extra credit]

A tag-Turing machine has two heads: one can only read, the other can only write. Initially, the read head is located at the left end of the tape, and the write head is located at the first blank after the input string. At each transition, the read head can either move one cell to the right or stay put, but the write head must write a symbol to its current cell and move one cell to the right. Neither head can ever move to the left.

Prove that any standard Turing machine can be simulated by a tag-Turing machine. That is, given any standard Turing machine $M$, formally describe a tag-Turing machine $M'$ that accepts and rejects exactly the same strings as $M$. Be sure to describe how a single transition of $M$ is simulated by $M'$.
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.
   (b) Describe and analyze a polynomial-time algorithm for BOXDEPTH. [Hint: Don’t try to optimize the running time; \( O(n^3) \) is good enough.]
   (c) Why don’t these two results imply that \( P=NP \)?

2. 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.](image)
   ![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.

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

   ![A triangle-free subset of 7 vertices.](image)
   This is not the largest triangle-free subset in this graph.

   In addition to submitting paper solutions, please also electronically submit your solution to this problem on CrowdGrader.

4. [Extra credit] Describe a direct polynomial-time reduction from 4COLOR to 3COLOR. (This is significantly harder than the opposite direction, which you’ll see in lab on Wednesday. Don’t go through the Cook-Levin Theorem.)
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.

$$\text{RevAccept} := \{ \langle M \rangle \mid M \text{ accepts } \langle M \rangle^R \}$$

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 $s$ if, given $w$ as input, $M$ accesses only the first $s$ cells on the tape and eventually accepts.

(a) Prove that the following language is decidable:

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

(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. [Extra credit] For each of the following languages, either prove that the language is decidable, or prove that the language is undecidable.

(a) $L_0 = \{ \langle M \rangle \mid \text{given any input string, } M \text{ eventually leaves its start state} \}$
(b) $L_1 = \{ \langle M \rangle \mid M \text{ decides } L_0 \}$
(c) $L_2 = \{ \langle M \rangle \mid M \text{ decides } L_1 \}$
(d) $L_3 = \{ \langle M \rangle \mid M \text{ decides } L_2 \}$
(e) $L_4 = \{ \langle M \rangle \mid M \text{ decides } L_3 \}$
1. Prove that every non-negative integer can be represented as the sum of distinct powers of 2. (“Write it in binary” is not a proof; it’s just a restatement of what you have to prove.)

2. Suppose you and your 8-year-old cousin Elmo decide to play a game with a rectangular bar of chocolate, which has been scored into an $n \times m$ grid of squares. You and Elmo alternate turns. On each turn, you or Elmo choose one of the available pieces of chocolate and break it along one of the grid lines into two smaller rectangles. Thus, at all times, each piece of chocolate is an $a \times b$ rectangle for some positive integers $a$ and $b$; in particular, a $1 \times 1$ piece cannot be broken into smaller pieces. The game ends when all the pieces are individual squares. The winner is the player who breaks the last piece.

   Describe a strategy for winning this game. When should you take the first move, and when should you offer it to Elmo? On each turn, how do you decide which piece to break and where? Prove your answers are correct. [Hint: Let’s play a $3 \times 3$ game. You go first. Oh, and I’m kinda busy right now, so could you just play for me whenever it’s my turn? Thanks.]

3. [To think about later] Now consider a variant of the previous chocolate-bar game, where on each turn you can either break a piece into two smaller pieces or eat a $1 \times 1$ piece. This game ends when all the chocolate is gone. The winner is the player who eats the last bite of chocolate (not the player who eats the most chocolate). Describe a strategy for winning this game, and prove that your strategy works.
These lab problems ask you to prove some simple claims about recursively-defined string functions and concatenation. In each case, we want a self-contained proof by induction that relies on the formal recursive definitions, not on intuition. In particular, your proofs must refer to the formal recursive definition of string concatenation:

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

You may also use any of the following facts, which we proved in class:

**Lemma 1:** Concatenating nothing does nothing: For every string \( w \), we have \( w \cdot \epsilon = w \).

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

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

---

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

\[ \#(0, 000010101010010100) = 12 \quad \text{and} \quad \#(1, 000010101010010100) = 6. \]

   (a) Give a formal recursive definition of \( \#(a, w) \).

   (b) Prove by induction that \( \#(a, w \cdot z) = \#(a, w) + \#(a, z) \) for any symbol \( a \) and any strings \( w \) and \( z \).

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

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

   (a) Prove that \((w \cdot x)^R = x^R \cdot w^R\) for all strings \( w \) and \( x \).

   (b) Prove that \((w^R)^R = w\) for every string \( w \).
Give regular expressions that describe each of the following languages over the alphabet \( \{0, 1\} \). We won’t get to all of these in section.

1. All strings containing at least three 0s.

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

3. All strings containing the substring 000.

4. All strings not containing the substring 000.

5. All strings in which every run of 0s has length at least 3.

6. All strings such that every substring 000 appears after every 1.

7. Every string except 000. [Hint: Don’t try to be clever.]

8. All strings \( w \) such that in every prefix of \( w \), the number of 0s and 1s differ by at most 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 00000 is not.)
Jeff showed the 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.

\[
S \rightarrow \varepsilon \mid S(S) \quad \text{properly nested parentheses}
\]

Here is a different grammar for the same language:

\[
S \rightarrow \varepsilon \mid (S) \mid SS \quad \text{properly nested parentheses}
\]

- \{0^m1^n \mid m \neq n\}. This is the set of all binary strings composed of some number of 0s followed by a different number of 1s.

\[
S \rightarrow A \mid B \quad \text{all strings } 0^m1^n \text{ where } m \neq n \\
A \rightarrow 0A \mid 0C \\nB \rightarrow B1 \mid C1 \\nC \rightarrow \varepsilon \mid 0C1 \\n\]

all strings \(0^m1^n \text{ where } m > n\) \\
all strings \(0^m1^n \text{ where } m < n\) \\
all strings \(0^n1^n \text{ for some integer } n\)

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. Binary palindromes: Strings over \(\{0, 1\}\) that are equal to their reversals. For example: 00111000 and 0100010, but not 0110.0.

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

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

4. \(\{0, 1\}^n \setminus \{0^{2n}1^n \mid n \geq 0\}\)

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 over \(\{0, 1\}\) where the number of 0s is equal to the number of 1s.

7. Strings over \(\{0, 1\}\) where the number of 0s is not equal to the number of 1s.
Construct DFA that accept each of the following languages over the alphabet \{0, 1\}. We won't get to all of these in section.

1. (a) \((0 + 1)^*\)
   (b) \(\emptyset\)
   (c) \(\{e\}\)

2. Every string except \(000\).

3. All strings containing the substring \(000\).

4. All strings \textit{not} containing the substring \(000\).

5. All strings in which the reverse of the string is the binary representation of an integer divisible by 3.

6. All strings \(w\) such that \textit{in every prefix of} \(w\), the number of 0s and 1s differ by at most 2.
Prove that each of the following languages is not regular.

1. Binary palindromes: Strings over \{0, 1\} that are equal to their reversals. For example: \texttt{00111100} and \texttt{0100010}, but not \texttt{01100}. \textit{[Hint: We did this in class.]}

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

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

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

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

6. \(\{0^{2^n} \mid n \geq 0\}\) — Strings of 0s whose length is a power of 2.

