CS/ECE 374 A ✧ Spring 2018

☙ Homework 0 ❧

Due Tuesday, January 23, 2018 at 8pm

- **Each student must submit individual solutions for this homework.** For all future homeworks, groups of up to three students can submit joint solutions.
- **Submit your solutions electronically to Gradescope as PDF files.** Submit a separate PDF file for each numbered problem. If you plan to typeset your solutions, please use the LaTeX solution template on the course web site. If you must submit scanned handwritten solutions, please use a black pen on blank white paper and a high-quality scanner app (or an actual scanner).
- You are ***not*** required to sign up on Gradescope or Piazza with your real name and your illinois.edu email address; you may use any email address and alias of your choice. However, to give you credit for the homework, we need to know who Gradescope thinks you are. **Please fill out the web form linked from the course web page.**

☞ Some important course policies ☜

- **You may use any source at your disposal**—paper, electronic, or human—but you ***must*** cite ***every*** source that you use, and you ***must*** write everything yourself in your own words. See the academic integrity policies on the course web site for more details.
- The answer ***"I don't know"*** (and *nothing* else) is worth 25% partial credit on any required problem or subproblem on any homework or exam. We will accept synonyms like "No idea" or "WTF" or "`\(ó_ò)/`", but you must write *something*.

  On the other hand, only the homework problems you submit actually contribute to your overall course grade, so submitting "I don't know" for an entire numbered homework problem will almost certainly hurt your grade more than submitting nothing at all.
- **Avoid the Three Deadly Sins!** Any homework or exam solution that breaks any of the following rules will be given an ***automatic zero***, unless the solution is otherwise perfect. Yes, we really mean it. We're not trying to be scary or petty (Honest!), but we do want to break a few common bad habits that seriously impede mastery of the course material.
  - Always give complete solutions, not just examples.
  - Always declare all your variables, in English. In particular, always describe the specific problem your algorithm is supposed to solve.
  - Never use weak induction.

See the course web site for more information.

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

---

1. The famous Basque computational arborist Gorka Oihanean has a favorite 26-node binary tree, in which each node is labeled with a letter of the alphabet. Intorder and postorder traversals of his tree visits the nodes in the following orders:

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

   (a) List the nodes in Professor Oihanean's tree according to a preorder traversal.
   (b) Draw Professor Oihanean's tree.

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

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

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

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

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

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

   You may assume without proof that $|x \bullet y| = |x| + |y|$, or any other result proved in class, in lab, or in the lecture notes. Otherwise, your proofs must be formal and self-contained, and they must invoke the *formal* definitions of length $|w|$, concatenation $\bullet$, and the *swap* function. Do not appeal to intuition!

3. Consider the set of strings $L \subseteq \{0,1\}^*$ defined recursively as follows:

   - The empty string ε is in L.
   - For any string x in L, the string $0x$ is also in L.
   - For any strings x and y in L, the string $1x1y$ is also in L.
   - These are the only strings in L.

   (a) Prove that the string 101110101101011 is in L.
   (b) Prove that every string $w \in L$ contains an even number of 1s.
   (c) Prove that every string $w \in \{0,1\}^*$ with an even number of 1s is a member of L.

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

$$\#(0, 101110101101011) = 5 \quad \text{and} \quad \#(1, 101110101101011) = 10.$$

   You may assume without proof that $\#(a,uv) = \#(a,u) + \#(a,v)$ for any symbol a and any strings u and v, or any other result proved in class, in lab, or in the lecture notes. Otherwise, your proofs must be formal and self-contained.

Each homework assignment will include at least one solved problem, similar to the problems assigned in that homework, together with the grading rubric we would apply *if* this problem appeared on a homework or exam. These model solutions illustrate our recommendations for structure, presentation, and level of detail in your homework solutions. Of course, the actual *content* of your solutions won't match the model solutions, because your problems are different!

Solved Problems

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

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

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

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

(b) Prove $w = w^R$ for every palindrome w (according to your recursive definition).

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

You may assume without proof the following statements for all strings x, y, and z:

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

Solution:

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

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

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

(b) Let w be an arbitrary palindrome.
Assume that $x = x^R$ for every palindrome x such that $|x| < |w|$.
There are three cases to consider (mirroring the definition of "palindrome"):

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

$x \in P$. In this case, we have

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

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

Rubric: 4 points: standard induction rubric (scaled)

(c) Let w be an arbitrary string such that $w = w^R$.
Assume that every string x such that $|x| < |w|$ and $x = x^R$ is a palindrome.
There are three cases to consider (mirroring the definition of "palindrome"):

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

  [At this point, we need to prove that $a = b$ and that y is a palindrome.]

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

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

  We conclude that w is a palindrome by definition.

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

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

Standard induction rubric. For problems worth 10 points:

- \+ 1 for explicitly considering an *arbitrary* object.
- \+ 2 for a valid ***strong*** induction hypothesis
  - – **Deadly Sin!** Automatic zero for stating a weak induction hypothesis, unless the rest of the proof is *absolutely perfect*.
- \+ 2 for explicit exhaustive case analysis
  - – No credit here if the case analysis omits an infinite number of objects. (For example: all odd-length palindromes.)
  - – −1 if the case analysis omits an finite number of objects. (For example: the empty string.)
  - – −1 for making the reader infer the case conditions. Spell them out!
  - – No penalty if the cases overlap (for example: even length at least 2, odd length at least 3, and length at most 5.)
- \+ 1 for cases that do not invoke the inductive hypothesis (“base cases”)
  - – No credit here if one or more “base cases” are missing.
- \+ 2 for correctly applying the ***stated*** inductive hypothesis
  - – No credit here for applying a ***different*** inductive hypothesis, even if that different inductive hypothesis would be valid.
- \+ 2 for other details in cases that invoke the inductive hypothesis (“inductive cases”)
  - – No credit here if one or more “inductive cases” are missing.

For (sub)problems worth less than 10 points, scale and round to the nearest half-integer.

CS/ECE 374 A ✧ Spring 2018

Homework 1

Due Tuesday, January 30, 2018 at 8pm

Starting with this homework, groups of up to three people can submit joint solutions. Each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the Gradescope names and email addresses of each group member. In addition, whoever submits the homework must tell Gradescope who their other group members are.

---

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

   (a) All strings except `001`.

   (b) All strings that end with the suffix `001001`.

   (c) All strings that contain the substring `001`.

   (d) All strings that contain the subsequence `001`.

   (e) All strings that do not contain the substring `001`.

   (f) All strings that do not contain the subsequence `001`.

2. Let L denote the set of all strings in $\{0, 1\}^*$ that contain all four strings `00`, `01`, `10`, and `11` as substrings. For example, the strings `110011` and `01001011101001` are in L, but the strings `00111` and `1010101` are not.

   Formally describe a DFA with input alphabet $\Sigma = \{0, 1\}$ that accepts the language L, by explicitly describing the states Q, the start state s, the accept states A, and the transition function δ. Do not attempt to *draw* your DFA; the smallest DFA for this language has 20 states, which is too many for a drawing to be understandable.

   Argue that your machine accepts every string in L and nothing else, by explaining what each state in your DFA *means*. Formal descriptions without English explanations will receive no credit, even if they are correct. (See the standard DFA rubric for more details.)

   This is an exercise in clear communication. We are not only asking you to design a *correct* DFA. We are also asking you to clearly, precisely, and convincingly explain your DFA to another human being who understands DFAs but has *not* thought about this particular problem. Excessive formality and excessive brevity will hurt you just as much as imprecision and handwaving.

3. Let L be the set of all strings in $\{0,1\}^*$ that contain *exactly one* occurrence of the substring 010.

   (a) Give a regular expression for L, and briefly argue why your expression is correct. *[Hint: You may find the shorthand notation $A^+ = AA^*$ useful.]*

   (b) Describe a DFA over the alphabet $\Sigma = \{0,1\}$ that accepts the language L.

   Argue that your machine accepts every string in L and nothing else, by explaining what each state in your DFA *means*. You may either draw the DFA or describe it formally, but the states Q, the start state s, the accepting states A, and the transition function δ must be clearly specified. Drawings or formal descriptions without English explanations will receive no credit, even if they are correct.

Solved problem

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

   - Line comments: Strings of the form $//\cdots$↲
   - Block comments: Strings of the form $/\star\cdots\star/$

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

   $$/\star\star\star/ \qquad //\diamond//\diamond\text{↲} \qquad /\star///\diamond\star\diamond\text{↲}\star\star/ \qquad /\star\diamond//\diamond\text{↲}\diamond\star/$$

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

   $$/\star/ \qquad //\diamond//\diamond\text{↲}\diamond\text{↲} \qquad /\star\diamond/\star\diamond\star/\diamond\star/$$

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

   Solution:

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

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

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

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

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

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

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

Rubric: Standard regular expression rubric

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

Solution:

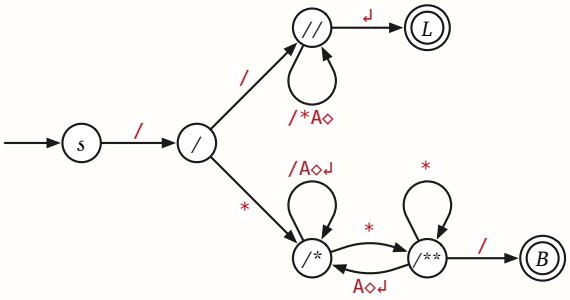
$$\big(\diamond + \text{↲} \;+\; //(/ + * + A + \diamond)^* \text{↲} + /*(/ + A + \diamond + \text{↲} + **^*(A + \diamond + \text{↲}))^* **^* / \big)^*$$

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

Rubric: Standard regular expression rubric

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

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



The states are labeled mnemonically as follows:

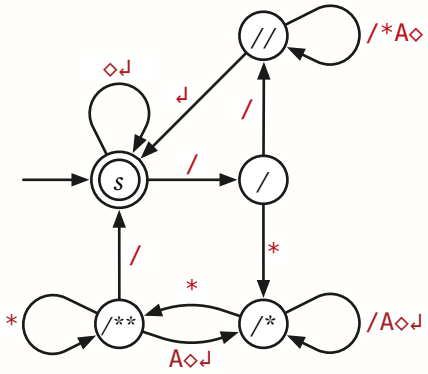
- s — We have not read anything.
- / — We just read the initial /.
- // — We are reading a line comment.
- L — We have just read a complete line comment.
- /* — We are reading a block comment, and we did not just read a * after the opening /*.
- /** — We are reading a block comment, and we just read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

■

Rubric: Standard DFA design rubric

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

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



The states are labeled mnemonically as follows:

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

■

Rubric: Standard DFA design rubric

Standard regular expression rubric. For problems worth 10 points:

- 2 points for a syntactically correct regular expression.
- **Homework only:** 4 points for a *brief* English explanation of your regular expression. This is how you argue that your regular expression is correct.
  - **Deadly Sin ("Declare your variables."): No credit for the problem if the English explanation is missing, *even if the regular expression is correct*.**
  - For longer expressions, you should explain each of the major components of your expression, and separately explain how those components fit together.
  - We do not want a *transciption*; don't just translate the regular-expression notation into English.
- 4 points for correctness. (8 points on exams, with all penalties doubled)
  - -1 for a single mistake: one typo, excluding exactly one string in the target language, or including exactly one string not in the target language.
  - -2 for incorrectly including/excluding more than one but a finite number of strings.
  - -4 for incorrectly including/excluding an infinite number of strings.
- Regular expressions that are longer than necessary may be penalized. Regular expressions that are *significantly* longer than necessary may get no credit at all.

Standard DFA design rubric. For problems worth 10 points:

- 2 points for an unambiguous description of a DFA, including the states set Q, the start state s, the accepting states A, and the transition function δ.
  - **For drawings:** Use an arrow from nowhere to indicate s, and doubled circles to indicate accepting states A. If $A = \emptyset$, say so explicitly. If your drawing omits a reject state, say so explicitly. **Draw neatly!** If we can't read your solution, we can't give you credit for it,.
  - **For text descriptions:** You can describe the transition function either using a 2d array, using mathematical notation, or using an algorithm.
  - **For product constructions:** You must give a complete description of the states and transition functions of the DFAs you are combining (as either drawings or text), together with the accepting states of the product DFA.
- **Homework only:** 4 points for *briefly* and correctly explaining the purpose of each state *in English*. This is how you justify that your DFA is correct.
  - **Deadly Sin ("Declare your variables."): No credit for the problem if the English description is missing, *even if the DFA is correct*.**
  - For product constructions, explaining the states in the factor DFAs is sufficient.
- 4 points for correctness. (8 points on exams, with all penalties doubled)
  - -1 for a single mistake: a single misdirected transition, a single missing or extra accept state, rejecting exactly one string that should be accepted, or accepting exactly one string that should be accepted.
  - -2 for incorrectly accepting/rejecting more than one but a finite number of strings.
  - -4 for incorrectly accepting/rejecting an infinite number of strings.
- DFA drawings with too many states may be penalized. DFA drawings with *significantly* too many states may get no credit at all.
- Half credit for describing an NFA when the problem asks for a DFA.

CS/ECE 374 A ✧ Spring 2018

Homework 2

Due Tuesday, February 6, 2018 at 8pm

---

1. Prove that the following languages are *not* regular.

   (a) $\{\mathtt{0}^a\mathtt{1}\,\mathtt{0}^b\mathtt{1}\,\mathtt{0}^c \mid a + b = c\}$

   (b) $\{w \in (\mathtt{0}+\mathtt{1})^* \mid \#(\mathtt{0}, w) \le 2 \cdot \#(\mathtt{1}, w)\}$

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

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

2. For each of the following regular expressions, describe or draw two finite-state machines:

   - An NFA that accepts the same language, constructed from the given regular expression using Thompson's algorithm (described in class and in the notes).
   - An equivalent DFA, constructed from your NFA using the incremental subset algorithm (described in class and in the notes). For each state in your DFA, identify the corresponding subset of states in your NFA. Your DFA should have no unreachable states.

   (a) $(\mathtt{0}+\mathtt{1})^* \cdot \mathtt{0001} \cdot (\mathtt{0}+\mathtt{1})^*$

   (b) $(\mathtt{1} + \mathtt{01} + \mathtt{001})^*\mathtt{0}^*$

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

   (a) Strings in which the substrings $\mathtt{00}$ and $\mathtt{11}$ do not appear the same number of times. For example, $\mathtt{1100011} \notin L$ because both substrings appear twice, but $\mathtt{01000011} \in L$.

   (b) Strings in which the substrings $\mathtt{01}$ and $\mathtt{10}$ do not appear the same number of times. For example, $\mathtt{1100011} \notin L$ because both substrings appear twice, but $\mathtt{01000011} \in L$.

   (c) $\{wxw \mid w, x \in \Sigma^*\}$

   (d) $\{wxw \mid w, x \in \Sigma^+\}$

   [Hint: Exactly two of these languages are regular. Strings can be empty.]

