CS/ECE 374 A ✧ Fall 2019

Homework 0

Due Tuesday, September 3, 2019 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 precise 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 infamous Scottish computational arborist Seòras na Coille has a favorite 26-node binary tree, whose nodes are labeled with the letters of the English alphabet. Preorder and inorder traversals of his tree yield the following letter sequences:

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

   (a) List the nodes in Professor na Coille's tree according to a postorder traversal.

   (b) Draw Professor na Coille's tree.

   You do *not* need to prove that your answers are correct. *[Hint: It may be easier to write a short Python program than to figure this out by hand.]*

2. For any string $w \in \{0,1\}^*$, let $\mathit{sort}(w)$ denote the string obtained by sorting the characters in w. For example, $\mathit{sort}(010101) = 000111$. The *sort* function can be defined recursively as follows:

$$\mathit{sort}(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ 0 \cdot \mathit{sort}(x) & \text{if } w = 0x \\ \mathit{sort}(x) \bullet 1 & \text{if } w = 1x \end{cases}$$

   (a) Prove that $\#(0, \mathit{sort}(w)) = \#(0, w)$ for every string $w \in \{0,1\}^*$.

   (b) Prove that $\mathit{sort}(w \bullet 1) = \mathit{sort}(w) \bullet 1$ for every string $w \in \{0,1\}^*$.

   (c) Prove that $\mathit{sort}(\mathit{sort}(w)) = \mathit{sort}(w)$ for every string $w \in \{0,1\}^*$.

   Think about these two problems on your own; do not submit solutions:

   (d) Prove that $x \bullet 0 \neq y \bullet 1$ for all strings $x, y \in \{0,1\}^*$.

   (e) Prove that $\mathit{sort}(w) \neq x \bullet 10 \bullet y$, for all strings $w, x, y \in \{0,1\}^*$.

   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. Your proofs for later parts of this problem can assume earlier parts even if you don't prove them. Otherwise, your proofs must be formal and self-contained.

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

   - The empty string ε is in L.
   - For any strings x in L, the strings $0x1$ and $1x0$ are also in L.
   - For any two *nonempty* strings x and y in L, the string $x \bullet y$ is also in L.
   - These are the only strings in L.

   This problem asks you to prove that L is the set of all strings w where the number of 0s is equal to the number of 1s. More formally, for any string w, let $\Delta(w) = \#(1, w) - \#(0, w)$, or equivalently,

$$\Delta(w) = \begin{cases} 0 & \text{if } w = \varepsilon \\ \Delta(x) - 1 & \text{if } w = 0x \\ \Delta(x) + 1 & \text{if } w = 1x \end{cases}$$

   (a) Prove that the string 11011100101000 is in L.

   (b) Prove that $\Delta(w) = 0$ for every string $w \in L$.

   (c) Prove that L contains every string $w \in \{0, 1\}^*$ such that $\Delta(w) = 0$.

   You can assume the following properties of the Δ function, for all strings w and z.

   - Addition: $\Delta(wz) = \Delta(w) + \Delta(z)$.
   - Downward interpolation: If $\Delta(wz) > 0$ and $\Delta(z) < 0$, then there are strings x and y such that $w = xy$ and $\Delta(yz) = 0$.
   - Upward interpolation: If $\Delta(wz) < 0$ and $\Delta(z) > 0$, then there are strings x and y such that $w = xy$ and $\Delta(yz) = 0$.

   The interpolation properties are a type of "intermediate value theorem". **Think about how to prove these properties yourself.**

   You can also assume any other result proved in class, in lab, or in the lecture notes. Otherwise, your proofs must be formal and self-contained.

Solved Problems

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!*

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

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

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, if $w = axa$ for some symbol $a \in \Sigma$ and some palindrome $x \in P$,

then

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

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 ✧ Fall 2019

Homework 1

Due Tuesday, September 10, 2019 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 regular expression is correct.

   (a) All strings except `010`.

   (b) All strings that contain the substring `010`.

   (c) All strings that contain the subsequence `010`.

   (d) All strings that do not contain the substring `010`.

   (e) All strings that do not contain the subsequence `010`.

   (The technical terms "substring" and "subsequence" are defined in the lecture notes.)

2. Let L be the set of all strings in $\{0,1\}^*$ that contain *at least two* occurrences of the substring `010`.

   (a) Give a regular expression for L, and briefly argue why your expression is correct.