7. Strings of the form \(w_1\#w_2\#\cdots\#w_n\) for some \(n \geq 2\), where each substring \(w_i\) is a string in \(\{0, 1\}^*\), and some pair of substrings \(w_i\) and \(w_j\) are equal.
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. $\{0^n1^n | n \geq 0\}$

2. $\{0^n1^n0^n | n \geq 0\}$

3. $\{w0^n1^n0^n | w \in \Sigma^*\}$

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

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

6. Strings such that in every substring, the number of $0$s and the number of $1$s differ by at most 2.
Here are several problems that are easy to solve in $O(n)$ time, essentially by brute force. Your task is to design algorithms for these problems that are significantly faster.

1. (a) Suppose $A[1..n]$ is an array of $n$ distinct integers, sorted so that $A[1] < A[2] < \cdots < A[n]$. Each integer $A[i]$ could be positive, negative, or zero. Describe a fast algorithm that either computes an index $i$ such that $A[i] = i$ or correctly reports that no such index exists..

(b) Now suppose $A[1..n]$ is a sorted array of $n$ distinct positive integers. Describe an even faster algorithm that either computes an index $i$ such that $A[i] = i$ or correctly reports that no such index exists. [Hint: This is really easy.]

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

```
9 7 7 2 1 3 7 5 4 7 3 3 4 8 6 9
```

Describe and analyze a fast algorithm that returns the index of one local minimum. For example, given the array above, your algorithm could return the integer 5, because $A[5]$ is a local minimum. [Hint: With the given boundary conditions, any array must contain at least one local minimum. Why?]

3. (a) 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

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

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

(b) **To think about on your own:** 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

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

your algorithm should return the integer 7.
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, and prove that your algorithm is correct.

   (b) Now suppose $A[1..n]$ is a sorted array of $n$ distinct positive integers. Describe an even faster algorithm that either computes an index $i$ such that $A[i] = i$ or correctly reports that no such index exists. [Hint: This is really easy.]

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

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

   Describe and analyze a fast algorithm that returns the index of one local minimum. For example, given the array above, your algorithm could return the integer 5, because $A[5]$ is a local minimum. [Hint: With the given boundary conditions, any array must contain at least one local minimum. Why?]

3. (a) 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

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

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

   (b) To think about on your own: 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

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

   your algorithm should return the integer 7.
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, USEQU**, and the empty string $\epsilon$ 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 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).
A **subsequence** of a sequence (for example, an array, a linked list, or a string), obtained by removing zero or more elements and keeping the rest in the same sequence order. A subsequence is called a **substring** if its elements are contiguous in the original sequence. For example:

- **SUBSEQUENCE**, **UBSEQU**, and the empty string ϵ are all substrings of 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..ℓ] is increasing if B[i] > B[i − 1] for every index i ≥ 2.

2. Given an array A[1..n] of integers, compute the length of a longest **decreasing** subsequence of A. A sequence B[1..ℓ] is decreasing if B[i] < B[i − 1] for every index i ≥ 2.

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


5. Given an array A[1..n], compute the length of a longest **palindrome** subsequence of A. Recall that a sequence B[1..ℓ] is a palindrome if B[i] = B[ℓ − 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. Don't describe how to solve the problem at this stage; just describe what the problem actually is. Otherwise, the reader has no way to know what your recursive algorithm is supposed to compute.

   (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 can your recursive algorithm call itself, starting with some initial input? For example, the argument to RECFIBO is always an integer between 0 and $n$.

   (b) **Analyze space and running time.** The number of possible distinct subproblems determines the space complexity of your memoized algorithm. To compute the time complexity, add up the running times of all possible subproblems, **ignoring the recursive calls**. For example, if we already know $F_{i-1}$ and $F_{i-2}$, we can compute $F_i$ in $O(1)$ time, so computing the first $n$ Fibonacci numbers takes $O(n)$ time.

   (c) **Choose a data structure to memoize intermediate results.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. For some problems, however, a more complicated data structure is required.

   (d) **Identify dependencies between subproblems.** 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, this means you should consider the base cases first, then the subproblems that depends only on base cases, and so on. More formally, the dependencies you identified in the previous step define a partial order over the subproblems; in this step, you need to find a linear extension of that partial order. **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.
1. It’s almost time to show off your flippin’ sweet dancing skills! Tomorrow is the big dance contest you’ve been training for your entire life, except for that summer you spent with your uncle in Alaska hunting wolverines. You’ve obtained an advance copy of the the list of \( n \) songs that the judges will play during the contest, in chronological order.

You know all the songs, all the judges, and your own dancing ability extremely well. For each integer \( k \), you know that if you dance to the \( k \)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 \( \text{Wait}[1..n] \).

2. 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.

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

Similarly, the strings \( \text{PRODGYRANAMMIINCG} \) and \( \text{DYPRONGARMAMMICING} \) are both shuffles of \( \text{DYNAMIC} \) and \( \text{PROGRAMMING} \):

\[
\begin{align*}
\text{PRODGYRANAMMIINCG} & \quad \text{DYPRONGARMAMMICING} \\
\text{PRODGYRANAMMIINCG} & \quad \text{DYPRONGARMAMMICING}
\end{align*}
\]

Describe and analyze an efficient algorithm to determine, given three strings \( A[1..m], B[1..n], \) and \( C[1..m+n] \), whether \( C \) is a shuffle of \( A \) and \( B \).
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. Don’t describe how to solve the problem at this stage; just describe what the problem actually is. Otherwise, the reader has no way to know what your recursive algorithm is supposed to compute.

   (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 can your recursive algorithm call itself, starting with some initial input? For example, the argument to RECfibo is always an integer between 0 and n.

   (b) **Analyze space and running time.** The number of possible distinct subproblems determines the space complexity of your memoized algorithm. To compute the time complexity, add up the running times of all possible subproblems, ignoring the recursive calls. For example, if we already know $F_{i-1}$ and $F_{i-2}$, we can compute $F_i$ in $O(1)$ time, so computing the first $n$ Fibonacci numbers takes $O(n)$ time.

   (c) **Choose a data structure to memoize intermediate results.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. For some problems, however, a more complicated data structure is required.

   (d) **Identify dependencies between subproblems.** 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, this means you should consider the base cases first, then the subproblems that depends only on base cases, and so on. More formally, the dependencies you identified in the previous step define a partial order over the subproblems; in this step, you need to find a linear extension of that partial order. **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.
1. Suppose you are given a sequence of non-negative integers separated by + and × signs; for example:

\[ 2 \times 3 + 0 \times 6 \times 1 + 4 \times 2 \]

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

\[ 2 \times (3 + 0 \times (6 \times (1 + (4 \times 2)))) = 6 \]
\[ (((((2 \times 3) + 0) \times 6) \times 1) + 4) \times 2 = 80 \]
\[ ((2 \times 3) + (0 \times 6)) \times (1 + (4 \times 2)) = 108 \]
\[ (((2 \times 3) + 0) \times 6) \times ((1 + 4) \times 2) = 360 \]

Describe and analyze an algorithm to compute, given a list of integers separated by + and × signs, the largest possible value we can obtain by inserting parentheses.

Your input is an array \( A[0..2n] \) where each \( A[i] \) is an integer if \( i \) is even and + or × if \( i \) is odd. Assume any arithmetic operation in your algorithm takes \( O(1) \) time.
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. Don't describe how to solve the problem at this stage; just describe what the problem actually is. Otherwise, the reader has no way to know what your recursive algorithm is supposed to compute.

   (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 can your recursive algorithm call itself, starting with some initial input? For example, the argument to RECIBIO is always an integer between 0 and n.