Solved problem

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

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

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

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

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

   - If x contains a 0, then we can write $w = 01^n0y$ for some integer n and some string y; but this is impossible, because the prefix 01^n0 is a palindrome of length at least 2.
   - Otherwise, $x = 1^n$ for some integer n. Every prefix of w has the form 01^m for some integer $m \le n$. Any palindrome that starts with 0 must end with 0, so the only palindrome prefixes of w are ε and 0, both of which have length less than 2.

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

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

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

   Solution: Not regular. Call this language L_b.

   I claim that the infinite language $F = (012)^+$ is a fooling set for L_b.

   Let x and y be arbitrary distinct strings in F.

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

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

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

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

   We conclude that F is a fooling set for L_b, as claimed.

   Because F is infinite, L_b cannot be regular. ■

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

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

Solution: Not regular. Call this language L_c.
I claim that the infinite language $F = (001101)^+$ is a fooling set for L_c.
Let x and y be arbitrary distinct strings in F.
Then $x = (001101)^i$ and $y = (001101)^j$ for some positive integers $i \neq j$.
Without loss of generality, assume $i < j$.
Let z be the suffix $(101100)^i$.

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

We conclude that F is a fooling set for L_c, as claimed.
Because F is infinite, L_c cannot be regular. ■

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

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

Solution: Regular. Call this language L_d.

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

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

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

Solution: Regular. Call this language L_d.

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

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

■

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

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

Standard fooling set rubric. For problems worth 5 points:

- 2 points for the fooling set:
  - + 1 for explicitly describing the proposed fooling set F.
  - + 1 if the proposed set F is actually a fooling set for the target language.
  - — No credit for the proof if the proposed set is not a fooling set.
  - — No credit for the *problem* if the proposed set is finite.
- 3 points for the proof:
  - ◦ The proof must correctly consider *arbitrary* strings $x, y \in F$.
    - — No credit for the proof unless both x and y are *always* in F.
    - — No credit for the proof unless x and y can be *any* strings in F.
  - + 1 for correctly describing a suffix z that distinguishes x and y.
  - + 1 for proving either $xz \in L$ or $yz \in L$.
  - + 1 for proving either $yz \notin L$ or $xz \notin L$, respectively.

As usual, scale partial credit (rounded to nearest ½) for problems worth fewer points.

CS/ECE 374 A ✧ Spring 2018

ᔕ Homework 3 ᔓ

Due Tuesday, February 13, 2018 at 8pm

---

1. Describe context-free grammars for the following languages over the alphabet $\Sigma = \{0, 1\}$. For each non-terminal in your grammars, describe in English the language generated by that non-terminal.

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

   (b) $\left\{w \in (0+1)^* \;\middle|\; \#(0,w) \le 2 \cdot \#(1,w)\right\}$

   (c) Strings in which the substrings 00 and 11 appear the same number of times. For example, $1100011 \in L$ because both substrings appear twice, but $01000011 \notin L$. *[Hint: This is the complement of the language you considered in HW2.]*

2. Let $inc : \{0,1\}^* \to \{0,1\}^*$ denote the *increment* function, which transforms the binary representation of an arbitrary integer n into the binary representation of $n+1$, truncated to the same number of bits. For example:

   $$inc(0010) = 0011 \qquad inc(0111) = 1000 \qquad inc(1111) = 0000 \qquad inc(\varepsilon) = \varepsilon$$

   Let $L \subseteq \{0,1\}^*$ be an arbitrary regular language. Prove that $inc(L) = \{inc(w) \mid w \in L\}$ is also regular.

3. A ***shuffle*** of two strings x and y is any string obtained by interleaving the symbols in x and y, but keeping them in the same order. For example, the following strings are shuffles of HOGWARTS and BRAKEBILLS:

   HOGWARTSBRAKEBILLS HOGBRAKEWARTSBILLS BHROAGKWEABRITLSLS

   More formally, a string z is a shuffle of strings x and y if and only if (at least) one of the following conditions holds:

   - $x = \varepsilon$ and $z = y$
   - $y = \varepsilon$ and $z = x$
   - $x = ax'$ and $z = az'$ where z' is a shuffle of x' and y
   - $y = ay'$ and $z = az'$ where z' is a shuffle of x and y'

   For any two languages L and L' over the alphabet $\{0,1\}$, define

   $$shuffles(L, L') = \left\{z \in \{0,1\}^* \;\middle|\; z \text{ is a shuffle of some } x \in L \text{ and } y \in L'\right\}$$

   Prove that if L and L' are regular languages, then $shuffles(L, L')$ is also a regular language.

Solved problem

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

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

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

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

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

■

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

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

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

Solution: Consider the regular language $L = \mathtt{0}^*\mathtt{1}$.
Expanding the regular expression lets us rewrite $L = \{\mathtt{0}^n\mathtt{1} \mid n \geq 0\}$. It follows that $\mathit{double}(L) = \{\mathtt{0}^n\mathtt{1}\mathtt{0}^n\mathtt{1} \mid n \geq 0\}$. I claim that this language is not regular.
Let x and y be arbitrary distinct strings in L.
Then $x = \mathtt{0}^i\mathtt{1}$ and $y = \mathtt{0}^j\mathtt{1}$ for some integers $i \neq j$.
Then x is a distinguishing suffix of these two strings, because

- $xx \in \mathit{double}(L)$ by definition, but
- $yx = \mathtt{0}^i\mathtt{1}\mathtt{0}^j\mathtt{1} \notin \mathit{double}(L)$ because $i \neq j$.

We conclude that L is a fooling set for $\mathit{double}(L)$.
Because L is infinite, $\mathit{double}(L)$ cannot be regular. ■

Solution: Consider the regular language $L = \Sigma^* = (\mathtt{0} + \mathtt{1})^*$.
I claim that the language $\mathit{double}(\Sigma^*) = \{ww \mid w \in \Sigma^*\}$ is not regular.
Let F be the infinite language $\mathtt{0}\mathtt{1}^*\mathtt{0}$.
Let x and y be arbitrary distinct strings in F.
Then $x = \mathtt{0}\mathtt{1}^i\mathtt{0}$ and $y = \mathtt{0}\mathtt{1}^j\mathtt{0}$ for some integers $i \neq j$.
The string $z = \mathtt{1}^i$ is a distinguishing suffix of these two strings, because

- $xz = \mathtt{0}\mathtt{1}^i\mathtt{0}\mathtt{1}^i = ww$ where $w = \mathtt{0}\mathtt{1}^i$, so $xz \in \mathit{double}(\Sigma^*)$, but
- $yx = \mathtt{0}\mathtt{1}^j\mathtt{0}\mathtt{1}^i \notin \mathit{double}(\Sigma^*)$ because $i \neq j$.

We conclude that F is a fooling set for $\mathit{double}(\Sigma^*)$.
Because F is infinite, $\mathit{double}(\Sigma^*)$ cannot be regular. ■

Rubric: 5 points:

- 2 points for describing a regular language L such that $\mathit{double}(L)$ is not regular.
- 1 point for describing an infinite fooling set for $\mathit{double}(L)$:
  - + ½ for explicitly describing the proposed fooling set F.
  - + ½ if the proposed set F is actually a fooling set.
  - – No credit for the proof if the proposed set is not a fooling set.
  - – No credit for the *problem* if the proposed set is finite.
- 2 points for the proof:
  - + ½ for correctly considering *arbitrary* strings x and y
    - – No credit for the proof unless both x and y are *always* in F.
    - – No credit for the proof unless both x and y can be *any* string in F.
  - + ½ for correctly stating a suffix z that distinguishes x and y.
  - + ½ for proving either $xz \in L$ or $yz \in L$.
  - + ½ for proving either $yz \notin L$ or $xz \notin L$, respectively.

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

Standard langage transformation rubric. For problems worth 10 points:

- \+ 2 for a formal, complete, and unambiguous description of the output automaton, including the states, the start state, the accepting states, and the transition function, as functions of an *arbitrary* input DFA. The description must state whether the output automaton is a DFA, an NFA without ε-transitions, or an NFA with ε-transitions.
  - No points for the rest of the problem if this is missing.
- \+ 2 for a *brief* English explanation of the output automaton. We explicitly do *not* want a formal proof of correctness, or an English *transcription*, but a few sentences explaining how your machine works and justifying its correctness. What is the overall idea? What do the states represent? What is the transition function doing? Why these accepting states?
  - **Deadly Sin:** No points for the rest of the problem if this is missing.
- \+ 6 for correctness
  - \+ 3 for accepting *all* strings in the target language
  - \+ 3 for accepting *only* strings in the target language
  - – 1 for a single mistake in the formal description (for example a typo)
  - Double-check correctness when the input language is $\emptyset$, or $\{\varepsilon\}$, or 0^*, or Σ^*.

CS/ECE 374 A ✧ Spring 2018

☙ Homework 4 ❧

Due Tuesday, February 27, 2018 at 8pm

---

1. At the end of the second act of the action blockbuster *Fast and Impossible XIII¾: Guardians of Expendable Justice Reloaded*, the villainous Dr. Metaphor hypnotizes the entire Hero League/Force/Squad, arranges them in a long line at the edge of a cliff, and instructs each hero to shoot the closest taller heroes to their left and right, at a prearranged signal.

   Suppose we are given the heights of all n heroes, in clockwise order around the circle, in an array $Ht[1..n]$. (To avoid salary arguments, the producers insisted that no two heroes have the same height.) Then we can compute the Left and Right targets of each hero in $O(n^2)$ time using the following algorithm.

```
WhoTargetsWhom(Ht[1..n]):
  for j ← 1 to n
      ⟨⟨Find the left target L[j] for hero j⟩⟩
      L[j] ← None
      for i ← 1 to j − 1
          if Ht[i] > Ht[j]
              L[j] ← i

      ⟨⟨Find the right target R[j] for hero j⟩⟩
      R[j] ← None
      for k ← n down to j + 1
          if Ht[k] > Ht[j]
              R[j] ← k

  return L[1..n], R[1..n]
```

   (a) Describe a divide-and-conquer algorithm that computes the output of WhoTargetsWhom in $O(n \log n)$ time.

   (b) Prove that at least $\lfloor n/2 \rfloor$ of the n heroes are targets. That is, prove that the output arrays $R[0..n-1]$ and $L[0..n-1]$ contain at least $\lfloor n/2 \rfloor$ distinct values (other than None).

   (c) Alas, Dr. Metaphor's diabolical plan is successful. At the prearranged signal, all the heroes simultaneously shoot their targets, and all targets fall over the cliff, apparently dead. Metaphor repeats his dastardly experiment over and over; after each massacre, he forces the remaining heroes to choose new targets, following the same algorithm, and then shoot their targets at the next signal. Eventually, only the shortest member of the Hero Crew/Alliance/Posse is left alive.[1]

   Describe an algorithm that computes the number of rounds before Dr. Metaphor's deadly process finally ends. For full credit, your algorithm should run in $O(n)$ time.

[1] In the thrilling final act, Retcon the Squirrel, the last surviving member of the Hero Team/Group/Society, saves everyone by traveling back in time and retroactively replacing the other $n-1$ heroes with lifelike balloon sculptures.

2. Describe and analyze a recursive algorithm to reconstruct an arbitrary binary tree, given its preorder and inorder node sequences as input.

   The input to your algorithm is a pair of arrays $Pre[1..n]$ and $In[1..n]$, each containing a permutation of the same set of n distinct symbols. Your algorithm should return an n-node binary tree whose nodes are labeled with those n symbols (or an error code if no binary tree is consistent with the input arrays). You solved an instance of this problem in Homework 0.

3. Suppose we are given a set S of n items, each with a *value* and a *weight*. For any element $x \in S$, we define two subsets:

   - $S_{<x}$ is the set of all elements of S whose value is smaller than the value of x.
   - $S_{>x}$ is the set of all elements of S whose value is larger than the value of x.

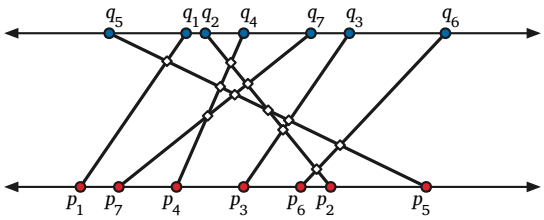
   For any subset $R \subseteq S$, let $w(R)$ denote the sum of the weights of elements in R. The ***weighted median*** of R is any element x such that $w(S_{<x}) \leq w(S)/2$ and $w(S_{>x}) \leq w(S)/2$.

   Describe and analyze an algorithm to compute the weighted median of a given weighted set in $O(n)$ time. Your input consists of two unsorted arrays $S[1..n]$ and $W[1..n]$, where for each index i, the ith element has value $S[i]$ and weight $W[i]$. You may assume that all values are distinct and all weights are positive.

   [Hint: Use or modify the linear-time selection algorithm described in class on Thursday.]

Solved problem

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



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

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

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

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

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

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

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

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

Rubric: This is enough for full credit.

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

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

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

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

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

Rubric: 10 points = 2 for base case + 3 for divide (split and recurse) + 3 for conquer (merge and count) + 2 for time analysis. Max 3 points for a correct $O(n^2)$-time algorithm. This is neither the only way to correctly describe this algorithm nor the only correct $O(n \log n)$-time algorithm. No proof of correctness is required.

Notice that each boxed algorithm is preceded by an English description of the task that algorithm performs. **Omitting these descriptions is a Deadly Sin.**

CS/ECE 374 A ✧ Spring 2018

Homework 5

Due Tuesday, March 6, 2018 at 8pm

---

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 list of n songs that the judges will play during the contest, in chronological order.

   You know all the songs, all the judges, and your own dancing ability extremely well. For each integer k, you know that if you dance to the kth song on the schedule, you will be awarded exactly $Score[k]$ points, but then you will be physically unable to dance for the next $Wait[k]$ songs (that is, you cannot dance to songs $k+1$ through $k+Wait[k]$). The dancer with the highest total score at the end of the night wins the contest, so you want your total score to be as high as possible.

   Describe and analyze an efficient algorithm to compute the maximum total score you can achieve. The input to your sweet algorithm is the pair of arrays $Score[1..n]$ and $Wait[1..n]$.

2. Suppose you are given a NFA $M = (\{0,1\}, Q, s, A, \delta)$ and a binary string $w \in \{0,1\}^*$. Describe and analyze an efficient algorithm to determine whether M accepts w. Concretely, the input NFA M is represented as follows:

   - $Q = \{1, 2, \ldots, k\}$ for some integer k.
   - The start state s is state 1.
   - Accepting states are indicated by a boolean array $A[1..k]$, where $A[q] =$ TRUE if and only if $q \in A$
   - The transition function δ is represented by a boolean array $inDelta[1..k, 0..1, 1..k]$, where $inDelta[p, a, q] =$ TRUE if and only if $q \in \delta(p, a)$.

   Finally, the input string is given as an array $w[1..n]$. Your algorithm should return TRUE if M accepts w, and FALSE if M does not accept w. Report the running time of your algorithm as a function of k (the number of states in M) and n (the length of w). *[Hint: Do not convert M to a DFA!!]*

3. Recall that a ***palindrome*** is any string that is exactly the same as its reversal, like the empty srting, or `I`, or `DEED`, or `RACECAR`, or `AMANAPLANACATACANALPANAMA`.