   (b) **Analyze space and running time.** The number of possible distinct subproblems determines the space complexity of your memoized algorithm. To compute the time complexity, add up the running times of all possible subproblems, ignoring the recursive calls. For example, if we already know $F_{i-1}$ and $F_{i-2}$, we can compute $F_i$ in $O(1)$ time, so computing the first n Fibonacci numbers takes $O(n)$ time.

   (c) **Choose a data structure to memoize intermediate results.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. For some problems, however, a more complicated data structure is required.

   (d) **Identify dependencies between subproblems.** 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, this means you should consider the base cases first, then the subproblems that depends only on base cases, and so on. More formally, the dependencies you identified in the previous step define a partial order over the subproblems; in this step, you need to find a linear extension of that partial order. **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.
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 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).

[Hint: Exactly three of these greedy strategies actually work.]

1. Choose the course $x$ that ends last, discard classes that conflict with $x$, and recurse.
2. Choose the course $x$ that starts first, discard all classes that conflict with $x$, and recurse.
3. Choose the course $x$ that starts last, discard all classes that conflict with $x$, and recurse.
4. Choose the course $x$ with shortest duration, discard all classes that conflict with $x$, and recurse.
5. Choose a course $x$ that 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 longest duration and recurse.
7. If no classes conflict, choose them all. Otherwise, discard a course that conflicts with the most other courses and recurse.
8. Let $x$ be the class with the earliest start time, and let $y$ be the class with the 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 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. Upward straight arrows are ladders; downward wavy arrows are snakes.](image)

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

   At the end of the competition, \( m \) games have been played. You realized that you had forgotten to implement a proper rating system, and therefore decided to produce the overall ranking of all \( n \) players as you see fit. However, to avoid being too suspicious, if player \( A \) ranked better than player \( B \) in any game, then \( A \) must rank better than \( B \) in the overall ranking.

   You are given the list of players involved and the ranking in each of the \( m \) games. Describe and analyze an algorithm to produce an overall ranking of the \( n \) players that satisfies the condition, or correctly reports that it is impossible.

2. 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 \( e \) has an associated toll of \( c_e \) dollars, where \( c_e \) is a positive integer. A teleport-way can be used multiple times, but the toll must be paid every time it is used.

   Judy wants to travel from galaxy \( u \) to galaxy \( v \), but teleportation is not very pleasant and she would like to minimize the number of times she needs to teleport. However, she wants the total cost to be a multiple of five dollars, because carrying small bills is not pleasant either.

   (a) Describe and analyze an algorithm to compute the smallest number of times Judy needs to teleport to travel from galaxy \( u \) to galaxy \( v \) while the total cost is a multiple of five dollars.
   
   (b) Solve (a), but now assume that Judy has a coupon that allows her to waive the toll once.
Suppose we are given both an undirected graph $G$ with weighted edges and a minimum spanning tree $T$ of $G$.

1. Describe an efficient algorithm to update the minimum spanning tree when the weight of one edge $e \in T$ is decreased.

2. Describe an efficient algorithm to update the minimum spanning tree when the weight of one edge $e \not\in T$ is increased.

3. Describe an efficient algorithm to update the minimum spanning tree when the weight of one edge $e \in T$ is increased.

4. Describe an efficient algorithm to update the minimum spanning tree when the weight of one edge $e \not\in T$ is decreased.

In all cases, the input to your algorithm is the edge $e$ and its new weight; your algorithms should modify $T$ so that it is still a minimum spanning tree. Of course, we could just recompute the minimum spanning tree from scratch in $O(E \log V)$ time, but you can do better.
1. A looped tree is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has non-negative weight.

![A looped tree.](image)

(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.

2. After graduating you accept a job with Aerophobes-Ř-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of all the flights on the planet.

Suppose one of your customers wants to fly from city $X$ to city $Y$. Describe an algorithm to find a sequence of flights that minimizes the total time in transit—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights. [Hint: Build an appropriate graph from the input data and apply Dijkstra’s algorithm.]
Describe Turing machines that compute the following functions.

In particular, specify the transition functions $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{-1,+1\}$ for each machine either by writing out a table or by drawing a graph. Recall that $\delta(p, \$) = (q, @, +1)$ means that if the Turing machine is in state $p$ and reads the symbol $\$ from the tape, then it will change to state $q$, write the symbol @ to the tape, and move one step to the right. In a drawing of a Turing machine, this transition is indicated by an edge from $p$ to $q$ with the label “$\$/@,+1$”.

**Give your states short mnemonic names that suggest their purpose.** Naming your states well won’t just make it easier to understand; it will also make it easier to design.

1. **DOUBLE**: Given a string $w \in \{0,1\}^*$ as input, return the string $ww$ as output.

2. **POWER**: Given a string of the form $1^n$ as input, return the string $1^{2n}$ as output.
Describe how to simulate an arbitrary Turing machine to make it error-tolerant. Specifically, given an arbitrary Turing machine $M$, describe a new Turing machine $M'$ that accepts and rejects exactly the same strings as $M$, even though an evil pixie named Lenny will move the head of $M'$ to an arbitrary location on the tape some finite number of unknown times during the execution of $M'$.

You do not have to describe $M'$ in complete detail, but do give enough details that a seasoned Turing machine programmer could work out the remaining mechanical details.

As stated, this problem has no solution! If $M$ halts on all inputs after a finite number of steps, then Lenny can make any substring of the input string completely invisible to $M$. For example, if the true input string is INPUT-STRING, Lenny can make $M$ believe the input string is actually IMPING, by moving the head to the second I whenever it tries to move to R, and by moving the head to P when it tries to move to U. Because $M$ halts after a finite number of steps, Lenny only has a finite number of opportunities to move the head.

In fact, with more care, Lenny can make $M$ think the input string is any string that uses only symbols from the actual input string; if the true input string is INPUT-STRING, Lenny can make $M$ believe the input string is actually GRINNING-PUTIN-IS-GRINNING.)

However, there are several different ways to rescue the problem. For each of the following restrictions on Lenny’s behavior, and for any Turing machine $M$, one can design a Turing machine $M'$ that simulates $M$ despite Lenny’s interference.

- Lenny can move the head only a bounded number of times. For example: Lenny can move the head at most 374 times.
- Whenever Lenny moves the head, he changes the state of the machine to a special error state lenny.
- Whenever Lenny moves the head, he moves it to the left end of the tape.
- Whenever Lenny moves the head, he moves it to a blank cell to the right of all non-blank cells.
- Whenever Lenny moves the head, he moves it to a cell containing a particular symbol in the input alphabet, say 0.
Describe algorithms for the following problems. The input for each problem is string \( \langle M, w \rangle \) that encodes a standard (one-tape, one-track, one-head) Turing machine \( M \) whose tape alphabet is \( \{ \mathbf{0}, \mathbf{1}, \square \} \) and a string \( w \in \{ \mathbf{0}, \mathbf{1} \}^* \).

1. Does \( M \) accept \( w \) after at most \(|w|^2\) steps?
2. If we run \( M \) with input \( w \), does \( M \) ever move its head to the right?
2½. If we run \( M \) with input \( w \), does \( M \) ever move its head to the right twice in a row?
2¾. If we run \( M \) with input \( w \), does \( M \) move its head to the right more than \( 2^{|w|} \) times?
3. If we run \( M \) with input \( w \), does \( M \) ever change a symbol on the tape?
3½. If we run \( M \) with input \( w \), does \( M \) ever change a \( \square \) on the tape to either \( \mathbf{0} \) or \( \mathbf{1} \)?
4. If we run \( M \) with input \( w \), does \( M \) ever leave its start state?

In contrast, as we will see later, the following problems are all undecidable!

1. Does \( M \) accept \( w \)?
1½. If we run \( M \) with input \( w \), does \( M \) ever halt?
2. If we run \( M \) with input \( w \), does \( M \) ever move its head to the right three times in a row?
3. If we run \( M \) with input \( w \), does \( M \) ever change a \( \square \) on the tape to \( \mathbf{1} \)?
3½. If we run \( M \) with input \( w \), does \( M \) ever change either \( \mathbf{0} \) or \( \mathbf{1} \) on the tape to \( \square \)?
4. If we run \( M \) with input \( w \), does \( M \) ever reenter its start state?
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. Formally, **valid 3-coloring** of a graph $G = (V, E)$ is a function $c: V \to \{1, 2, 3\}$ such that $c(u) \neq c(v)$ for all $uv \in E$. Less formally, a valid 3-coloring assigns each vertex a color, which is either red, green, or blue, such that the endpoints of every edge have different colors.

   Suppose you are given a magic black box that somehow answers the following problem in polynomial time:
   - **Input**: An undirected graph $G$.
   - **Output**: True if $G$ has a valid 3-coloring, and False otherwise.

   Using this black box as a subroutine, describe an algorithm that solves the **3-coloring problem** in polynomial time:
   - **Input**: An undirected graph $G$.
   - **Output**: A valid 3-coloring of $G$, or None if there is no such coloring.

   [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.

- 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 almost always requires two separate steps:
  
  - 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. Recall that a Hamiltonian cycle in a graph $G$ is a cycle that goes through every vertex of $G$ exactly once. Now, a tonian cycle in a graph $G$ is a cycle that goes through at least half of the vertices of $G$, and a Hamiltonian circuit in a graph $G$ is a closed walk that goes through every vertex in $G$ exactly twice.

   (a) Prove that it is NP-hard to determine whether a given graph contains a tonian cycle.
   
   (b) Prove that it is NP-hard to determine whether a given graph contains a Hamiltonian circuit.
Proving that a problem $X$ is NP-hard requires several steps:

- Choose a problem $Y$ that you already know is NP-hard.

- 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 almost always requires two separate steps:
  
  – 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.

Recall that a Hamiltonian cycle in a graph $G$ is a cycle that visits every vertex of $G$ exactly once.

1. In class on Thursday, Jeff proved that it is NP-hard to determine whether a given directed graph contains a Hamiltonian cycle. Prove that it is NP-hard to determine whether a given undirected graph contains a Hamiltonian cycle.

2. A double Hamiltonian circuit in a graph $G$ is a closed walk that goes through every vertex in $G$ exactly twice. Prove that it is NP-hard to determine whether a given undirected graph contains a double Hamiltonian circuit.
Proving that a language $L$ is undecidable by reduction requires several steps:

- Choose a language $L'$ that you already know is undecidable. Typical choices for $L'$ include:

\[
\text{ACCEPT} := \{ \langle M, w \rangle \mid M \text{ accepts } w \} \\
\text{REJECT} := \{ \langle M, w \rangle \mid M \text{ rejects } w \} \\
\text{HALT} := \{ \langle M, w \rangle \mid M \text{ halts on } w \} \\
\text{DIVERGE} := \{ \langle M, w \rangle \mid M \text{ diverges on } w \} \\
\text{NEVERACCEPT} := \{ \langle M \rangle \mid \text{ACCEPT}(M) = \emptyset \} \\
\text{NEVERREJECT} := \{ \langle M \rangle \mid \text{REJECT}(M) = \emptyset \} \\
\text{NEVERHALT} := \{ \langle M \rangle \mid \text{HALT}(M) = \emptyset \} \\
\text{NEVERDIVERGE} := \{ \langle M \rangle \mid \text{DIVERGE}(M) = \emptyset \}
\]

- Describe an algorithm (really a Turing machine) $M'$ that decides $L'$, using a Turing machine $M$ that decides $L$ as a black box. Typically this algorithm has the following form:

Given a string $w$, transform it into another string $x$, such that $M$ accepts $x$ if and only if $w \in L'$.

- Prove that your Turing machine is correct. This almost always requires two separate steps:

  - Prove that if $M$ accepts $w$ then $w \in L'$.
  - Prove that if $M$ rejects $w$ then $w \notin L'$.

Prove that the following languages are undecidable:

1. \text{ACCEPTILLINI} := \{ \langle M \rangle \mid M \text{ accepts the string 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} \}
Prove that the following languages are undecidable using Rice’s Theorem:

**Rice’s Theorem.** Let $\mathcal{X}$ be any nonempty proper subset of the set of acceptable languages. The language $\text{AcceptIn}\mathcal{X} := \{\langle M \rangle \mid \text{Accept}(M) \in \mathcal{X} \}$ is undecidable.

1. $\text{AcceptRegular} := \{\langle M \rangle \mid \text{Accept}(M) \text{ is regular} \}$
2. $\text{AcceptIllini} := \{\langle M \rangle \mid M \text{ accepts the string ILLINI} \}$
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 languages are undecidable? How do you prove it?

1. $\text{Accept}\{\epsilon\} := \{\langle M \rangle \mid M \text{ only accepts the string } \epsilon, \text{ i.e. } \text{Accept}(M) = \{\epsilon\} \}$
2. $\text{Accept}\emptyset := \{\langle M \rangle \mid M \text{ does not accept any strings, i.e. } \text{Accept}(M) = \emptyset \}$
3. $\text{Accept}\emptyset := \{\langle M \rangle \mid \text{Accept}(M) \text{ is not an acceptable language} \}$
4. $\text{Accept}=\text{Reject} := \{\langle M \rangle \mid \text{Accept}(M) = \text{Reject}(M) \}$
5. $\text{Accept} \neq \text{Reject} := \{\langle M \rangle \mid \text{Accept}(M) \neq \text{Reject}(M) \}$
6. $\text{Accept} \cup \text{Reject} := \{\langle M \rangle \mid \text{Accept}(M) \cup \text{Reject}(M) = \Sigma^* \}$
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) For all languages $L_1$ and $L_2$, the language $L_1 \cup L_2$ is regular.

(c) For all languages $L \subseteq \Sigma^*$, if $L$ is not regular, then $\Sigma^* \setminus L$ cannot be represented by a regular expression.

(d) For all languages $L_1$ and $L_2$, if $L_1 \subseteq L_2$ and $L_1$ is regular, then $L_2$ is regular.

(e) For all languages $L_1$ and $L_2$, if $L_1 \subseteq L_2$ and $L_1$ is not regular, then $L_2$ is not regular.

(f) For all languages $L$, if $L$ is regular, then $L$ has no infinite fooling set.

(g) The language \( \{0^m1^n \mid 0 \leq m + n \leq 374 \} \) is regular.

(h) The language \( \{0^m1^n \mid 0 \leq m - n \leq 374 \} \) is regular.

(i) For every language $L$, if the language $L^R = \{w^R \mid w \in L\}$ is regular, then $L$ is also regular. (Here $w^R$ denotes the reversal of string $w$; for example, $(\text{BACKWARD})^R = \text{DRAWKCA}$.)

(j) Every context-free language is regular.

2. Let $L$ be the set of strings in $\{0,1\}^*$ in which every run of consecutive 0s has even length and every run of consecutive 1s has odd length.

(a) Give a regular expression that represents $L$.

(b) Construct a DFA that recognizes $L$.

You do not need to prove that your answers are correct.