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

   `BUB` • `BASEESAB` • `ANANA`

   `B` • `U` • `BB` • `ASEESA` • `B` • `ANANA`

   `BUB` • `B` • `A` • `SEES` • `ABA` • `N` • `ANA`

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

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

   (a) Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example:

   - Given the string `PALINDROME`, your algorithm should return the integer 10.
   - Given the string `BUBBASEESABANANA`, your algorithm should return the integer 3.
   - Given the string `RACECAR`, your algorithm should return the integer 1.

   (b) A ***metapalindrome*** is a decomposition of a string into a sequence of non-empty palindromes, such that the sequence of palindrome lengths is itself a palindrome. For example, the decomposition

   `BUB` • `B` • `ALA` • `SEES` • `ABA` • `N` • `ANA`

   is a metapalindrome for the string `BUBBALASEESABANANA`, with the palindromic length sequence $(3, 1, 3, 4, 3, 1, 3)$. Describe and analyze an efficient algorithm to find the length of the shortest metapalindrome for a given string. For example:

   - Given the string `BUBBALASEESABANANA`, your algorithm should return the integer 7.
   - Given the string `PALINDROME`, your algorithm should return the integer 10.
   - Given the string `DEPOPED`, your algorithm should return the integer 1.

Solved Problem

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

BANANA$_{\text{ANANAS}}$ BAN$_{\text{ANA}}$ANA$_{\text{NAS}}$ B$_{\text{AN}}$AN$_{\text{A}}$A$_{\text{NA}}$NA$_{\text{S}}$

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

PRO$^{\text{D}}$G$^{\text{Y}}$R$^{\text{NAM}}$AMMI$^{\text{I}}$N$^{\text{C}}$G DY$_{\text{PRO}}$N$_{\text{G}}$A$_{\text{R}}$M$_{\text{AMM}}$IC$_{\text{ING}}$

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

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

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

We need to compute $Shuf(m, n)$.

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

```
SHUFFLE?(A[1..m], B[1..n], C[1..m+n]):
  Shuf[0,0] ← TRUE
  for j ← 1 to n
      Shuf[0,j] ← Shuf[0,j-1] ∧ (B[j] = C[j])
  for i ← 1 to n
      Shuf[i,0] ← Shuf[i-1,0] ∧ (A[i] = B[i])
      for j ← 1 to n
          Shuf[i,j] ← FALSE
          if A[i] = C[i+j]
              Shuf[i,j] ← Shuf[i,j] ∨ Shuf[i-1,j]
          if B[i] = C[i+j]
              Shuf[i,j] ← Shuf[i,j] ∨ Shuf[i,j-1]
  return Shuf[m,n]
```

The algorithm runs in $O(mn)$ *time*. ■

Rubric: Max 10 points: Standard dynamic programming rubric. No proofs required. Max 7 points for a slower polynomial-time algorithm; scale partial credit accordingly.

Standard dynamic programming rubric. For problems worth 10 poins:

- 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.) **Deadly Sin: Automatic zero if the English description is missing.**
  - + 1 point for stating how to call your function to get the final answer.
  - + 1 point for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 points for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 4 points for details of the dynamic programming algorithm
  - + 1 point for describing the memoization data structure
  - + 2 points for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested loops, be sure to specify the nesting order.
  - + 1 point for time analysis
- It is *not* necessary to state a space bound.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem says otherwise.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative pseudocode is not required for full credit***. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. (But you still need to describe the underlying recursive function in English.)
- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we have included them, significantly more students turned in algorithms that meet the target time bound but didn't work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).

CS/ECE 374 A ✧ Spring 2018

Homework 6

Due Tuesday, March 13, 2018 at 8pm

---

1. Suppose you are given an array $A[1..n]$ of positive integers, each of which is colored either red or blue. An *increasing back-and-forth subsequence* is an sequence of indices $I[1..\ell]$ with the following properties:

   - $1 \le I[j] \le n$ for all j.
   - $A[I[j]] < A[I[j+1]]$ for all $j < \ell$.
   - If $A[I[j]]$ is red, then $I[j+1] > I[j]$.
   - If $A[I[j]]$ is blue, then $I[j+1] < I[j]$.

   Less formally, suppose we start with a token on some integer $A[j]$, and then repeatedly move the token Left (if it's on a bLue square) or Right (if it's on a Red square), always moving from a smaller number to a larger number. Then the sequence of token positions is an increasing back-and-forth subsequence.

   Describe and analyze an efficient algorithm to compute the length of the longest increasing back-and-forth subsequence of a given array of n red and blue integers. For example, given the input array

1	1	0	2	5	9	6	6	4	5	8	9	7	7	3	2	3	8	4	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

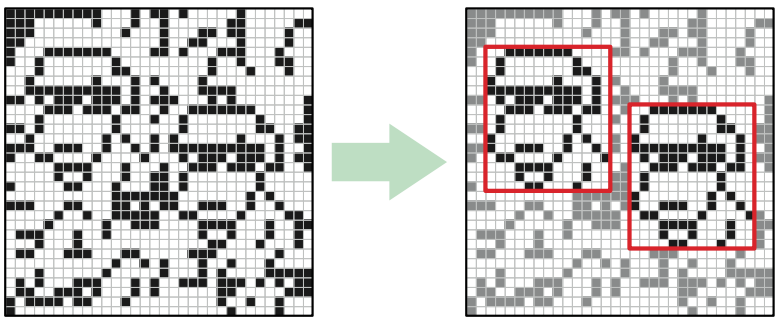
   your algorithm should return the integer 9, which is the length of the following increasing back-and-forth subsequence:

0	1	2	3	4	6	7	8	9
20	1	16	17	9	8	13	11	12

   (The small numbers are indices into the input array.)

2. Describe and analyze an algorithm that finds the largest rectangular pattern that appears more than once in a given bitmap. Your input is a two-dimensional array $M[1..n, 1..n]$ of bits; your output is the area of the repeated pattern. (The two copies of the pattern might overlap, but must not actually coincide.)

   For example, given the bitmap shown on the left in the figure below, your algorithm should return $15 \times 13 = 195$, because the same 15×13 doggo appears twice, as shown on the right, and this is the largest such pattern.



3. ***AVL trees*** were the earliest self-balancing balanced binary search trees, first described in 1962 by Georgy Adelson-Velsky and Evgenii Landis. An AVL tree is a binary search tree where for every node v, the height of the left subtree of v and the height of the right subtree of v differ by at most 1.

   Describe and analyze an efficient algorithm to construct an optimal AVL tree for a given set of keys and frequencies. Your input consists of a sorted array $A[1..n]$ of search keys and an array $f[1..n]$ of frequency counts, where $f[i]$ is the number of searches for $A[i]$. Your task is to construct an AVL tree for the given keys such that the total cost of all searches is as small as possible. This is exactly the same cost function that we considered in Thursday's class; the only difference is that the output tree must satisfy the AVL balance constraint.

   *[Hint: You do **not** need to know or use the insertion and deletion algorithms that keep the AVL tree balanced.]*

Solved Problems

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

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

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

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

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

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

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

Here $\boldsymbol{A[i] \sim A[k]}$ indicates that $A[i]$ and $A[k]$ are matching delimiters: Either $A[i] =$ (and $A[k] =$) or $A[i] =$ [and $A[k] =$].

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

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

■

Rubric: 10 points, standard dynamic programming rubric

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

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

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

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

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

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

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

(These recurrences do not require separate base cases, because $\sum \varnothing = 0$.) We can memoize these functions by adding two additional fields $v.yes$ and $v.no$ to each node v in the tree. The values at each node depend only on the vales at its children, so we can compute all $2n$ values using a postorder traversal of T.

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

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

(Yes, this is still dynamic programming; we're only traversing the tree recursively because that's the most natural way to traverse trees![a]) The algorithm spends $O(1)$ time at each node, and therefore runs in $\boldsymbol{O(n)}$ ***time*** altogether. ■

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

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

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

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

The function $MaxFun$ obeys the following recurrence:

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

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

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

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

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

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

[a] Like the previous solution, a direct recursive implementation would run in $O(\phi^n)$ time in the worst case, where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio.

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

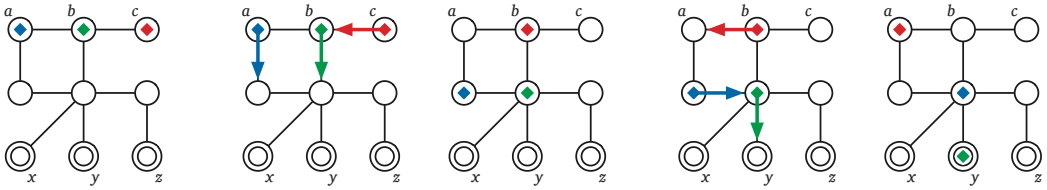
CS/ECE 374 A ✧ Spring 2018

Homework 7

Due Tuesday, March 27, 2018 at 8pm
(*after* Spring Break)

---

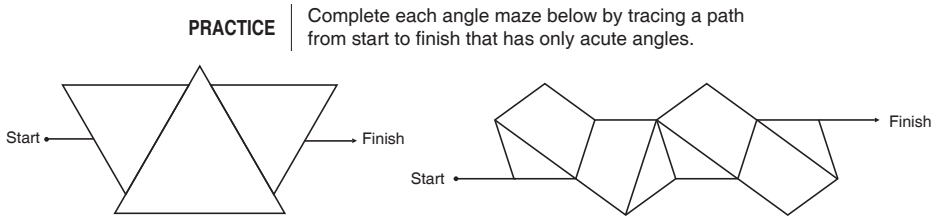
1. Consider the following solitaire game, played on a connected undirected graph G. Initially, tokens are placed on three start vertices a, b, c. In each turn, you *must* move *all three* tokens, by moving each token along an edge from its current vertex to an adjacent vertex. At the end of each turn, the three tokens *must* lie on three different vertices. Your goal is to move the tokens onto three goal vertices x, y, z; it does not matter which token ends up on which goal vertex.



The initial configuration of the puzzle and the first two turns of a solution.

Describe and analyze an algorithm to determine whether this puzzle is solvable. Your input consists of the graph G, the start vertices a, b, c, and the goal vertices x, y, z. Your output is a single bit: TRUE or FALSE. *[Hint: You've seen this sort of thing before.]*

2. The following puzzles appear in my daughter's elementary-school math workbook.[1]



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

You are given a connected undirected graph G, whose vertices are points in the plane and whose edges are line segments. Edges do not intersect, except at their endpoints. For example, a drawing of the letter X would have five vertices and four edges; the first maze above has 13 vertices and 15 edges. You are also given two vertices Start and Finish.

Your algorithm should return TRUE if G contains a walk from Start to Finish that has only acute angles, and FALSE otherwise. Formally, a walk through G is valid if, for any two consecutive edges $u \to v \to w$ in the walk, either $\angle uvw = \pi$ or $0 < \angle uvw < \pi/2$. Assume you have a subroutine that can determine in $O(1)$ time whether two segments with a common vertex define a straight, obtuse, right, or acute angle.

---

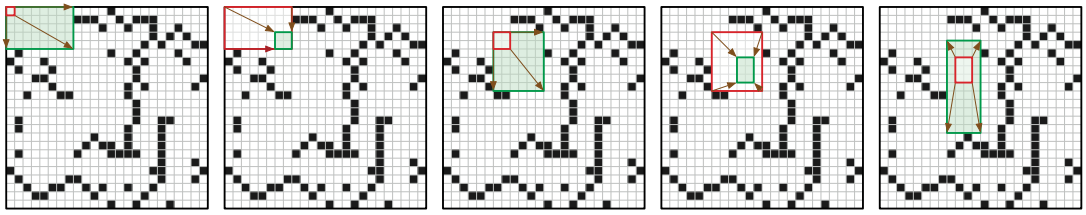
[1] Jason Batterson and Shannon Rogers, *Beast Academy Math: Practice 3A*, 2012. See https://www.beastacademy.com/resources/printables.php for more examples.

3. ***Rectangle Walk*** is a new abstract puzzle game, available for only 99¢ on Steam, iOS, Android, Xbox One, Playstation 5, Nintendo Wii U, Atari 2600, Palm Pilot, Commodore 64, TRS-80, Sinclair ZX-1, DEC PDP-8, ILLIAC V, Zuse Z3, Duramesc, Odhner Arithmometer, Analytical Engine, Jacquard Loom, Horologium mirabile Lundense, Leibniz Stepped Reckoner, Antikythera Mechanism, and Pile of Sticks.

   The game is played on an $n \times n$ grid of black and white squares. The player moves a rectangle through this grid, subject to the following conditions:

   - The rectangle must be aligned with the grid; that is, the top, bottom, left, and right coordinates must be integers.
   - The rectangle must fit within the $n \times n$ grid, and it must contain at least one grid cell.
   - The rectangle must not contain a black square.
   - In a single move, the player can replace the current rectangle r with any rectangle r' that either contains r or is contained in r.

   Initially, the player's rectangle is a 1×1 square in the upper right corner. The player's goal is to reach a 1×1 square in the bottom left corner using as few moves as possible.



The first five steps in a Rectangle Walk.

   Describe and analyze an algorithm to compute the length of the shortest Rectangle Walk in a given bitmap. Your input is an array $M[1..n, 1..n]$, where $M[i, j] = 1$ indicates a black square and $M[i, j] = 0$ indicates a white square. You can assume that a valid rectangle walk exists; in particular, $M[1, 1] = 0$ and $M[n, n] = 0$. For example, given the bitmap shown above, (I think) your algorithm should return the integer 18.

Solved Problem

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

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

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

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

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

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

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

   Solution: Let A, B, C denote the capacities of the three jars. We reduce the problem to breadth-first search in the following directed graph:

   - $V = \{(a,b,c) \mid 0 \le a \le A \text{ and } 0 \le b \le B \text{ and } 0 \le c \le C\}$. Each vertex corresponds to a possible ***configuration*** of water in the three jars. There are $(A+1)(B+1)(C+1) = O(ABC)$ vertices altogether.
   - The graph has a directed edge $(a,b,c) \to (a',b'c')$ whenever it is possible to move from the first configuration to the second in one step. Specifically, there is an edge from (a,b,c) to each of the following vertices (except those already equal to (a,b,c)):
     - $(0,b,c)$ and $(a,0,c)$ and $(a,b,0)$ — dumping a jar into the lake
     - (A,b,c) and (a,B,c) and (a,b,C) — filling a jar from the lake
     - $\begin{Bmatrix} (0, a+b, c) & \text{if } a+b \le B \\ (a+b-B, B, c) & \text{if } a+b \ge B \end{Bmatrix}$ — pouring from jar 1 into jar 2
     - $\begin{Bmatrix} (0, b, a+c) & \text{if } a+c \le C \\ (a+c-C, b, C) & \text{if } a+c \ge C \end{Bmatrix}$ — pouring from jar 1 into jar 3

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

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

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

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

Rubric: 10 points: standard graph reduction rubric (see next page)

- Brute force construction is fine.