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

(a) The set of all strings in which the substrings 00 and 11 appear the same number of times.

(b) The set of all strings in which the substrings 01 and 10 appear the same number of times.

For example, both of these languages contain the string 1100001101101.
4. Consider the following recursive function:

\[
stutter(w) := \begin{cases} 
\epsilon & \text{if } w = \epsilon \\
ax \cdot stutter(x) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^*
\end{cases}
\]

For example, \(stutter(00101) = 0000110011\).

**Prove** that for any regular language \(L\), the following languages are also regular.

(a) \(STUTTER(L) := \{stutter(w) \mid w \in L\}\).
(b) \(STUTTER^{-1}(L) := \{w \mid stutter(w) \in L\}\).

5. Recall that string concatenation and string reversal are formally defined as follows:

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

\[
w^R := \begin{cases} 
\epsilon & \text{if } w = \epsilon \\
x^R \cdot a & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^*
\end{cases}
\]

**Prove** that \((w \cdot x)^R = x^R \cdot w^R\), for all strings \(w\) and \(x\). Your proof should be complete, concise, formal, and self-contained.
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 $2 + 2 = 5$, then Jeff is not the Queen of England.
(b) For all languages $L$, the language $L^*$ is regular.
(c) For all languages $L \subseteq \Sigma^*$, if $L$ is can be represented by a regular expression, then $\Sigma^* \setminus L$ can also be represented by a regular expression.
(d) For all languages $L_1$ and $L_2$, if $L_2$ is regular and $L_1 \subseteq L_2$, then $L_1$ is regular.
(e) For all languages $L_1$ and $L_2$, if $L_2$ is not regular and $L_1 \subseteq L_2$, then $L_1$ is not regular.
(f) For all languages $L$, if $L$ is not regular, then every fooling set for $L$ is infinite.
(g) The language $\{0^m10^n | 0 \leq n - m \leq 374\}$ is regular.
(h) The language $\{0^m10^n | 0 \leq n + m \leq 374\}$ is regular.
(i) For every language $L$, if $L$ is not regular, then the language $L^R = \{w^R | w \in L\}$ is also not regular. (Here $w^R$ denotes the reversal of string $w$; for example, (BACKWARD)$^R$ = DRAWKBCAB.)
(j) Every context-free language is regular.

2. Let $L$ be the set of strings in $\{0, 1\}^*$ in which every run of consecutive $0$s has odd length and the total number of $1$s is even.

For example, the string 111100001011011000 is in $L$, because it has eight $1$s and three runs of consecutive $0$s, with lengths 5, 1, and 3.

(a) Give a regular expression that represents $L$.
(b) Construct a DFA that recognizes $L$.

You do not need to prove that your answers are correct.

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

(a) The set of all strings in which the substrings 10 and 01 appear the same number of times.
(b) The set of all strings in which the substrings 00 and 01 appear the same number of times.

For example, both of these languages contain the string 110001101101.
4. Consider the following recursive function:

\[
\text{odds}(w) := \begin{cases} 
  w & \text{if } |w| \leq 1 \\
  a \cdot \text{odds}(x) & \text{if } w = abx \text{ for some } a, b \in \Sigma \text{ and } x \in \Sigma^*
\end{cases}
\]

Intuitively, \( \text{odds} \) removes every other symbol from the input string, starting with the second symbol. For example, \( \text{odds}(0101110) = 0010 \).

**Prove** that for any regular language \( L \), the following languages are also regular.

(a) \( \text{ODDS}(L) := \{ \text{odds}(w) \mid w \in L \} \).

(b) \( \text{ODDS}^{-1}(L) := \{ w \mid \text{odds}(w) \in L \} \).

5. Recall that string concatenation and string reversal are formally defined as follows:

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

\[
w^R := \begin{cases} 
  \epsilon & \text{if } w = \epsilon \\
  x^R \cdot a & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^*
\end{cases}
\]

**Prove** that \((w \cdot x)^R = x^R \cdot w^R\), for all strings \( w \) and \( x \). Your proof should be complete, concise, formal, and self-contained. You may assume the following identities, which we proved in class:

- \( w \cdot (x \cdot y) = (w \cdot x) \cdot y \) for all strings \( w, x, \) and \( y \).
- \( |w \cdot x| = |w| + |x| \) for all strings \( w \) and \( x \).
1. **Clearly** indicate the edges of the following spanning trees of the weighted graph pictured below. (Pretend that the person grading your exam has bad eyesight.) Some of these subproblems have more than one correct answer. Yes, that edge on the right has negative weight.

(a) A depth-first spanning tree rooted at $s$
(b) A breadth-first spanning tree rooted at $s$
(c) A shortest-path tree rooted at $s$
(d) A minimum spanning tree

2. An array $A[0..n-1]$ of $n$ distinct numbers is **bitonic** if there are unique indices $i$ and $j$ such that $A[(i-1) \mod n] < A[i] > A[(i+1) \mod n]$ and $A[(j-1) \mod n] > A[j] < A[(j+1) \mod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,

$$
\begin{array}{cccccccc}
4 & 6 & 9 & 8 & 7 & 5 & 1 & 2 \\
\end{array}
$$

is bitonic, but

$$
\begin{array}{cccccccc}
3 & 6 & 9 & 8 & 7 & 5 & 1 & 4 \\
\end{array}
$$

is not bitonic.

Describe and analyze an algorithm to find the index of the **smallest** element in a given bitonic array $A[0..n-1]$ in $O(\log n)$ time. You may assume that the numbers in the input array are distinct. For example, given the first array above, your algorithm should return 6, because $A[6] = 1$ is the smallest element in that array.

3. Suppose you are given a directed graph $G = (V, E)$ and two vertices $s$ and $t$. Describe and analyze an algorithm to determine if there is a walk in $G$ from $s$ to $t$ (possibly repeating vertices and/or edges) whose length is divisible by 3.

For example, given the graph below, with the indicated vertices $s$ and $t$, your algorithm should return **TRUE**, because the walk $s \rightarrow w \rightarrow y \rightarrow x \rightarrow s \rightarrow w \rightarrow t$ has length 6.

[HInt: Build a (different) graph.]
4. The new swap-puzzle game *Candy Swap Saga XIII* involves \( n \) cute animals numbered 1 through \( n \). Each animal holds one of three types of candy: circus peanuts, Heath bars, and Cioccolateria Gardini chocolate truffles. You also have a candy in your hand; at the start of the game, you have a circus peanut.

To earn points, you visit each of the animals in order from 1 to \( n \). For each animal, you can either keep the candy in your hand or exchange it with the candy the animal is holding.

- If you swap your candy for another candy of the *same* type, you earn one point.
- If you swap your candy for a candy of a *different* type, you lose one point. (Yes, your score can be negative.)
- If you visit an animal and decide not to swap candy, your score does not change.

You *must* visit the animals in order, and once you visit an animal, you can never visit it again.

Describe and analyze an efficient algorithm to compute your maximum possible score. Your input is an array \( C[1..n] \), where \( C[i] \) is the type of candy that the \( i \)th animal is holding.

5. Let \( G \) be a directed graph with weighted edges, and let \( s \) be a vertex of \( G \). Suppose every vertex \( v \neq s \) stores a pointer \( \text{pred}(v) \) to another vertex in \( G \). Describe and analyze an algorithm to determine whether these predecessor pointers correctly define a single-source shortest path tree rooted at \( s \). Do *not* assume that \( G \) has no negative cycles.
1. **Clearly** indicate the edges of the following spanning trees of the weighted graph pictured below. (Pretend that the person grading your exam has bad eyesight.) Some of these subproblems have more than one correct answer. Yes, that edge on the right has negative weight.

   (a) A depth-first spanning tree rooted at s
   (b) A breadth-first spanning tree rooted at s
   (c) A shortest-path tree rooted at s
   (d) A minimum spanning tree

2. Farmers Boggis, Bunce, and Bean have set up an obstacle course for Mr. Fox. The course consists of a row of $n$ booths, each with an integer painted on the front with bright red paint, which could be positive, negative, or zero. Let $A[i]$ denote the number painted on the front of the $i$th booth. Everyone has agreed to the following rules:

   - At each booth, Mr. Fox must say either “Ring!” or “Ding!”.
   - If Mr. Fox says “Ring!” at the $i$th booth, he earns a reward of $A[i]$ chickens. (If $A[i] < 0$, Mr. Fox pays a penalty of $-A[i]$ chickens.)
   - If Mr. Fox says “Ding!” at the $i$th booth, he pays a penalty of $A[i]$ chickens. (If $A[i] < 0$, Mr. Fox earns a reward of $-A[i]$ chickens.)
   - Mr. Fox is forbidden to say the same word more than three times in a row. For example, if he says “Ring!” at booths 6, 7, and 8, then he must say “Ding!” at booth 9.
   - All accounts will be settled at the end; Mr. Fox does not actually have to carry chickens through the obstacle course.
   - If Mr. Fox violates any of the rules, or if he ends the obstacle course owing the farmers chickens, the farmers will shoot him.

Describe and analyze an algorithm to compute the largest number of chickens that Mr. Fox can earn by running the obstacle course, given the array $A[1..n]$ of booth numbers as input.

3. Let $G$ be a directed graph with weighted edges, and let $s$ be a vertex of $G$. Suppose every vertex $v \neq s$ stores a pointer $\text{pred}(v)$ to another vertex in $G$. Describe and analyze an algorithm to determine whether these predecessor pointers correctly define a single-source shortest path tree rooted at $s$. Do not assume that $G$ has no negative cycles.
4. An array \( A[0..n-1] \) of \( n \) distinct numbers is \textit{bitonic} if there are unique indices \( i \) and \( j \) such that 
\[ A[(i-1) \mod n] < A[i] > A[(i+1) \mod n] \] 
and 
\[ A[(j-1) \mod n] > A[j] < A[(j+1) \mod n] \].

In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,

\[
\begin{bmatrix}
4 & 6 & 9 & 8 & 7 & 5 & 1 & 2 & 3
\end{bmatrix}
\]

is bitonic, but

\[
\begin{bmatrix}
3 & 6 & 9 & 8 & 7 & 5 & 1 & 2 & 4
\end{bmatrix}
\]

is not bitonic.

Describe and analyze an algorithm to find the index of the \textit{smallest} element in a given bitonic array \( A[0..n-1] \) in \( O(\log n) \) time. You may assume that the numbers in the input array are distinct. For example, given the first array above, your algorithm should return 6, because \( A[6] = 1 \) is the smallest element in that array.

5. Suppose we are given an undirected graph \( G \) in which every vertex has a positive weight.

(a) Describe and analyze an algorithm to find a spanning tree of \( G \) with minimum total weight. (The total weight of a spanning tree is the sum of the weights of its vertices.)

(b) Describe and analyze an algorithm to find a path in \( G \) from one given vertex \( s \) to another given vertex \( t \) with minimum total weight. (The total weight of a path is the sum of the weights of its vertices.)
“CS 374”: Algorithms and Models of Computation, Fall 2014
Final Exam — Version A — December 16, 2014

Name: _____________________________________________
NetID: _____________________________________________
Section: 1 2 3

<table>
<thead>
<tr>
<th>#</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>20</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>70</td>
</tr>
<tr>
<td>Max</td>
<td>20</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>70</td>
</tr>
</tbody>
</table>

• 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.

• You have 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.

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

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

• 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”.

• 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:

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 + 2 = 4$</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>$x + y = 5$</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>3SAT can be solved in polynomial time.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Jeff is not the Queen of England.</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

There are 40 yes/no choices altogether, each worth $\frac{1}{2}$ point.

(a) Which of the following statements is true for every language $L \subseteq \{0,1\}^*$?

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$ is non-empty.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$L$ is decidable or $L$ is infinite (or both).</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$L$ is accepted by some DFA with 42 states if and only if $L$ is accepted by some NFA with 42 states.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>If $L$ is regular, then $L \in NP$.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$L$ is decidable if and only if its complement $\overline{L}$ is undecidable.</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

(b) Which of the following computational models can be simulated by a deterministic Turing machine with three read/write heads, with at most polynomial slow-down in time, assuming $P \neq NP$?

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Java program</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>A deterministic Turing machine with one head</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>A deterministic Turing machine with 3 tapes, each with 5 heads</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>A nondeterministic Turing machine with one head</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>A nondeterministic finite-state automaton (NFA)</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
(c) Which of the following languages are decidable?

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>0 { 0^n 1^n }</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{ \langle M \rangle \mid M \text{ is a Turing machine} }</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{ \langle M \rangle \mid M \text{ accepts } \langle M \rangle \cdot \langle M \rangle }</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{ \langle M \rangle \mid M \text{ accepts a finite number of non-palindromes} }</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{ \langle M \rangle \mid M \text{ accepts } \emptyset }</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{ \langle M, w \rangle \mid M \text{ accepts } w^R }</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{ \langle M, w \rangle \mid M \text{ accepts } w \text{ after at most }</td>
<td>w</td>
<td>^2 \text{ transitions} }</td>
</tr>
<tr>
<td>{ \langle M, w \rangle \mid M \text{ changes a blank on the tape to a non-blank, given input } w }</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{ \langle M, w \rangle \mid M \text{ changes a non-blank on the tape to a blank, given input } w }</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

(d) Let \( M \) be a standard Turing machine (with a single one-track tape and a single head) that decides the regular language \( 0^* 1^+ \). Which of the following must be true?

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given an empty initial tape, ( M ) eventually halts.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( M ) accepts the string ( 1111 ).</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( M ) rejects the string ( 0110 ).</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( M ) moves its head to the right at least once, given input ( 1100 ).</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( M ) moves its head to the right at least once, given input ( 0101 ).</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( M ) never accepts before reading a blank.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>For some input string, ( M ) moves its head to the left at least once.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>For some input string, ( M ) changes at least one symbol on the tape.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( M ) always halts.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>If ( M ) accepts a string ( w ), it does so after at most ( O(</td>
<td>w</td>
<td>^2) ) steps.</td>
</tr>
</tbody>
</table>
(e) Consider the following pair of languages:

- \( \text{HAMITLONIANPath} := \{ G \mid G \text{ contains a Hamiltonian path} \} \)
- \( \text{CONNECTED} := \{ G \mid G \text{ is connected} \} \)

Which of the following \textbf{must} be true, assuming \( P \neq NP \)?

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connected ( \in \text{NP} )</td>
<td></td>
</tr>
<tr>
<td>HamiltonianPath ( \in \text{NP} )</td>
<td></td>
</tr>
<tr>
<td>HamiltonianPath is decidable.</td>
<td></td>
</tr>
<tr>
<td>There is no polynomial-time reduction from HamiltonianPath to Connected.</td>
<td></td>
</tr>
<tr>
<td>There is no polynomial-time reduction from Connected to HamiltonianPath.</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ \text{ALWAYSHALTS} := \{ \langle M \rangle \mid M \text{ halts on every input string} \} \]

Rocket J. Squirrel suggests a reduction from the standard halting language

\[ \text{Halt} := \{ \langle M, w \rangle \mid M \text{ halts on inputs } w \} \].