— 1 for calling Dijkstra instead of BFS

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

Standard rubric for graph reduction problems. For problems out of 10 points:

- + 1 for correct vertices, *including English explanation for each vertex*
- + 1 for correct edges
  - −½ for forgetting "directed" if the graph is directed
- + 1 for stating the correct problem (in this case, "shortest path")
  - − "Breadth-first search" is not a problem; it's an algorithm!
- + 1 for correctly applying the correct algorithm (in this case, "breadth-first search from $(0,0,0)$ and then examine every target vertex")
- + 1 for time analysis in terms of the input parameters.
- + 5 for other details of the reduction
  - – If your graph is constructed by naive brute force, you do not need to describe the construction algorithm; in this case, points for vertices, edges, problem, algorithm, and running time are all doubled.
  - – Otherwise, apply the appropriate rubric, *including Deadly Sins*, to the construction algorithm. For example, for a solution that uses dynamic programming to build the graph quickly, apply the standard dynamic programming rubric.

CS/ECE 374 A ✧ Spring 2018

☙ Homework 8 ❧

Due Tuesday, April 3, 2018 at 8pm

This is the last homework before Midterm 2.

---

1. After moving to a new city, you decide to choose a walking route from your home to your new office. To get a good daily workout, your route must consist of an uphill path (for exercise) followed by a downhill path (to cool down), or just an uphill path, or just a downhill path.[1] (You'll walk the same path home, so you'll get exercise one way or the other.) But you also want the *shortest* path that satisfies these conditions, so that you actually get to work on time.

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

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

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

2. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero.

   (a) How could we delete an arbitrary vertex v from this graph, without changing the shortest-path distance between any other pair of vertices? Describe an algorithm that constructs a directed graph $G' = (V', E')$ with weighted edges, where $V' = V \setminus \{v\}$, and the shortest-path distance between any two nodes in G' is equal to the shortest-path distance between the same two nodes in G, in $O(V^2)$ time.

   (b) Now suppose we have already computed all shortest-path distances in G'. Describe an algorithm to compute the shortest-path distances in the original graph G from v to every other vertex, and from every other vertex to v, all in $O(V^2)$ time.

   (c) Combine parts (a) and (b) into another all-pairs shortest path algorithm that runs in $O(V^3)$ time. (The resulting algorithm is *almost* the same as Floyd-Warshall!)

---

[1] A *hill* is an area of land that extends above the surrounding terrain, usually at a fairly gentle gradient. Like a building, but smoother and made of dirt and rock and trees instead of steel and concrete. It's hard to explain.

3. The first morning after returning from a glorious spring break, Alice wakes to discover that her car won't start, so she has to get to her classes at Sham-Poobanana University by public transit. She has a complete transit schedule for Poobanana County. The bus routes are represented in the schedule by a directed graph G, whose vertices represent bus stops and whose edges represent bus routes between those stops. For each edge $u \to v$, the schedule records three positive real numbers:

   - $\ell(u \to v)$ is the length of the bus ride from stop u to stop v (in minutes)
   - $f(u \to v)$ is the first time (in minutes past 12am) that a bus leaves stop u for stop v.
   - $\Delta(u \to v)$ is the time between successive departures from stop u to stop v (in minutes).

   Thus, the first bus for this route leaves u at time $f(u \to v)$ and arrives at v at time $f(u \to v) + \ell(u \to v)$, the second bus leaves u at time $f(u \to v) + \Delta(u \to v)$ and arrives at v at time $f(u \to v) + \Delta(u \to v) + \ell(u \to v)$, the third bus leaves u at time $f(u \to v) + 2 \cdot \Delta(u \to v)$ and arrives at v at time $f(u \to v) + 2 \cdot \Delta(u \to v) + \ell(u \to v)$, and so on.

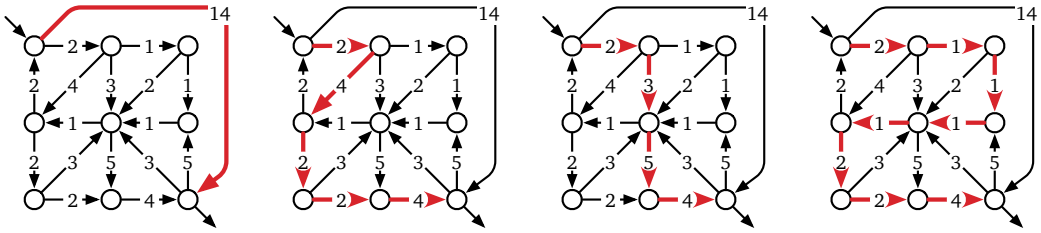
   Alice wants to leaves from stop s (her home) at a certain time and arrive at stop t (The See-Bull Center for Fake News Detection) as quickly as possible. If Alice arrives at a stop on one bus at the exact time that another bus is scheduled to leave, she can catch the second bus. Because she's a student at SPU, Alice can ride the bus for free, so she doesn't care how many times she has to change buses.

   Describe and analyze an algorithm to find the earliest time Alice can reach her destination. Your input consists of the directed graph $G = (V, E)$, the vertices s and t, the values $\ell(e), f(e), \Delta(e)$ for each edge $e \in E$, and Alice's starting time (in minutes past 12am).

   [Hint: In this rare instance, modifying the algorithm may be more efficient than modifying the input graph. Don't describe the algorithm from scratch; just describe your changes.]

Solved Problem

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



Four (of many) equal-length shortest paths.

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

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

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

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

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

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

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

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

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

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

CS/ECE 374 A ✧ Spring 2018

☙ Homework 9 ❧

Due Tuesday, April 17, 2018 at 8pm

---

1. For any integer k, the problem kSAT is defined as follows:

   - INPUT: A boolean formula Φ in conjunctive normal form, with exactly k distinct literals in each clause.
   - OUTPUT: TRUE if Φ has a satisfying assignment, and FALSE otherwise.

   (a) Describe a polynomial-time reduction from **2**SAT to **3**SAT, and prove that your reduction is correct.

   (b) Describe and analyze a polynomial-time algorithm for **2**SAT. *[Hint: This problem is strongly connected to topics covered earlier in the semester.]*

   (c) Why don't these results imply a polynomial-time algorithm for 3SAT?

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

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

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

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

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

3. *Pebbling* is a solitaire game played on an undirected graph G, where each vertex has zero or more *pebbles*. A single *pebbling move* removes two pebbles from some vertex v and adds one pebble to an arbitrary neighbor of v. (Obviously, v must have at least two pebbles before the move.) The PEBBLECLEARING problem asks, given a graph $G = (V, E)$ and a pebble count $p(v)$ for each vertex v, whether is there a sequence of pebbling moves that removes all but one pebble. Prove that PEBBLECLEARING is NP-hard.

Solved Problem

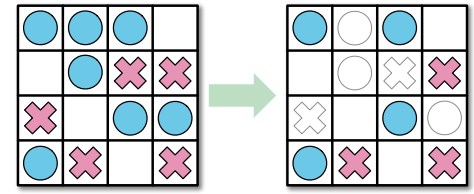
4. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions:

   (1) Every row contains at least one stone.

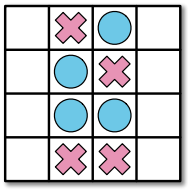
   (2) No column contains stones of both colors.

   For some initial configurations of stones, reaching this goal is impossible; see the example below.

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



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

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

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

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

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

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

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

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

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

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

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

This reduction clearly requires only polynomial time. ■

Rubric (Standard polynomial-time reduction rubric): 10 points =

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

- An incorrect polynomial-time reduction that still satisfies half of the correctness proof is worth at most 4/10.
- A reduction in the wrong direction is worth 0/10.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

SteamedHams: Aurora borealis? At this time of year, at this time of day, in this part of the country, localized entirely within your kitchen? May I see it?

CS/ECE 374 A ✧ Spring 2018

Homework 10

Due Tuesday, April 24, 2018 at 8pm

This is the last graded homework before the final exam.

---

1. (a) A subset S of vertices in an undirected graph G is ***half-independent*** if each vertex in S is adjacent to *at most one* other vertex in S. Prove that finding the size of the largest half-independent set of vertices in a given undirected graph is NP-hard.

   (b) A subset S of vertices in an undirected graph G is ***sort-of-independent*** if if each vertex in S is adjacent to *at most 374* other vertices in S. Prove that finding the size of the largest sort-of-independent set of vertices in a given undirected graph is NP-hard.

2. Fix an alphabet $\Sigma = \{0, 1\}$. Prove that the following problems are NP-hard.[1]

   (a) Given a regular expression R over the alphabet Σ, is $L(R) \neq \Sigma^*$?

   (b) Given an NFA M over the alphabet Σ, is $L(M) \neq \Sigma^*$?

   *[Hint: Encode all the **bad** choices for some problem into a regular expression R, so that if **all** choices are bad, then $L(R) = \Sigma^*$.]*

3. Let $\langle M \rangle$ denote the encoding of a Turing machine M (or if you prefer, the Python source code for the executable code M). Recall that $x \bullet y$ denotes the concatenation of strings x and y. Prove that the following language is undecidable.

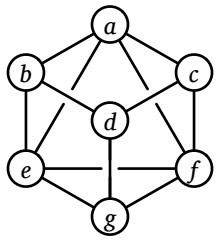
   $$\textsc{SelfSelfAccept} := \{\langle M \rangle \mid M \text{ accepts the string } \langle M \rangle \bullet \langle M \rangle\}$$

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

[1] In fact, both of these problems are NP-hard even when $|\Sigma| = 1$, but proving that is much more difficult.

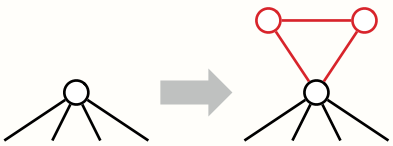
Solved Problem

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



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

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



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

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

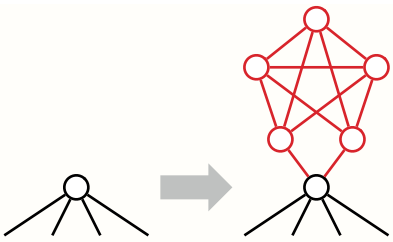
$\Longrightarrow$ Suppose G has a Hamiltonian cycle $v_1 \to v_2 \to \cdots \to v_n \to v_1$. We can construct a double-Hamiltonian tour of H by replacing each vertex v_i with the following walk:

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

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

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

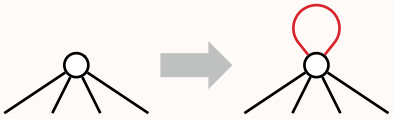
With more effort, we can construct a graph H that contains a double-Hamiltonian tour ***that traverses each edge of H at most once*** if and only if G contains a Hamiltonian cycle. For each vertex v in G we attach a more complex gadget containing five vertices and eleven edges, as shown on the next page.



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

■

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

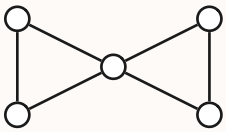


An incorrect vertex gadget.

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

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

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



This graph has a double-Hamiltonian tour.

♣

Rubric: 10 points, standard polynomial-time reduction rubric

CS/ECE 374 A ✧ Spring 2018

☙ "Homework" 11 ❧

"Due" Tuesday, May 1, 2018

This homework is optional. However, **similar undecidability questions may appear on the final exam**, so we still strongly recommend treating at least those questions as regular homework. Solutions will be released next Tuesday as usual.

---

1. 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 (or fewer) cells on its tape and eventually accepts.

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

   $$\textsc{SquareSpace} = \left\{ \langle M, w \rangle \;\middle|\; M \text{ accepts } w \text{ in space } |w|^2 \right\}$$

   (b) Prove that the following language is undecidable:

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

2. Consider the following language:

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

   (a) Prove that Picky is undecidable.

   (b) Sketch a Turing machine/algorithm that *accepts* Picky.

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

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

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

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

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

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

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

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

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

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

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

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

To think about later: Let $\#(a, w)$ denote the number of times symbol a appears in string w. For example, $\#(\texttt{X}, \texttt{WTF374}) = 0$ and $\#(\texttt{0}, \texttt{0000101010010100}) = 12$.

4. Give a formal recursive definition of $\#(a, w)$.
5. Prove that $\#(a, w \bullet z) = \#(a, w) + \#(a, z)$ for all symbols a and all strings w and z.
6. Prove that $\#(a, w^R) = \#(a, w)$ for all symbols a and all strings w.

Give regular expressions for each of the following languages over the alphabet {0, 1}.

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

Work on these later:

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

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

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

★10. All strings in which the substring 000 appears an even number of times.
(For example, 0001000 and 0000 are in this language, but 00000 is not.)

Describe deterministic finite-state automata that accept each of the following languages over the alphabet $\Sigma = \{0, 1\}$. Describe briefly what each state in your DFAs *means*.

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

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

Work on these later:

7. All strings w such that *in every prefix of* w, the number of `0`s and `1`s differ by at most 1.
8. All strings containing at least two `0`s and at least one `1`.
9. All strings w such that *in every prefix of* w, the number of `0`s and `1`s differ by at most 2.

*10. All strings in which the substring `000` appears an even number of times.
(For example, `0001000` and `0000` are in this language, but `00000` is not.)

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

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

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

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

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

Work on these later:

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

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

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

Prove that each of the following languages is ***not*** regular.

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

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

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

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

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

Work on these later:

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

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

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

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

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

   (c) **Think about later:** Convert the DFA you just constructed into a regular expression using Han and Wood's algorithm. You should *not* get the same regular expression you started with.

   (d) What is this language?

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

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

   (c) **Think about later:** Convert the DFA you just constructed into a regular expression using Han and Wood's algorithm. You should *not* get the same regular expression you started with.

   (d) What is this language?

Let L be an arbitrary regular language.

1. Prove that the language $\mathit{insert}\texttt{1}(L) := \{x\texttt{1}y \mid xy \in L\}$ is regular.

   Intuitively, $\mathit{insert}\texttt{1}(L)$ is the set of all strings that can be obtained from strings in L by inserting exactly one `1`. For example, if $L = \{\varepsilon, \texttt{OOK!}\}$, then $\mathit{insert}\texttt{1}(L) = \{\texttt{1}, \texttt{1OOK!}, \texttt{O1OK!}, \texttt{OO1K!}, \texttt{OOK1!}, \texttt{OOK!1}\}$.

2. Prove that the language $\mathit{delete}\texttt{1}(L) := \{xy \mid x\texttt{1}y \in L\}$ is regular.

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

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

3. Consider the following recursively defined function on strings:

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

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

   - $\mathit{stutter}(\texttt{PRESTO}) = \texttt{PPRREESSTTOO}$
   - $\mathit{stutter}(\texttt{HOCUS}\diamond\texttt{POCUS}) = \texttt{HHOOCCUUSS}\diamond\diamond\texttt{PPOOCCUUSS}$

   Let L be an arbitrary regular language.

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

   (b) Prove that the language $\mathit{stutter}(L) := \{\mathit{stutter}(w) \mid w \in L\}$ is regular.

4. Consider the following recursively defined function on strings:

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

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

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

   Once again, let L be an arbitrary regular language.