Specifically, suppose there is a Turing machine \( AH \) that decides \( \text{ALWAYSHALTS} \). Rocky claims that the following Turing machine \( H \) decides \( \text{HALT} \). Given an arbitrary encoding \( \langle M, w \rangle \) as input, machine \( H \) writes the encoding \( \langle M' \rangle \) of a new Turing machine \( M' \) to the tape and passes it to \( AH \), where \( M' \) implements the following algorithm:

\[
\begin{align*}
M'(x): & \\
& \text{if } M \text{ accepts } w \\
& \quad \text{reject} \\
& \text{if } M \text{ rejects } w \\
& \quad \text{accept}
\end{align*}
\]

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 ) accepts ( w ), then ( M' ) halts on every input string.</td>
<td></td>
</tr>
<tr>
<td>If ( M ) diverges on ( w ), then ( M' ) halts on every input string.</td>
<td></td>
</tr>
<tr>
<td>If ( M ) accepts ( w ), then ( AH ) accepts ( \langle M' \rangle ).</td>
<td></td>
</tr>
<tr>
<td>If ( M ) rejects ( w ), then ( H ) rejects ( \langle M, w \rangle ).</td>
<td></td>
</tr>
<tr>
<td>( H ) decides the language ( \text{HALT} ). \text{(That is, Rocky's reduction is correct.)}</td>
<td></td>
</tr>
</tbody>
</table>
2. A relaxed 3-coloring of a graph $G$ assigns each vertex of $G$ one of three colors (for example, red, green, and blue), such that at most one edge in $G$ has both endpoints the same color.

(a) Give an example of a graph that has a relaxed 3-coloring, but does not have a proper 3-coloring (where every edge has endpoints of different colors).

(b) **Prove** that it is NP-hard to determine whether a given graph has a relaxed 3-coloring.
3. Give a complete, formal, self-contained description of a DFA that accepts all strings in \(\{0, 1\}^*\) containing at least ten 0s and at most ten 1s. Specifically:

(a) What are the states of your DFA?
(b) What is the start state of your DFA?
(c) What are the accepting states of your DFA?
(d) What is your DFA’s transition function?
4. Suppose you are given three strings \( A[1..n] \), \( B[1..n] \), and \( C[1..n] \). Describe and analyze an algorithm to find the maximum length of a common subsequence of all three strings. For example, given the input strings

\[
A = \text{AxxBxxCDxEF}, \quad B = \text{yyABCDyEyFy}, \quad C = \text{zAzzBCDzEFz},
\]

your algorithm should output the number 6, which is the length of the longest common subsequence \( \text{ABCDEF} \).
5. 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.

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

Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution. For example, given the maze shown below, your algorithm would return the number 8.

```
3 5 7 4 6
5 3 1 5 3
2 8 3 1 4
4 5 7 2 3
3 1 3 2 ★
```

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

**SelfReject** := \( \{ (M) \mid M \text{ rejects } (M) \} \)

**SelfAccept** := \( \{ (M) \mid M \text{ accepts } (M) \} \)

**SelfHalt** := \( \{ (M) \mid M \text{ halts on } (M) \} \)

**SelfDiverge** := \( \{ (M) \mid M \text{ does not halt on } (M) \} \)

**Reject** := \( \{ (M, w) \mid M \text{ rejects } w \} \)

**Accept** := \( \{ (M, w) \mid M \text{ accepts } w \} \)

**Halt** := \( \{ (M, w) \mid M \text{ halts on } w \} \)

**Diverge** := \( \{ (M, w) \mid M \text{ does not halt on } w \} \)

**NeverReject** := \( \{ (M) \mid \text{Reject}(M) = \emptyset \} \)

**NeverAccept** := \( \{ (M) \mid \text{Accept}(M) = \emptyset \} \)

**NeverHalt** := \( \{ (M) \mid \text{Halt}(M) = \emptyset \} \)

**NeverDiverge** := \( \{ (M) \mid \text{Diverge}(M) = \emptyset \} \)
“CS 374”: Algorithms and Models of Computation, Fall 2014
Final Exam (Version B) — December 16, 2014

Name:

NetID:

Section: 1 2 3

<table>
<thead>
<tr>
<th>#</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>20</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>70</td>
</tr>
</tbody>
</table>

Grader

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.

You have 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.

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

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

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”.

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 worth \( \frac{1}{2} \) point.

(a) Which of the following statements is true for every language \( L \subseteq \{0,1\}^* \)?

- \( L \) is non-empty.
- \( L \) is decidable or \( L \) is infinite (or both).
- \( L \) is accepted by some DFA with 42 states if and only if \( L \) is accepted by some NFA with 42 states.
- If \( L \) is regular, then \( L \in NP \).
- \( L \) is decidable if and only if its complement \( \overline{L} \) is undecidable.

(b) Which of the following computational models can simulate a deterministic Turing machine with three read/write heads, with at most polynomial slow-down in time, assuming \( P \neq NP \)?

- A C++ program
- A deterministic Turing machine with one head
- A deterministic Turing machine with 3 tapes, each with 5 heads
- A nondeterministic Turing machine with one head
- A nondeterministic finite-state automaton (NFA)
(c) Which of the following languages are decidable?

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{ww \mid w \text{ is a palindrome}}</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{\langle M \rangle \mid M \text{ is a Turing machine}}</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{\langle M \rangle \mid M \text{ accepts } \langle M \rangle \cdot \langle M \rangle}</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{\langle M \rangle \mid M \text{ accepts an infinite number of palindromes}}</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{\langle M \rangle \mid M \text{ accepts } \emptyset}</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{\langle M, w \rangle \mid M \text{ accepts } www}</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{\langle M, w \rangle \mid M \text{ accepts } w \text{ after at least }</td>
<td>w</td>
<td>^2 \text{ transitions}}</td>
</tr>
<tr>
<td>{\langle M, w \rangle \mid M \text{ changes a non-blank on the tape to a blank, given input } w}</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>{\langle M, w \rangle \mid M \text{ changes a blank on the tape to a non-blank, given input } w}</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

(d) Let \( M \) be a standard Turing machine (with a single one-track tape and a single head) such that \( \text{Accept}(M) \) is the regular language \( 0^*1^* \). Which of the following \textbf{must} be true?

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given an empty initial tape, ( M ) eventually halts.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( M ) accepts the string \textbf{1111}.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( M ) rejects the string \textbf{0110}.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( M ) moves its head to the right at least once, given input \textbf{1100}.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( M ) moves its head to the right at least once, given input \textbf{0101}.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( M ) must read a blank before it accepts.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>For some input string, ( M ) moves its head to the left at least once.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>For some input string, ( M ) changes at least one symbol on the tape.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( M ) always halts.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>If ( M ) accepts a string ( w ), it does so after at most ( O(</td>
<td>w</td>
<td>^2) ) steps.</td>
</tr>
</tbody>
</table>
(e) Consider the following pair of languages:
   - \textsc{HamiltonianPath} := \{G \mid G \text{ contains a Hamiltonian path}\}
   - \textsc{Connected} := \{G \mid G \text{ is connected}\}

Which of the following \textbf{must} be true, assuming P \neq NP?