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

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

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

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

$$\mathit{flipOdds}(0000111101010101) = 1010010111111111$$

Solution: Let $M = (Q, s, A, \delta)$ be a DFA that accepts L. We construct a new DFA $M' = (Q', s', A', \delta')$ that accepts $\textsc{FlipOdds}(L)$ as follows.

Intuitively, M' receives some string $\mathit{flipOdds}(w)$ as input, restores every other bit to obtain w, and simulates M on the restored string w.

Each state (q, flip) of M' indicates that M is in state q, and we need to flip the next input bit if $\mathit{flip} = \textsc{True}$

$$Q' = Q \times \{\textsc{True}, \textsc{False}\}$$
$$s' = (s, \textsc{True})$$
$$A' =$$
$$\delta'((q, \mathit{flip}), a) =$$

■

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

$$\mathit{flipOdd1s}(0000\underline{1}\underline{1}10\underline{1}010\underline{1}01) = 00000\underline{1}0\underline{1}00\underline{1}0\underline{0}01$$

Solution: Let $M = (Q, s, A, \delta)$ be a DFA that accepts L. We construct a new DFA $M' = (Q', s', A', \delta')$ that accepts $\textsc{UnflipOdd1s}(L)$ as follows.

Intuitively, M' receives some string w as input, flips every other 1 bit, and simulates M on the transformed string.

Each state (q, flip) of M' indicates that M is in state q, and we need to flip the next 1 bit of and only if $\mathit{flip} = \textsc{True}$.

$$Q' = Q \times \{\textsc{True}, \textsc{False}\}$$
$$s' = (s, \textsc{True})$$
$$A' =$$
$$\delta'((q, \mathit{flip}), a) =$$

■

3. FLIPODD1S$(L) := \{flipOdd1s(w) \mid w \in L\}$, where the function *flipOdd1* is defined as in the previous problem.

Solution: Let $M = (Q, s, A, \delta)$ be a DFA that accepts L. We construct a new **NFA** $M' = (Q', s', A', \delta')$ that accepts FLIPODD1S(L) as follows.

Intuitively, M' receives some string $flipOdd1s(w)$ as input, ***guesses*** which `0` bits to restore to `1`s, and simulates M on the restored string w. No string in FLIPODD1S(L) has two `1`s in a row, so if M' ever sees `11`, it rejects.

Each state $(q, flip)$ of M' indicates that M is in state q, and we need to flip a `0` bit before the next `1` if $flip =$ TRUE.

$$Q' = Q \times \{\text{TRUE}, \text{FALSE}\}$$
$$s' = (s, \text{TRUE})$$
$$A' =$$
$$\delta'((q, flip), a) =$$

■

4. FARO$(L) := \left\{faro(w, x) \;\middle|\; w, x \in L \text{ and } |w| = |x|\right\}$, where the function *faro* is defined recursively as follows:

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

For example, $faro($`0001101`, `1111001`$) =$ `0101011110011`. (A "faro shuffle" splits a deck of cards into two equal piles and then perfectly interleaves them.)

Solution: Let $M = (Q, s, A, \delta)$ be a DFA that accepts L. We construct a DFA $M' = (Q', s', A', \delta')$ that accepts FARO(L) as follows.

Intuitively, M' reads the string $faro(w, x)$ as input, splits the string into the subsequences w and x, and passes each of those strings to an independent copy of M.

Each state $(q_1, q_2, next)$ indicates that the copy of M that gets w is in state q_1, the copy of M that gets x is in state q_2, and *next* indicates which copy gets the next input bit.

$$Q' = Q \times Q \times \{1, 2\}$$
$$s' = (s, s, 1)$$
$$A' =$$
$$\delta'((q_1, q_2, next), a) =$$

■

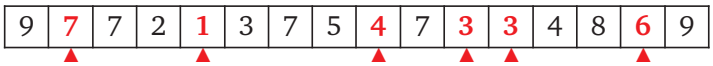
Here are several problems that are easy to solve in $O(n)$ time, essentially by brute force. Your task is to design algorithms for these problems that are significantly faster.

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

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

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

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

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

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

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

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

To think about later:

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

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

   your algorithm should return the integer 7.

In lecture, Jeff described an algorithm of Karatsuba that multiplies two n-digit integers using $O(n^{\lg 3})$ single-digit additions, subtractions, and multiplications. In this lab we'll look at some extensions and applications of this algorithm.

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

2. Describe an algorithm to compute the decimal representation of 2^n in $O(n^{\lg 3})$ time.

   [Hint: Repeated squaring. The standard algorithm that computes one decimal digit at a time requires $\Theta(n^2)$ time.]

3. Describe a divide-and-conquer algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(n^{\lg 3})$ time.

   [Hint: Let $x = a \cdot 2^{n/2} + b$. Watch out for an extra log factor in the running time.]

Think about later:

4. Suppose we can multiply two n-digit numbers in $O(M(n))$ time. Describe an algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(M(n)\log n)$ time.

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

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

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

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

   For example, given the array

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

   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

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

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

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

   For example, given the array

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

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

To think about later:

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

   For example, given the array

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

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

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

   For example, given the array

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

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

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

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

Describe and analyze ***dynamic programming*** algorithms for the following problems. For the first three, use the backtracking algorithms you developed on Wednesday.

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

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

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

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

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

Basic steps in developing a dynamic programming algorithm

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

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

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

2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:

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

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

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

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

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

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

Lenny Adve, the founding dean of the new Maximilian Q. Levchin College of Computer Science, has commissioned a series of snow ramps on the south slope of the Orchard Downs sledding hill[1] and challenged Bill Kudeki, head of the Department of Electrical and Computer Engineering, to a sledding contest. Bill and Lenny will both sled down the hill, each trying to maximize their air time. The winner gets to expand their department/college into Siebel Center, the new ECE Building, *and* the English Building; the loser has to move their entire department/college under the Boneyard bridge next to Everitt Lab (along with the English department).

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

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

   Describe and analyze an algorithm to determine the maximum *total* distance that Lenny and Bill can travel through the air. *[Hint: Do whatever you **feel** like you wanna do. Gosh!]*

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

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

3. **To think about later:** When the lawyers realized that imposing their restriction didn't immediately shut down the contest, they added a new restriction: No ramp can be used more than once! Disgusted by the legal interference, Lenny and Bill give up on their bet and decide to cooperate to put on a good show for the spectators.

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

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

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

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

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

Think about later:

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

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

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

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

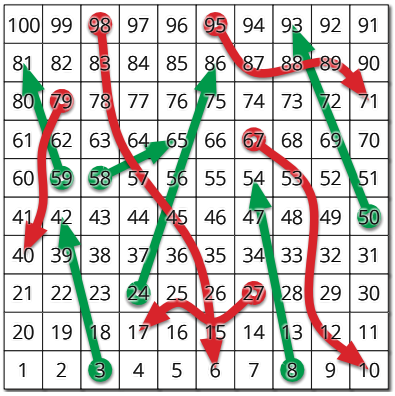
   Describe and analyze an algorithm to compute, given a list of integers separated by $+$ and $-$ signs, the maximum possible value the expression can take by adding parentheses. Parentheses must be used only to group additions and subtractions; in particular, do not use them to create implicit multiplication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

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

- What are the vertices? What does each vertex represent?
- What are the edges? Are they directed or undirected?
- If the vertices and/or edges have associated values, what are they?
- What 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.

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). Then if the token is at the *top* of a snake, you ***must*** slide the token down to the bottom of that snake, and if the token is at the *bottom* of a ladder, you ***may*** move the token up to the top of that ladder.

Describe and analyze an efficient algorithm to compute the smallest number of moves required for the token to reach the last square of the Snakes and Ladders board.

2. Let G be an 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 efficient 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).

Think about later:

3. Let G be an undirected graph. Suppose we start with 374 coins on 374 arbitrarily chosen vertices of G. At every step, each coin ***must*** move to an adjacent vertex. Describe and analyze an efficient 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 starting vertices $s_1, s_2, \ldots, s_{374}$ (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 does each vertex represent?
- What are the edges? Are they directed or undirected?
- If the vertices and/or edges have associated values, what are they?
- What problem do you need to solve on this graph?
- What standard algorithm are you using to solve that problem?
- What is the running time of your entire algorithm, *including* the time to build the graph, *as a function of the original input parameters*?

---

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

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

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

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

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

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

To think about later:

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

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

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

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

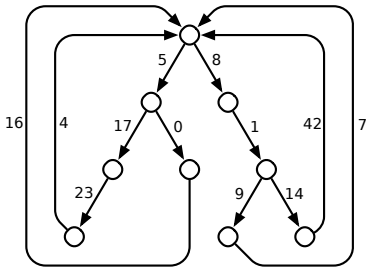
2. You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cites that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving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

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

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

To think about later:

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



A looped tree.

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

   (b) Describe and analyze a faster algorithm.

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

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

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

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

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

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

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

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

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

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

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

[1] No, you can't.

1. ***Flappy Bird*** is a popular mobile game written by Nguyễn Hà Đông, originally released in May 2013. The game features a bird named "Faby", who flies to the right at constant speed. Whenever the player taps the screen, Faby is given a fixed upward velocity; between taps, Faby falls due to gravity. Faby flies through a landscape of pipes until it touches either a pipe or the ground, at which point the game is over. Your task, should you choose to accept it, is to develop an algorithm to play Flappy Bird automatically.

   Well, okay, not Flappy Bird exactly, but the following drastically simplified variant, which I will call ***Flappy Pixel***. Instead of a bird, Faby is a single point, specified by three integers: horizontal position x (in pixels), vertical position y (in pixels), and vertical speed y' (in pixels per frame). Faby's environment is described by two arrays $Hi[1..n]$ and $Lo[1..n]$, where for each index i, we have $0 < Lo[i] < Hi[i] < h$ for some fixed height value h. The game is described by the following piece of pseudocode:

```
FlappyPixel(Hi[1..n], Lo[1..n]):
  y ← ⌈h/2⌉
  y' ← 0
  for x ← 1 to n
      if the player taps the screen
          y' ← 10              ⟨⟨flap⟩⟩
      else
          y' ← y' − 1          ⟨⟨fall⟩⟩
      y ← y + y'
      if y < Lo[x] or y > Hi[x]
          return False         ⟨⟨player loses⟩⟩
  return True                  ⟨⟨player wins⟩⟩
```

   Notice that in each iteration of the main loop, the player has the option of tapping the screen.

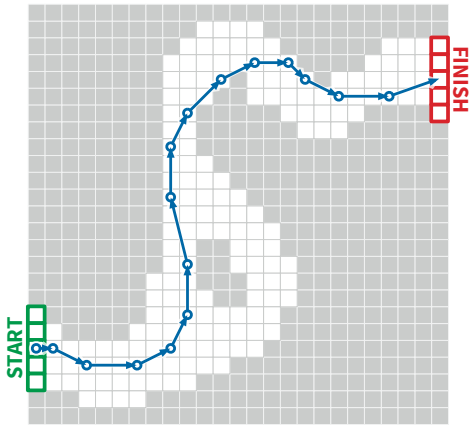
   Describe and analyze an algorithm to determine the minimum number of times that the player must tap the screen to win Flappy Pixel, given the integer h and the arrays $Hi[1..n]$ and $Lo[1..n]$ as input. If the game cannot be won at all, your algorithm should return ∞. Describe the running time of your algorithm as a function of n and h.

[Problem 2 is on the back.]

2. ***Racetrack*** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.[1] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

   Each car has a *position* and a *velocity*, both with integer x- and y-coordinates. A subset of grid squares is marked as the *starting area*, and another subset is marked as the *finishing area*. The initial position of each car is chosen by the player somewhere in the starting area; the initial velocity of each car is always $(0, 0)$. At each step, the player optionally changes each component of the velocity by at most 1. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race.[2] The race ends when the first car reaches a position inside the finishing area.

velocity	position
$(0, 0)$	$(1, 5)$
$(1, 0)$	$(2, 5)$
$(2, -1)$	$(4, 4)$
$(3, 0)$	$(7, 4)$
$(2, 1)$	$(9, 5)$
$(1, 2)$	$(10, 7)$
$(0, 3)$	$(10, 10)$
$(-1, 4)$	$(9, 14)$
$(0, 3)$	$(9, 17)$
$(1, 2)$	$(10, 19)$
$(2, 2)$	$(12, 21)$
$(2, 1)$	$(14, 22)$
$(2, 0)$	$(16, 22)$
$(1, -1)$	$(17, 21)$
$(2, -1)$	$(19, 20)$
$(3, 0)$	$(22, 20)$
$(3, 1)$	$(25, 21)$



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

Suppose the racetrack is represented by an $n \times n$ array of bits, where each `0` bit represents a grid point inside the track, each `1` bit represents a grid point outside the track, the "starting line" consists of all `0` bits in column 1, and the "finishing line" consists of all `0` bits in column n.

Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting line to the finish line of a given racetrack.

[Hint: Your initial analysis can be improved.]

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

[2] However, it is not necessary for the entire line segment between the old position and the new position to lie inside the track. Sometimes Speed Racer has to push the A button.

To think about later:

3. Consider the following variant of Flappy Pixel. The mechanics of the game are unchanged, but now the environment is specified by an array $Points[1..n, 1..h]$ of integers, which could be positive, negative, or zero. If Faby falls off the top or bottom edge of the environment, the game immediately ends and the player gets nothing. Otherwise, at each frame, the player earns $Points[x, y]$ points, where (x, y) is Faby's current position. The game ends when Faby reaches the right end of the environment.

```
FlappyPixel2(Points[1..n]):
  score ← 0
  y ← ⌈h/2⌉
  y' ← 0
  for x ← 1 to n
      if the player taps the screen
          y' ← 10              ⟨⟨flap⟩⟩
      else
          y' ← y' − 1          ⟨⟨fall⟩⟩
      y ← y + y'
      if y < 1 or y > h
          return −∞            ⟨⟨fail⟩⟩
      score ← score + Points[x, y]
  return score
```

Describe and analyze an algorithm to determine the maximum possible score that a player can earn in this game.

4. We can also consider a similar variant of Racetrack. Instead of bits, the "track" is described by an array $Points[1..n, 1..n]$ of *numbers*, which could be positive, negative, or zero. Whenever the car lands on a grid cell (i, j), the player receives $Points[i, j]$ points. Forbidden grid cells are indicated by $Points[i, j] = -\infty$.

Describe and analyze an algorithm to find the largest possible score that a player can earn by moving a car from column 1 (the starting line) to column n (the finish line).

[Hint: Wait, what if all the point values are positive?]

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

   - INPUT: A boolean circuit K with n inputs and one output.
   - OUTPUT: TRUE if there are input values $x_1, x_2, \ldots, x_n \in \{\text{TRUE}, \text{FALSE}\}$ that make K output TRUE, and FALSE otherwise.

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

   - INPUT: A boolean circuit K with n inputs and one output.
   - OUTPUT: Input values $x_1, x_2, \ldots, x_n \in \{\text{TRUE}, \text{FALSE}\}$ that make K output TRUE, or NONE if there are no such inputs.