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connected \in NP</td>
<td></td>
</tr>
<tr>
<td>HamiltonianPath \in NP</td>
<td></td>
</tr>
<tr>
<td>HamiltonianPath is undecidable.</td>
<td></td>
</tr>
<tr>
<td>There is a polynomial-time reduction from HamiltonianPath to Connected.</td>
<td></td>
</tr>
<tr>
<td>There is a polynomial-time reduction from Connected to HamiltonianPath.</td>
<td></td>
</tr>
</tbody>
</table>

(f) Suppose we want to prove that the following language is undecidable.
\[
\textsc{AlwaysHalts} := \{\langle M \rangle \mid M \text{ halts on every input string}\}
\]

Bullwinkle J. Moose suggests a reduction from the standard halting language
\[
\textsc{Halt} := \{\langle M, w \rangle \mid M \text{ halts on inputs } w\}.
\]

Specifically, suppose there is a Turing machine \textsc{AH} that decides \textsc{AlwaysHalts}.
Bullwinkle claims that the following Turing machine \textsc{H} decides \textsc{Halt}.
Given an arbitrary encoding \langle M, w \rangle as input, machine \textsc{H} writes the encoding \langle M' \rangle of a new
Turing machine \textsc{M'} to the tape and passes it to \textsc{AH}, where \textsc{M'} implements the following algorithm:

\[
\begin{align*}
\text{M'}(x): & \\
& \text{if } M \text{ accepts } w \text{ reject} \\
& \text{if } M \text{ rejects } w \text{ accept}
\end{align*}
\]

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 \textsc{M} accepts \textit{w}, then \textsc{M'} halts on every input string.</td>
<td></td>
</tr>
<tr>
<td>If \textsc{M} rejects \textit{w}, then \textsc{M'} halts on every input string.</td>
<td></td>
</tr>
<tr>
<td>If \textsc{M} rejects \textit{w}, then \textsc{H} rejects \langle M, w \rangle.</td>
<td></td>
</tr>
<tr>
<td>If \textsc{M} diverges on \textit{w}, then \textsc{H} diverges on \langle M, w \rangle.</td>
<td></td>
</tr>
<tr>
<td>\textsc{H} does not correctly decide the language \textsc{Halt}. (That is, Bullwinkle’s reduction is incorrect.)</td>
<td></td>
</tr>
</tbody>
</table>
2. A near-Hamiltonian cycle in a graph $G$ is a closed walk in $G$ that visits one vertex exactly twice and every other vertex exactly once.

(a) Give an example of a graph that contains a near-Hamiltonian cycle, but does not contain a Hamiltonian cycle (which visits every vertex exactly once).

(b) Prove that it is NP-hard to determine whether a given graph contains a near-Hamiltonian cycle.
3. Give a complete, formal, self-contained description of a DFA that accepts all strings in \{0, 1\}^* such that every fifth bit is 0 and the length is not divisible by 12. For example, your DFA should accept the strings 1110111101 and 11. Specifically:

(a) What are the states of your DFA?
(b) What is the start state of your DFA?
(c) What are the accepting states of your DFA?
(d) What is your DFA’s transition function?
4. Suppose you are given three strings $A[1..n]$, $B[1..n]$, and $C[1..n]$. Describe and analyze an algorithm to find the maximum length of a common subsequence of all three strings. For example, given the input strings

$$A = AxxBCDxExF, \quad B = yyABCdyEyFy, \quad C = zAzBCDzEFz,$$

your algorithm should output the number 6, which is the length of the longest common subsequence $ABCDEF$. 
5. 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.

(a) $\{www \mid w \in \Sigma^*\}$

(b) $\{wxw \mid w, x \in \Sigma^*\}$
6. A **number maze** is an \( n \times n \) grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner.

- On each turn, you are allowed to move the token up, down, left, or right.
- The distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three steps down, three steps left, or three steps right.
- However, you are never allowed to move the token off the edge of the board. In particular, if the current number is too large, you may not be able to move at all.

Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution. For example, given the maze shown below, your algorithm would return the number 8.

![3x5 grid](image1.png)

A 5 \( \times \) 5 number maze that can be solved in eight moves.
(scratch paper)
You may assume the following problems are NP-hard:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CircuitSat</td>
<td>Given a boolean circuit, are there any input values that make the circuit output True?</td>
</tr>
<tr>
<td>3Sat</td>
<td>Given a boolean formula in conjunctive normal form, with exactly three literals per clause, does the formula have a satisfying assignment?</td>
</tr>
<tr>
<td>MaxIndependentSet</td>
<td>Given an undirected graph $G$, what is the size of the largest subset of vertices in $G$ that have no edges among them?</td>
</tr>
<tr>
<td>MaxClique</td>
<td>Given an undirected graph $G$, what is the size of the largest complete subgraph of $G$?</td>
</tr>
<tr>
<td>MinVertexCover</td>
<td>Given an undirected graph $G$, what is the size of the smallest subset of vertices that touch every edge in $G$?</td>
</tr>
<tr>
<td>3Color</td>
<td>Given an undirected graph $G$, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?</td>
</tr>
<tr>
<td>HamiltonianPath</td>
<td>Given an undirected graph $G$, is there a path in $G$ that visits every vertex exactly once?</td>
</tr>
<tr>
<td>HamiltonianCycle</td>
<td>Given an undirected graph $G$, is there a cycle in $G$ that visits every vertex exactly once?</td>
</tr>
<tr>
<td>DirectedHamiltonianCycle</td>
<td>Given an directed graph $G$, is there a directed cycle in $G$ that visits every vertex exactly once?</td>
</tr>
<tr>
<td>TravelingSalesman</td>
<td>Given a graph $G$ (either directed or undirected) with weighted edges, what is the minimum total weight of any Hamiltonian path/cycle in $G$?</td>
</tr>
<tr>
<td>Draughts</td>
<td>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?</td>
</tr>
<tr>
<td>Super Mario</td>
<td>Given an $n \times n$ level for Super Mario Brothers, can Mario reach the castle?</td>
</tr>
</tbody>
</table>

You may assume the following languages are undecidable:

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelfReject := ${M \mid M$ rejects $\langle M \rangle}$</td>
<td></td>
</tr>
<tr>
<td>SelfAccept := ${M \mid M$ accepts $\langle M \rangle}$</td>
<td></td>
</tr>
<tr>
<td>SelfHalt := ${M \mid M$ halts on $\langle M \rangle}$</td>
<td></td>
</tr>
<tr>
<td>SelfDiverge := ${M \mid M$ does not halt on $\langle M \rangle}$</td>
<td></td>
</tr>
<tr>
<td>Reject := ${M,w \mid M$ rejects $w}$</td>
<td></td>
</tr>
<tr>
<td>Accept := ${M,w \mid M$ accepts $w}$</td>
<td></td>
</tr>
<tr>
<td>Halt := ${M,w \mid M$ halts on $w}$</td>
<td></td>
</tr>
<tr>
<td>Diverge := ${M,w \mid M$ does not halt on $w}$</td>
<td></td>
</tr>
<tr>
<td>NeverReject := ${M \mid$ Reject$(M) = \emptyset}$</td>
<td></td>
</tr>
<tr>
<td>NeverAccept := ${M \mid$ Accept$(M) = \emptyset}$</td>
<td></td>
</tr>
<tr>
<td>NeverHalt := ${M \mid$ Halt$(M) = \emptyset}$</td>
<td></td>
</tr>
<tr>
<td>NeverDiverge := ${M \mid$ Diverge$(M) = \emptyset}$</td>
<td></td>
</tr>
</tbody>
</table>