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

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

   - INPUT: An undirected graph G and an integer k.
   - OUTPUT: TRUE if G has an independent set of size k, and FALSE otherwise.

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

   - INPUT: An undirected graph G.
   - OUTPUT: The size of the largest independent set in G.

   [Hint: You've seen this problem before.]

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

   - INPUT: An undirected graph G.
   - OUTPUT: An independent set in G of maximum size.

To think about later:

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

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

   - INPUT: An undirected graph G and an integer k.
   - OUTPUT: TRUE if G has a proper coloring with k colors, and FALSE otherwise.

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

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

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

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

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

---

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

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

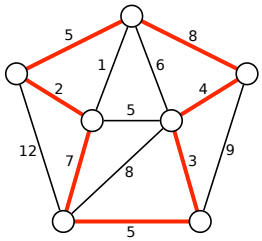
   (b) Prove that kCOLOR problem is NP-hard for any $k \geq 3$.

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

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

To think about later:

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

Prove that the following languages are undecidable.

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

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

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

```
DECIDEHALT(⟨M, w⟩):
  Encode the following Turing machine M′:
      M′(x):
        run M on input w
        return TRUE
  if DECIDEACCEPTILLINI(⟨M′⟩)
      return TRUE
  else
      return FALSE
```

We prove this reduction correct as follows:

$\Longrightarrow$ Suppose M halts on input w.

Then M' accepts *every* input string x.

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

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

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

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

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

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

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

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

In both cases, DECIDEHALT is correct. But that's impossible, because HALT is undecidable. We conclude that the algorithm DECIDEACCEPTILLINI does not exist. ■

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

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

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

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

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

You may find the following Turing machines useful:

- M_{ACCEPT} accepts every input.
- M_{REJECT} rejects every input.
- M_{HANG} infinite-loops on every input.

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

1. $\text{ACCEPTREGULAR} := \{\langle M \rangle \mid \text{ACCEPT}(M) \text{ is regular}\}$
2. $\text{ACCEPTILLINI} := \{\langle M \rangle \mid M \text{ accepts the string } \texttt{ILLINI}\}$
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 would you prove that? Remember that we know several ways to prove undecidability:

- Diagonalization: Assume the language is decidable, and derive an algorithm with self-contradictory behavior.
- Reduction: Assume the language is decidable, and derive an algorithm for a known undecidable language, like HALT or SELFREJECT or NEVERACCEPT.
- Rice's Theorem: Find an appropriate family of languages $\mathcal{L}$, a machine Y that accepts a language in $\mathcal{L}$, and a machine N that does not accept a language in $\mathcal{L}$.
- Closure: If two languages L and L' are decidable, then the languages $L \cap L'$ and $L \cup L'$ and $L \setminus L'$ and $L \oplus L'$ and L^* are all decidable, too.

---

6. $\text{ACCEPT}\{\{\varepsilon\}\} := \{\langle M \rangle \mid M \text{ accepts only the string } \varepsilon\text{; that is, } \text{ACCEPT}(M) = \{\varepsilon\}\}$
7. $\text{ACCEPT}\{\emptyset\} := \{\langle M \rangle \mid M \text{ does not accept any strings; that is, } \text{ACCEPT}(M) = \emptyset\}$
8. $\text{ACCEPT}\emptyset := \{\langle M \rangle \mid \text{ACCEPT}(M) \text{ is not an acceptable language}\}$
9. $\text{ACCEPT=REJECT} := \{\langle M \rangle \mid \text{ACCEPT}(M) = \text{REJECT}(M)\}$
10. $\text{ACCEPT}\neq\text{REJECT} := \{\langle M \rangle \mid \text{ACCEPT}(M) \neq \text{REJECT}(M)\}$
11. $\text{ACCEPT}\cup\text{REJECT} := \{\langle M \rangle \mid \text{ACCEPT}(M) \cup \text{REJECT}(M) = \Sigma^*\}$

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, check "Yes" if the statement is ***always*** true and "No" 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) Every infinite language is regular.

   (b) If L is not regular, then for every string $w \in L$, there is a DFA that accepts w.

   (c) If L is context-free and L has a finite fooling set, then L is regular.

   (d) If L is regular and $L' \cap L = \emptyset$, then L' is regular.

   (e) The language $\{0^i 1^j 0^k \mid i + j + k \geq 374\}$ is not regular.

   (f) The language $\{0^i 1^j 0^k \mid i + j - k \geq 374\}$ is not regular.

   (g) Let $M = (Q, \{0,1\}, s, A, \delta)$ be an arbitrary DFA, and let $M' = (Q, \{0,1\}, s, A, \delta')$ be the DFA obtained from M by changing every 0-transition into a 1-transition and vice versa. More formally, M and M' have the same states, input alphabet, starting state, and accepting states, but $\delta'(q,0) = \delta(q,1)$ and $\delta'(q,1) = \delta(q,0)$. Then $L(M) \cap L(M') = \emptyset$.

   (h) Let $M = (Q, \Sigma, s, A, \delta)$ be an arbitrary NFA, and $M' = (Q', \Sigma, s, A', \delta')$ be any NFA obtained from M by deleting some subset of the states. More formally, we have $Q' \subseteq Q$, $A' = A \cap Q'$, and $\delta'(q,a) = \delta(q,a) \cap Q'$ for all $q \in Q'$. Then $L(M') \subseteq L(M)$.

   (i) For every regular language L, the language $\left\{0^{|w|} \;\middle|\; w \in L\right\}$ is also regular.

   (j) For every context-free language L, the language $\left\{0^{|w|} \;\middle|\; w \in L\right\}$ is also context-free.

2. For any language L, define

$$\textsc{StripInit0s}(L) = \left\{w \;\middle|\; 0^j w \in L \text{ for some } j \geq 0\right\}$$

   Less formally, $\textsc{StripInit0s}(L)$ is the set of all strings obtained by stripping any number of initial 0s from strings in L. For example, if L is the one-string language $\{00011010\}$, then

$$\textsc{StripInit0s}(L) = \{00011010, 0011010, 011010, 11010\}.$$

   Prove that if L is a regular language, then $\textsc{StripInit0s}(L)$ is also a regular language.

3. For each of the following languages L over the alphabet $\Sigma = \{0, 1\}$, give a regular expression that represents L ***and*** describe a DFA that recognizes L.

   (a) $\{0^n w 1^n \mid n > 1 \text{ and } w \in \Sigma^*\}$

   (b) All strings in 0^*10^* whose length is a multiple of 3.

4. The *parity* of a bit-string is 0 if the number of 1 bits is even, and 1 if the number of 1 bits is odd. For example:

$$\mathit{parity}(\varepsilon) = 0 \qquad \mathit{parity}(0010100) = 0 \qquad \mathit{parity}(00101110100) = 1$$

   (a) Give a *self-contained*, formal, recursive definition of the *parity* function. In particular, do ***not*** refer to # or other functions defined in class.

   (b) Let L be an arbitrary regular language. Prove that the language $\mathit{EvenParity}(L) := \{w \in L \mid \mathit{parity}(w) = 0\}$ is also regular.

   (c) Let L be an arbitrary regular language. Prove that the language $\mathit{AddParity}(L) := \{w \bullet \mathit{parity}(w) \mid w \in L\}$ is also regular. For example, if L contains the string 11100 and 11000, then $\mathit{AddParity}(L)$ contains the strings 111001 and 110000.

5. Let L be the language $\left\{0^i 1^j 0^k \;\middle|\; i = j \text{ or } j = k\right\}$.

   (a) ***Prove*** that L is not a regular language.

   (b) Describe a context-free grammar for L.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, check "Yes" if the statement is ***always*** true and "No" 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) No infinite language is regular.

   (b) If L is regular, then for every string $w \in L$, there is a DFA that rejects w.

   (c) If L is context-free and L has a finite fooling set, then L is not regular.

   (d) If L is regular and $L' \cap L = \emptyset$, then L' is not regular.

   (e) The language $\{\mathtt{0}^i\mathtt{1}^j\mathtt{0}^k \mid i + j + k \geq 374\}$ is regular.

   (f) The language $\{\mathtt{0}^i\mathtt{1}^j\mathtt{0}^k \mid i + j - k \geq 374\}$ is regular.

   (g) Let $M = (Q, \{\mathtt{0},\mathtt{1}\}, s, A, \delta)$ be an arbitrary DFA, and let $M' = (Q, \{\mathtt{0},\mathtt{1}\}, s, A, \delta')$ be the DFA obtained from M by changing every $\mathtt{0}$-transition into a $\mathtt{1}$-transition and vice versa. More formally, M and M' have the same states, input alphabet, starting state, and accepting states, but $\delta'(q,\mathtt{0}) = \delta(q,\mathtt{1})$ and $\delta'(q,\mathtt{1}) = \delta(q,\mathtt{0})$. Then $L(M) \cup L(M') = \{\mathtt{0},\mathtt{1}\}^*$.

   (h) Let $M = (Q, \Sigma, s, A, \delta)$ be an arbitrary NFA, and $M' = (Q', \Sigma, s, A', \delta')$ be any NFA obtained from M by deleting some subset of the states. More formally, we have $Q' \subseteq Q$, $A' = A \cap Q'$, and $\delta'(q,a) = \delta(q,a) \cap Q'$ for all $q \in Q'$. Then $L(M') \subseteq L(M)$.

   (i) For every non-regular language L, the language $\left\{\mathtt{0}^{|w|} \;\middle|\; w \in L\right\}$ is also non-regular.

   (j) For every context-free language L, the language $\left\{\mathtt{0}^{|w|} \;\middle|\; w \in L\right\}$ is also context-free.

2. For any language L, define

$$\textsc{StripFinal0s}(L) = \{w \mid w\mathtt{0}^n \in L \text{ for some } n \geq 0\}$$

   Less formally, $\textsc{StripFinal0s}(L)$ is the set of all strings obtained by stripping any number of final $\mathtt{0}$s from strings in L. For example, if L is the one-string language $\{\mathtt{01101000}\}$, then

$$\textsc{StripFinal0s}(L) = \{\mathtt{01101},\ \mathtt{011010},\ \mathtt{0110100},\ \mathtt{01101000}\}.$$

   Prove that if L is a regular language, then $\textsc{StripFinal0s}(L)$ is also a regular language.

3. For each of the following languages L over the alphabet $\Sigma = \{0, 1\}$, give a regular expression that represents L ***and*** describe a DFA that recognizes L.

   (a) $\left\{0^n w 1^n \;\middle|\; n \geq 1 \text{ and } w \in \Sigma^+\right\}$

   (b) All strings in $0^*1^*0^*$ whose length is even.

4. The *parity* of a bit-string is 0 if the number of 1 bits is even, and 1 if the number of 1 bits is odd. For example:

$$\mathit{parity}(\varepsilon) = 0 \qquad \mathit{parity}(0010100) = 0 \qquad \mathit{parity}(00101110100) = 1$$

   (a) Give a *self-contained*, formal, recursive definition of the *parity* function. In particular, do ***not*** refer to # or other functions defined in class.

   (b) Let L be an arbitrary regular language. Prove that the language $\mathit{OddParity}(L) := \{w \in L \mid \mathit{parity}(w) = 1\}$ is also regular.

   (c) Let L be an arbitrary regular language. Prove that the language $\mathit{AddParity}(L) := \{\mathit{parity}(w) \cdot w \mid w \in L\}$ is also regular. For example, if L contains the strings 01110 and 01100, then $\mathit{AddParity}(L)$ contains the strings 101110 and 001100.

5. Let L be the language $\left\{0^i 1^j 0^k \;\middle|\; 2i = k \text{ or } i = 2k\right\}$.

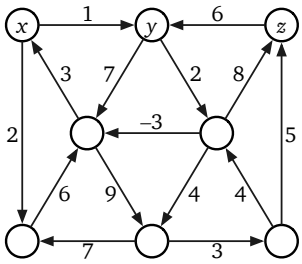
   (a) ***Prove*** that L is not a regular language.

   (b) Describe a context-free grammar for L.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. ***Clearly*** indicate the following structures in the directed graph below, or write NONE if the indicated structure does not exist. Don't be subtle; to indicate a collection of edges, draw a heavy black line along the entire length of each edge.



   (a) A depth-first tree rooted at x.

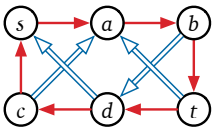
   (b) A breadth-first tree rooted at y.

   (c) A shortest-path tree rooted at z.

   (d) The shortest directed cycle.

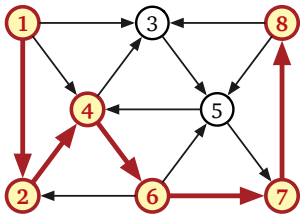
2. Suppose you are given a directed graph G where some edges are red and the remaining edges are blue. Describe an algorithm to find the shortest walk in G from one vertex s to another vertex t in which no three consecutive edges have the same color. That is, if the walk contains two red edges in a row, the next edge must be blue, and if the walk contains two blue edges in a row, the next edge must be red.

   For example, if you are given the graph below (where single arrows are red and double arrows are blue), your algorithm should return the integer 7, because the shortest legal walk from s to t is $s \to a \to b \Rightarrow d \to c \Rightarrow a \to b \to c$.



3. Let G be an arbitrary (*not* necessarily acyclic) directed graph in which every vertex v has an integer label $\ell(v)$. Describe an algorithm to find the longest directed path in G whose vertex labels define an increasing sequence. Assume all labels are distinct.

   For example, given the following graph as input, your algorithm should return the integer 5, which is the length of the increasing path $1 \to 2 \to 4 \to 6 \to 7 \to 8$.



4. Suppose you have an integer array $A[1..n]$ that *used* to be sorted, but Swedish hackers have overwritten k entries of A with random numbers. Because you carefully monitor your system for intrusions, you know *how many* entries of A are corrupted, but not *which* entries or what the values are.

   Describe an algorithm to determine whether your corrupted array A contains an integer x. Your input consists of the array A, the integer k, and the target integer x. For example, if A is the following array, $k = 4$, and $x = 17$, your algorithm should return TRUE. (The corrupted entries of the array are shaded.)

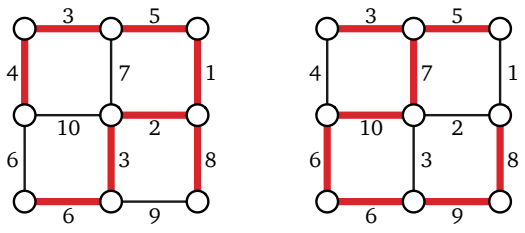
2	3	99	7	11	13	17	19	25	29	31	−5	41	43	47	53	8	61	67	71

   Assume that x is not equal to any of the the corrupted values, and that all n array entries are distinct. Report the running time of your algorithm as a function of n and k. A solution only for the special case $k = 1$ is worth 5 points; a complete solution for arbitrary k is worth 10 points. *[Hint: First consider $k = 0$; then consider $k = 1$.]*

5. Suppose you give one of your interns at Twitbook an undirected graph G with weighted edges, and you ask them to compute a shortest-path tree rooted at a particular vertex. Two weeks later, your intern finally comes back with a spanning tree T of G. Unfortunately, the intern didn't record the shortest-path distances, the direction of the shortest-path edges, or even the source vertex (which you and the intern have both forgotten).

   Describe and analyze an algorithm to determine, given a weighted undirected graph G and a spanning tree T of G, whether T is in fact a *shortest-path* tree in G. Assume all edge weights are non-negative.

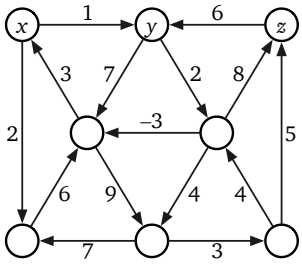
   For example, given the inputs shown below, your algorithm should return TRUE for the example on the left, because T is a shortest-path tree rooted at the upper right vertex of G, but your algorithm should return FALSE for the example on the right.



Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. ***Clearly*** indicate the following structures in the directed graph below, or write NONE if the indicated structure does not exist. Don't be subtle; to indicate a collection of edges, draw a heavy black line along the entire length of each edge.



   (a) A depth-first tree rooted at x.

   (b) A breadth-first tree rooted at y.

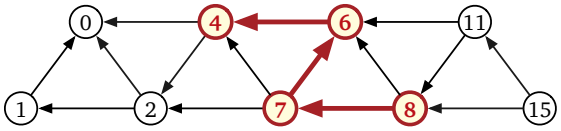
   (c) A shortest-path tree rooted at z.

   (d) The shortest directed cycle.

2. Let G be a ***directed*** graph, where every vertex v has an associated height $h(v)$, and for every edge $u \to v$ we have the inequality $h(u) > h(v)$. Assume all heights are distinct. The *span* of a path from u to v is the height difference $h(u) - h(v)$.

   Describe and analyze an algorithm to find the ***minimum span*** of a path in G with ***at least*** k edges. Your input consists of the graph G, the vertex heights $h(\cdot)$, and the integer k. Report the running time of your algorithm as a function of V, E, and k.

   For example, given the following labeled graph and the integer $k = 3$ as input, your algorithm should return the integer 4, which is the span of the path $8 \to 7 \to 6 \to 4$.



3. Suppose you have an integer array $A[1..n]$ that *used* to be sorted, but Swedish hackers have overwritten k entries of A with random numbers. Because you carefully monitor your system for intrusions, you know *how many* entries of A are corrupted, but not *which* entries or what the values are.

   Describe an algorithm to determine whether your corrupted array A contains an integer x. Your input consists of the array A, the integer k, and the target integer x. For example, if A is the following array, $k = 4$, and $x = 17$, your algorithm should return TRUE. (The corrupted entries of the array are shaded.)

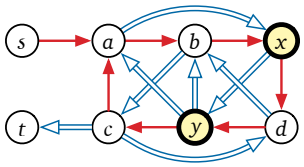
2	3	99	7	11	13	17	19	25	29	31	−5	41	43	47	53	8	61	67	71

   Assume that x is not equal to any of the the corrupted values, and that all n array entries are distinct. Report the running time of your algorithm as a function of n and k. A solution only for the special case $k = 1$ is worth 5 points; a complete solution for arbitrary k is worth 10 points. *[Hint: First consider $k = 0$; then consider $k = 1$.]*

4. Suppose you are given a directed graph G in which every edge is either red or blue, and a subset of the vertices are marked as *special*. A walk in G is *legal* if color changes happen only at special vertices. That is, for any two consecutive edges $u \to v \to w$ in a legal walk, if the edges $u \to v$ and $v \to w$ have different colors, the intermediate vertex v must be special.

   Describe and analyze an algorithm that either returns the length of the shortest legal walk from vertex s to vertex t, or correctly reports that no such walk exists.[1]

   For example, if you are given the following graph below as input (where single arrows are red, double arrows are blue), with special vertices x and y, your algorithm should return the integer 8, which is the length of the shortest legal walk $s \to x \to a \to b \to x \Rightarrow y \Rightarrow b \Rightarrow c \Rightarrow t$. The shorter walk $s \to a \to b \Rightarrow c \Rightarrow t$ is not legal, because vertex b is not special.



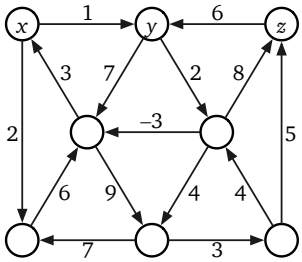
5. Let G be a directed graph with weighted edges, in which every vertex is colored either red, green, or blue. Describe and analyze an algorithm to compute the length of the shortest walk in G that starts at a red vertex, then visits any number of vertices of any color, then visits a green vertex, then visits any number of vertices of any color, then visits a blue vertex, then visits any number of vertices of any color, and finally ends at a red vertex. Assume all edge weights are positive.

[1] If you've read China Miéville's excellent novel *The City & the City*, this problem should look familiar. If you haven't read *The City & the City*, I can't tell you why this problem should look familiar without spoiling the book.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. ***Clearly*** indicate the following structures in the directed graph below, or write NONE if the indicated structure does not exist. Don't be subtle; to indicate a collection of edges, draw a heavy black line along the entire length of each edge.



   (a) A depth-first tree rooted at x.

   (b) A breadth-first tree rooted at y.

   (c) A shortest-path tree rooted at z.

   (d) The shortest directed cycle.

2. After a few weeks of following your uphill-downhill walking path to work, your boss demands that you start showing up to work on time, so you decide to change your walking strategy. Your new goal is to walk to the highest altitude you can (to maximize exercise), while keeping the total length of your walk from home to work below some threshold (to make sure you get to work on time). Describe and analyze an algorithm to compute your new favorite route.

   Your input consists of an ***undirected*** graph G, where each vertex v has a height $h(v)$ and each edge e has a positive length $\ell(e)$, along with a start vertex s, a target vertex t, and a maximum length L. Your algorithm should return the ***maximum height*** reachable by a walk from s to t in G, whose total length is at most L.

   [Hint: This is the same input as HW8 problem 1, but the problem is completely different. In particular, the number of uphill/downhill switches in your walk is irrelevant.]

3. Suppose you have an integer array $A[1..n]$ that *used* to be sorted, but Swedish hackers have overwritten k entries of A with random numbers. Because you carefully monitor your system for intrusions, you know *how many* entries of A are corrupted, but not *which* entries or what the values are.

   Describe an algorithm to determine whether your corrupted array A contains an integer x. Your input consists of the array A, the integer k, and the target integer x. For example, if A is the following array, $k = 4$, and $x = 17$, your algorithm should return TRUE. (The corrupted entries of the array are shaded.)

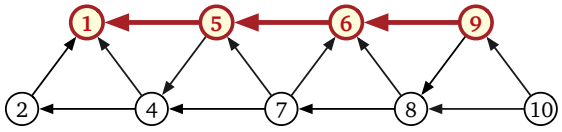
2	3	99	7	11	13	17	19	25	29	31	−5	41	43	47	53	8	61	67	71

   Assume that x is not equal to any of the the corrupted values, and that all n array entries are distinct. Report the running time of your algorithm as a function of n and k. A solution only for the special case $k = 1$ is worth 5 points; a complete solution for arbitrary k is worth 10 points. *[Hint: First consider $k = 0$; then consider $k = 1$.]*

4. Let G be a ***directed*** graph, where every vertex v has an associated height $h(v)$, and for every edge $u \to v$ we have the inequality $h(u) > h(v)$. Assume all heights are distinct. The *span* of a path from u to v is the height difference $h(u) - h(v)$.

   Describe and analyze an algorithm to find the ***maximum span*** of a path in G with at most k edges. Your input consists of the graph G, the vertex heights $h(\cdot)$, and the integer k. Report the running time of your algorithm as a function of V, E, and k.

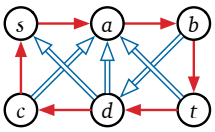
   For example, given the following labeled graph and the integer $k = 3$ as input, your algorithm should return the integer 8, which is the span of the downward path $9 \to 6 \to 5 \to 1$.



   [Hint: This is a very different question from problem 2.]

5. Suppose you are given a directed graph G where some edges are red and the remaining edges are blue, along with two vertices s and t. Describe an algorithm to compute the length of the shortest walk in G from s to t that traverses an even number of red edges and an even number of blue edges. If the walk traverses the same edge multiple times, each traversal counts toward the total for that color.

   For example, if you are given the graph below (where single arrows are red and double arrows are blue), your algorithm should return the integer 6, because the shortest legal walk from s to t is $s \to a \to b \Rightarrow d \Rightarrow a \to b \to t$.



CS/ECE 374 A ✧ Spring 2018

Final Exam

May 8, 2018

Real name:	
NetID:	

Gradescope name:	
Gradescope email:	

- ***Don't panic!***
- 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, please turn off and put away *all* medically unnecessary electronic devices.
- Please clearly print your real name, your university NetID, your Gradescope name, and your Gradescope email address in the boxes above. **We will not scan this page into Gradescope.**
- Please also print **only the name you are using on Gradescope** at the top of every page of the answer booklet, except this cover page. These are the pages we will scan into Gradescope.
- Please do not write outside the black boxes on each page; these indicate the area of the page that the scanner can actually see.
- **Please read the entire exam before writing anything.** Please ask for clarification if any question is unclear.
- **The exam lasts 180 minutes.**
- If you run out of space for an answer, continue on the back of the page, or on the blank pages at the end of this booklet, **but please tell us where to look.** Alternatively, feel free to tear out the blank pages and use them as scratch paper.
- As usual, answering any (sub)problem with "I don't know" (and nothing else) is worth 25% partial credit. **Yes, even for problem 1.** Correct, complete, but suboptimal solutions are *always* worth more than 25%. A blank answer is not the same as "I don't know".
- **Please return your cheat sheets and all scratch paper with your answer booklet.**
- ***Good luck!*** And thanks for a great semester!

Beware of the man who works hard to learn something,
learns it, and finds himself no wiser than before.

He is full of murderous resentment of people who are ignorant
without having come by their ignorance the hard way.

— Bokonon

CS/ECE 374 A ✧ Spring 2018 Final Exam ♫ Problem 1	Gradescope name:

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 ≠ NP.** If there is any other ambiguity or uncertainty, mark the "No" box. For example:

[X] Yes	[ ] No	$2+2=4$
[ ] Yes	[X] No	$x+y=5$
[ ] Yes	[X] No	3SAT can be solved in polynomial time.
[X] Yes	[ ] No	Jeff is not the Queen of England.
[X] Yes	[ ] No	If $P = NP$ then Jeff is the Queen of England.

There are 40 yes/no choices altogether. Each correct choice is worth +½ point; each incorrect choice is worth −¼ point. TO indicate "I don't know", write IDK to the left of the Yes/No boxes; each IDK is worth +1/8 point.

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

[ ] Yes	[ ] No	L is infinite.
[ ] Yes	[ ] No	L^* contains the empty string ε.
[ ] Yes	[ ] No	L^* is decidable.
[ ] Yes	[ ] No	If L is regular then $(L^*)^*$ is regular.
[ ] Yes	[ ] No	If L is the intersection of two decidable languages, then L is decidable.
[ ] Yes	[ ] No	If L is the intersection of two undecidable languages, then L is undecidable.
[ ] Yes	[ ] No	If L is the complement of a regular language, then L^* is regular.
[ ] Yes	[ ] No	If L has an infinite fooling set, then L is undecidable.
[ ] Yes	[ ] No	L is decidable if and only if its complement $\overline{L}$ is undecidable.

(b) Which of the following statements is true for ***every*** directed graph $G = (V, E)$?

Yes	No	$E \neq \emptyset$.
Yes	No	Given the graph G as input, Floyd-Warshall runs in $O(E^3)$ time.
Yes	No	If G has at least one source and at least one sink, then G is a dag.
Yes	No	We can compute a spanning tree of G using whatever-first search.
Yes	No	If the edges of G are weighted, we can compute the shortest path from any node s to any node t in $O(E \log V)$ time using Dijkstra's algorithm.

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

Yes	No	$\{0^m 1 0^n \mid m \leq n\}$
Yes	No	$\{0^m 1 0^n \mid m + n \geq 374\}$
Yes	No	Binary representations of all perfect squares
Yes	No	$\{xy \mid yx \text{ is a palindrome}\}$
Yes	No	$\{\langle M \rangle \mid M \text{ accepts a finite number of non-palindromes}\}$

(d) Which of the following languages are ***decidable***?

Yes	No	Binary representations of all perfect squares
Yes	No	$\{xy \in \{0, 1\}^* \mid yx \text{ is a palindrome}\}$
Yes	No	$\{\langle M \rangle \mid M \text{ accepts the binary representation of every perfect square}\}$
Yes	No	$\{\langle M \rangle \mid M \text{ accepts a finite number of non-palindromes}\}$
Yes	No	The set of all regular expressions that represent the language $\{0, 1\}^*$. (This is a language over the alphabet $\{\emptyset, \varepsilon, 0, 1, *, +, (,)\}$.)

(e) Which of the following languages can be proved undecidable ***using Rice's Theorem***?

Yes	No	$\{\langle M\rangle \mid M \text{ accepts a finite number of strings}\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts both } \langle M\rangle \text{ and } \langle M\rangle^R\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts exactly 374 palindromes}\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts some string } w \text{ after at most } \lvert w\rvert^2 \text{ steps}\}$

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

$$\textsc{Chalmers} := \{\langle M\rangle \mid M \text{ accepts both } \texttt{STEAMED} \text{ and } \texttt{HAMS}\}$$

Professor Skinner suggests a reduction from the standard halting language

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

Specifically, suppose there is a Turing machine Ch that decides Chalmers. Professor Skinner claims that the following algorithm decides Halt.

```
DecideHalt(⟨M⟩#w):
  Encode the following Turing machine:
    AuroraBorealis(x):
      if x = STEAMED or x = HAMS or x = ALBANY
          run M on input w
          return False
      else
          return True
  return Ch(⟨AuroraBorealis⟩)
```

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

Yes	No	If M accepts w, then AuroraBorealis accepts `CLAMS`.
Yes	No	If M rejects w, then AuroraBorealis rejects `UTICA`.
Yes	No	If M rejects w, then AuroraBorealis halts on every input string.
Yes	No	If M accepts w, then Ch accepts $\langle$AuroraBorealis$\rangle$.
Yes	No	DecideHalt decides the language Halt. (That is, Professor Skinner's reduction is actually correct.)
Yes	No	DecideHalt actually runs (or simulates) M.
Yes	No	We could have proved Chalmers is undecidable using Rice's theorem instead of this reduction.

(g) Consider the following pair of languages:

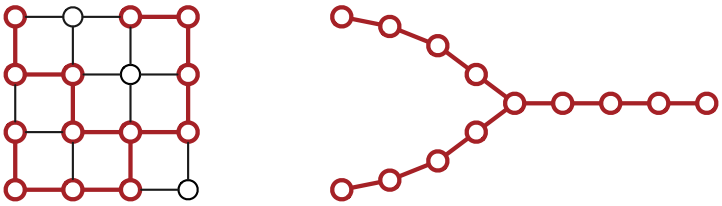
- 3Color $:= \{G \mid G \text{ is a 3-colorable undirected graph}\}$
- Tree $:= \{G \mid G \text{ is a connected acyclic undirected graph}\}$

(For concreteness, assume that in both of these languages, graphs are represented by adjacency matrices.) Which of the following ***must*** be true, assuming P≠NP?

Yes	No	Tree ∪ 3Color is NP-hard.
Yes	No	Tree ∩ 3Color is NP-hard.
Yes	No	3Color is undecidable.
Yes	No	There is a polynomial-time reduction from 3Color to Tree.
Yes	No	There is a polynomial-time reduction from Tree to 3Color.

CS/ECE 374 A ✧ Spring 2018 Final Exam ♫ Problem 2	Gradescope name:

A ***wye*** is an undirected graph that looks like the capital letter Y. More formally, a wye consists of three paths of equal length with one common endpoint, called the *hub*.



This grid graph contains a wye whose paths have length 4.

Prove that the following problem is NP-hard: Given an undirected graph G, what is the largest wye that is a subgraph of G? The three paths of the wye must not share any vertices except the hub, and they must have exactly the same length.

CS/ECE 374 A ✧ Spring 2018 Final Exam ♫ Problem 3	Gradescope name:

Fix the alphabet $\Sigma = \{0, 1\}$. Recall that a *run* in a string $w \in \Sigma^*$ is a maximal non-empty substring in which all symbols are equal. For example, the string `000001000111111101` consists of exactly six runs: `000001000111111110` = `00000` • `1` • `000` • `1111111` • `0` • `1`.

(a) Let L be the set of all strings in Σ^* where every run has odd length. For example, L contains the string `000100000`, but L does not contain the string `00011`.

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

(b) Let L' be the set of all strings in Σ^* that have the same number of even-length runs and odd-length runs. For example, L' does not contain the string `000011101`, because it has three odd-length runs but only one even-length run, but L' does contain the string `0000111011`, because it has two runs of each parity.

Prove that L' is not regular.

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 4**	Gradescope name:

Suppose we want to split an array $A[1..n]$ of integers into k contiguous intervals that partition the sum of the values as evenly as possible. Specifically, define the *cost* of such a partition as the maximum, over all k intervals, of the sum of the values in that interval; our goal is to minimize this cost. Describe and analyze an algorithm to compute the minimum cost of a partition of A into k intervals, given the array A and the integer k as input.

For example, given the array $A = [1, 6, -1, 8, 0, 3, 3, 9, 8, 8, 7, 4, 9, 8, 9, 4, 8, 4, 8, 2]$ and the integer $k = 3$ as input, your algorithm should return the integer 37, which is the cost of the following partition:

$$\Big[\overbrace{1, 6, -1, 8, 0, 3, 3, 9, 8}^{37} \Big| \overbrace{8, 7, 4, 9, 8}^{36} \Big| \overbrace{9, 4, 8, 4, 8, 2}^{35} \Big]$$

The numbers above each interval show the sum of the values in that interval.

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 5**	Gradescope name:

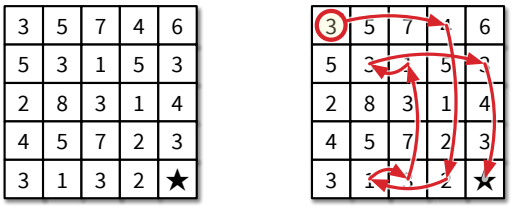
(a) Fix the alphabet $\Sigma = \{0, 1\}$. Describe and analyze an efficient algorithm for the following problem: Given an NFA M over Σ, does M accept at least one string? Equivalently, is $L(M) \neq \emptyset$?

(b) Recall from Homework 10 that deciding whether a given NFA accepts *every* string is NP-hard. Also recall that the complement of every regular language is regular; thus, for any NFA M, there is another NFA M' such that $L(M') = \Sigma^* \setminus L(M)$. So why doesn't your algorithm from part (a) imply that P=NP?

CS/ECE 374 A ✧ Spring 2018 Final Exam ♫ Problem 6	Gradescope name:

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.



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

(scratch paper)

(scratch paper)

(scratch paper)

(scratch paper)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

SteamedHams: Aurora borealis? At this time of year, at this time of day, in this part of the country, localized entirely within your kitchen? May I see it?

CS/ECE 374 A ✧ Spring 2018

Final Exam

May 8, 2018

Real name:	
NetID:	

Gradescope name:	
Gradescope email:	

- ***Don't panic!***
- 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, please turn off and put away *all* medically unnecessary electronic devices.
- Please clearly print your real name, your university NetID, your Gradescope name, and your Gradescope email address in the boxes above. **We will not scan this page into Gradescope.**
- Please also print **only the name you are using on Gradescope** at the top of every page of the answer booklet, except this cover page. These are the pages we will scan into Gradescope.
- Please do not write outside the black boxes on each page; these indicate the area of the page that the scanner can actually see.
- **Please read the entire exam before writing anything.** Please ask for clarification if any question is unclear.
- **The exam lasts 180 minutes.**
- If you run out of space for an answer, continue on the back of the page, or on the blank pages at the end of this booklet, **but please tell us where to look.** Alternatively, feel free to tear out the blank pages and use them as scratch paper.
- As usual, answering any (sub)problem with "I don't know" (and nothing else) is worth 25% partial credit. **Yes, even for problem 1.** Correct, complete, but suboptimal solutions are *always* worth more than 25%. A blank answer is not the same as "I don't know".
- **Please return your cheat sheets and all scratch paper with your answer booklet.**
- ***Good luck!*** And thanks for a great semester!

Beware of the man who works hard to learn something,
learns it, and finds himself no wiser than before.

He is full of murderous resentment of people who are ignorant
without having come by their ignorance the hard way.

— Bokonon

CS/ECE 374 A ✧ Spring 2018 Final Exam ♫ Problem 1	Gradescope name:

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 ≠ NP.** If there is any other ambiguity or uncertainty, mark the "No" box. For example:

[X] Yes	[ ] No	$2+2=4$
[ ] Yes	[X] No	$x+y=5$
[ ] Yes	[X] No	3SAT can be solved in polynomial time.
[X] Yes	[ ] No	Jeff is not the Queen of England.
[X] Yes	[ ] No	If $P = NP$ then Jeff is the Queen of England.

There are 40 yes/no choices altogether. Each correct choice is worth +½ point; each incorrect choice is worth −¼ point. TO indicate "I don't know", write IDK to the left of the Yes/No boxes; each IDK is worth +1/8 point.

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

[ ] Yes	[ ] No	L is finite.
[ ] Yes	[ ] No	L^* contains the empty string ε.
[ ] Yes	[ ] No	L^* is decidable.
[ ] Yes	[ ] No	If L is regular then $\Sigma^* \setminus L^*$ is regular.
[ ] Yes	[ ] No	If L is the intersection of two decidable languages, then L is decidable.
[ ] Yes	[ ] No	If L is the intersection of two undecidable languages, then L is undecidable.
[ ] Yes	[ ] No	If L^* is the complement of a regular language, then L is regular.
[ ] Yes	[ ] No	If L is undecidable, then every fooling set for L is infinite.
[ ] Yes	[ ] No	L is decidable if and only if its complement $\overline{L}$ is undecidable.

(b) Which of the following statements is true for ***every*** directed graph $G = (V, E)$?

Yes	No	$E \neq \emptyset$.
Yes	No	Given the graph G as input, Floyd-Warshall runs in $O(E^3)$ time.
Yes	No	If G has at least one source and at least one sink, then G is a dag.
Yes	No	We can compute a spanning tree of G using whatever-first search.
Yes	No	If the edges of G are weighted, we can compute the shortest path from any node s to any node t in $O(E \log V)$ time using Dijkstra's algorithm.

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

Yes	No	$\{0^m 1 0^n \mid m \geq n\}$
Yes	No	$\{0^m 1 0^n \mid m - n \geq 374\}$
Yes	No	Binary representations of all integers divisible by 374
Yes	No	$\{xy \mid yx \text{ is a palindrome}\}$
Yes	No	$\{\langle M \rangle \mid M \text{ accepts a finite number of non-palindromes}\}$

(d) Which of the following languages are ***decidable***?

Yes	No	Binary representations of all integers divisible by 374
Yes	No	$\{xy \in \{0, 1\}^* \mid yx \text{ is a palindrome}\}$
Yes	No	$\{\langle M \rangle \mid M \text{ accepts the binary representation of every integer divisible by 374}\}$
Yes	No	$\{\langle M \rangle \mid M \text{ accepts a finite number of non-palindromes}\}$
Yes	No	The set of all regular expressions that represent the language $\{0, 1\}^*$. (This is a language over the alphabet $\{\emptyset, \varepsilon, 0, 1, *, +, (,)\}$.)

(e) Which of the following languages can be proved undecidable ***using Rice's Theorem***?

Yes	No	$\{\langle M\rangle \mid M \text{ accepts an infinite number of strings}\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts either } \langle M\rangle \text{ or } \langle M\rangle^R\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts } \texttt{001100} \text{ but rejects } \texttt{110011}\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts some string } w \text{ after at most } \lvert w\rvert^2 \text{ steps}\}$

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

$$\textsc{Chalmers} := \{\langle M\rangle \mid M \text{ accepts both } \texttt{STEAMED} \text{ and } \texttt{HAMS}\}$$

Professor Skinner suggests a reduction from the standard halting language

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

Specifically, suppose there is a Turing machine Ch that decides Chalmers. Professor Skinner claims that the following algorithm decides Halt.

```
DecideHalt(⟨M⟩#w):
  Encode the following Turing machine:
      AuroraBorealis(x):
        if x = STEAMED or x = HAMS or x = ALBANY
            run M on input w
            return True
        else
            return False
  return Ch(⟨AuroraBorealis⟩)
```

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

Yes	No	If M accepts w, then AuroraBorealis accepts `CLAMS`.
Yes	No	If M rejects w, then AuroraBorealis rejects `UTICA`.
Yes	No	If M hangs on w, then AuroraBorealis accepts every input string.
Yes	No	If M accepts w, then Ch accepts $\langle$AuroraBorealis$\rangle$.
Yes	No	DecideHalt decides the language Halt. (That is, Professor Skinner's reduction is actually correct.)
Yes	No	DecideHalt actually runs (or simulates) M.
Yes	No	We could have proved Chalmers is undecidable using Rice's theorem instead of this reduction.

(g) Consider the following pair of languages:

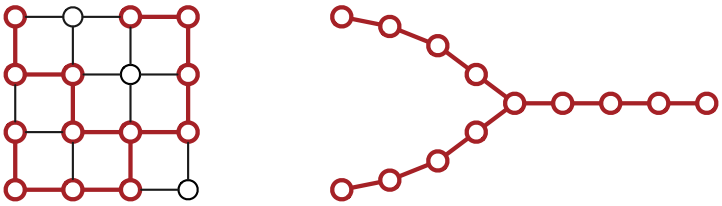
- 3Color $:= \{G \mid G \text{ is a 3-colorable undirected graph}\}$
- Tree $:= \{G \mid G \text{ is a connected acyclic undirected graph}\}$

(For concreteness, assume that in both of these languages, graphs are represented by adjacency matrices.) Which of the following ***must*** be true, assuming P≠NP?

Yes	No	Tree ∪ 3Color is NP-hard.
Yes	No	Tree ∩ 3Color is NP-hard.
Yes	No	3Color is undecidable.
Yes	No	There is a polynomial-time reduction from 3Color to Tree.
Yes	No	There is a polynomial-time reduction from Tree to 3Color.

CS/ECE 374 A ✧ Spring 2018 Final Exam ♫ Problem 2	Gradescope name:

A ***wye*** is an undirected graph that looks like the capital letter Y. More formally, a wye consists of three paths of equal length with one common endpoint, called the *hub*.



This grid graph contains a wye whose paths have length 4.

Prove that the following problem is NP-hard: Given an undirected graph G, what is the largest wye that is a subgraph of G? The three paths of the wye must not share any vertices except the hub, and they must have exactly the same length.

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 3**	Gradescope name:

Fix the alphabet $\Sigma = \{0, 1\}$. Recall that a *run* in a string $w \in \Sigma^*$ is a maximal non-empty substring in which all symbols are equal. For example, the string `000001000111111101` consists of exactly six runs: `000001000111111110` = `00000` • `1` • `000` • `1111111` • `0` • `1`.

(a) Let L be the set of all strings in Σ^* that contains at least one run whose length is divisible by 3. For example, L contains the string `00111111000`, but L does not contain the string `1000011`.

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

(b) Let L' be the set of all strings in Σ^* that have the same number of even-length runs and odd-length runs. For example, L' does not contain the string `000011101`, because it has three odd-length runs but only one even-length run, but L' does contain the string `0000111011`, because it has two runs of each parity.

Prove that L' is not regular.

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 4**	Gradescope name:

Suppose we want to split an array $A[1..n]$ of integers into k contiguous intervals that partition the sum of the values as evenly as possible. Specifically, define the *quality* of such a partition as the minimum, over all k intervals, of the sum of the values in that interval; our goal is to maximize quality. Describe and analyze an algorithm to compute the maximum quality of a partition of A into k intervals, given the array A and the integer k as input.

For example, given the array $A = [1, 6, -1, 8, 0, 3, 3, 9, 8, 8, 7, 4, 9, 8, 9, 4, 8, 4, 8, 2]$ and the integer $k = 3$ as input, your algorithm should return the integer 35, which is the quality of the following partition:

$$\Big[\overbrace{1, 6, -1, 8, 0, 3, 3, 9, 8}^{37} \Big| \overbrace{8, 7, 4, 9, 8}^{36} \Big| \overbrace{9, 4, 8, 4, 8, 2}^{35} \Big]$$

The numbers above each interval show the sum of the values in that interval.

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 5**	Gradescope name:

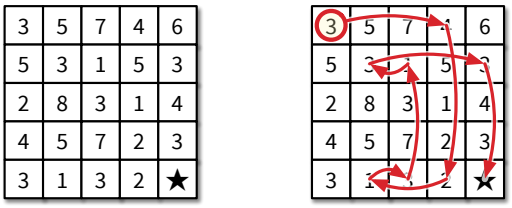
(a) Fix the alphabet $\Sigma = \{0, 1\}$. Describe and analyze an efficient algorithm for the following problem: Given a DFA M over Σ, does M reject *any* string? Equivalently, is $L(M) \neq \Sigma^*$?

(b) Recall from Homework 10 that the corresponding problem for NFAs is NP-hard. But any NFA can be transformed into equivalent DFA using the incremental subset construction. So why doesn't your algorithm from part (a) imply that P=NP?

CS/ECE 374 A ✧ Spring 2018 Final Exam ♫ Problem 6	Gradescope name:

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.



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

(scratch paper)

(scratch paper)

(scratch paper)

(scratch paper)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

SteamedHams: Aurora borealis? At this time of year, at this time of day, in this part of the country, localized entirely within your kitchen? May I see it?