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

   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. (See the standard DFA rubric for more details.)

   Argue that your DFA is correct by explaining what each state in your DFA *means*. Drawings or formal descriptions without English explanations will receive no credit, even if they are correct.

   [Hint: The shortest string in L has length 5.]

3. Let L denote the set of all strings $w \in \{0, 1\}^*$ that satisfy *at most two* of the following conditions:

   - The substring 01 appears in w an odd number of times.
   - $\#(1, w)$ is divisible by 3.
   - The binary value of w is *not* a multiple of 7.

   For example: The string 00100101 satisfies all three conditions, so 00100011 is ***not*** in L, and the empty string ε satisfies only the second condition, so $\varepsilon \in L$. (01 appears in ε zero times, and the binary value of ε is 0, because what else could it be?)

   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 accepting states A, and the transition function δ. Do not attempt to *draw* your DFA; the smallest DFA for this language has 84 states, which is *way* too many for a drawing to be understandable.

   Argue that your machine is correct 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.

Solved problem

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

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

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

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

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

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

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

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

- The opening `/*` or `//` of a comment must not be inside a string literal (`"`$\cdots$`"`) 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 `"/*\\\"*/"/*"/*\"/*"*/` is a valid string literal (representing the 5-character string `/*\"\*/`, which is itself a valid block comment!) followed immediately by a valid block comment. ***For this homework question, just pretend that the characters `'`, `"`, and `\` don't exist.***

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

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

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

Solution:

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

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 $(\mathsf{A} + \diamond + \Ldsh)$ or by the closing slash of the comment. ■

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

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

Solution:

$$\left(\diamond + \Ldsh + \texttt{//}(\texttt{/} + \star + \mathsf{A} + \diamond)^*\Ldsh + \texttt{/}\star(\texttt{/} + \mathsf{A} + \diamond + \Ldsh + \star\star^*(\mathsf{A} + \diamond + \Ldsh))^* \star\star^*\texttt{/}\right)^*$$

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

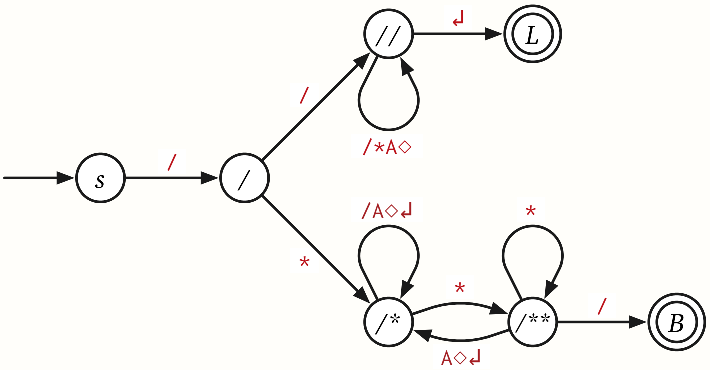
Rubric: Standard regular expression rubric. This is not the only correct solution.

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 *transcription*; 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 more complex than necessary may be penalized. Regular expressions that are *significantly* too complex may get no credit at all. On the other hand, minimal regular expressions are *not* required for full credit.

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

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



The states are labeled mnemonically as follows:

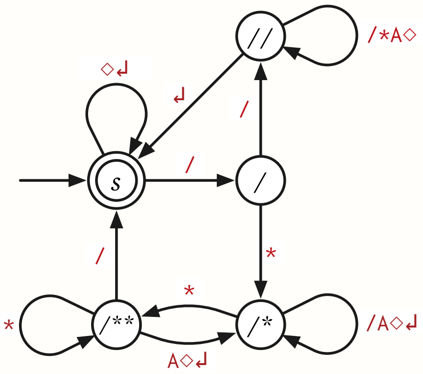
- *s* — We have not read anything.
- / — We just read the initial /.
- // — We are reading a line comment.
- *L* — We have just read a complete line comment.
- /* — We are reading a block comment, and we did not just read a * after the opening /*.
- /** — We are reading a block comment, and we just read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

■

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

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

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



The states are labeled mnemonically as follows:

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

■

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

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.
  - **Drawings:** Use an arrow from nowhere to indicate s, and doubled circles to indicate accepting states A. If $A = \varnothing$, say so explicitly. If your drawing omits a junk/trash/reject state, say so explicitly. **Draw neatly!** If we can't read your solution, we can't give you credit for it.
  - **Text descriptions:** You can describe the transition function either using a 2d array, using mathematical notation, or using an algorithm.
  - **Product constructions:** You must give a complete description of each the DFAs you are combining (as either drawings, text, or recursive products), together with the accepting states of the product DFA.
- **Homework only:** 4 points for *briefly* explaining the purpose of each state *in English*. This is how you argue 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 both necessary and 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 accepting 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.
- DFAs that are more complex than necessary may be penalized. DFAs that are *significantly* more complex than necessary may get no credit at all. On the other hand, *minimal* DFAs are *not* required for full credit, unless the problem explicitly asks for them.
- Half credit for describing an NFA when the problem asks for a DFA.

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

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

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

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

Prove that the reversal of every regular language is regular.

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

Assume for any proper subexpression s of s that there is a regular expression s' such that $L(s') = L(s)^R$.

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

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

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

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

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

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

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

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

Set $\boldsymbol{r' := s' + t'}$; then we have

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

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

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

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

Rubric: Standard induction rubric!!

CS/ECE 374 A ✧ Fall 2019

Homework 2

Due Tuesday, September 17, 2019 at 8pm

---

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

   (a) $\{0^m 1^n \mid m > n\}$

   (b) $\left\{w \in (0+1)^* \;\middle|\; \#(0,w)/\#(1,w) \text{ is an integer}\right\}$ *[Hint: $n/0$ is never an integer.]*

   (c) The set of all palindromes in $(0+1)^*$ whose length is divisible by 7.

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) $(0+11)^*(00+1)^*$

   (b) $(((0^*+1)^*+0)^*+1)^*$

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

   (a) $\left\{0^a 1^b 0^c \;\middle|\; (a \le b+c \text{ and } b \le a+c) \text{ or } c \le a+b\right\}$

   (b) $\left\{0^a 1^b 0^c \;\middle|\; a \le b+c \text{ and } (b \le a+c \text{ or } c \le a+b)\right\}$

   (c) $\left\{wxw^R \;\middle|\; w, x \in \Sigma^+\right\}$

   (d) $\left\{ww^R x \;\middle|\; w, x \in \Sigma^+\right\}$

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

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

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

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

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

   We conclude that $\mathtt{0}x \in L_a$ if and only if $x \in \mathtt{1}^*$. A similar argument implies that $\mathtt{1}x \in L_a$ if and only if $x \in \mathtt{0}^*$. 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 $(\mathtt{0}+\mathtt{1}+\mathtt{2})^*$ 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 $F = (\mathtt{012})^+$ is a fooling set for L_b.

   Let x and y be arbitrary distinct strings in F.

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

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

   Let z be the suffix $(\mathtt{210})^i$.

   - $xz = (\mathtt{012})^i(\mathtt{210})^i$ is a palindrome of length $6i \ge 2$, so $xz \notin L_b$.
   - $yz = (\mathtt{012})^j(\mathtt{210})^i$ has no palindrome prefixes except ε and $\mathtt{0}$, 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 $(\mathtt{0}+\mathtt{1})^*$ 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 $F = (\mathtt{001101})^+$ is a fooling set for L_c.

Let x and y be arbitrary distinct strings in F.

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

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

Let z be the suffix $(\mathtt{101100})^i$.

- $xz = (\mathtt{001101})^i(\mathtt{101100})^i$ is a palindrome of length $12i \geq 2$, so $xz \notin L_b$.
- $yz = (\mathtt{001101})^j(\mathtt{101100})^i$ has no palindrome prefixes except ε and $\mathtt{0}$, 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 $(\mathtt{0}+\mathtt{1})^*$ 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

$$(\mathtt{0}+\mathtt{1})^*(\mathtt{000}+\mathtt{010}+\mathtt{101}+\mathtt{111}+\mathtt{0110}+\mathtt{1001})(\mathtt{0}+\mathtt{1})^*$$

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 $\mathtt{0}$ or $\mathtt{1}$ to any of the underlined strings creates a palindrome suffix of length 3 or 4.

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

■

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 ✧ Fall 2019

Homework 3

Due Tuesday, September 24, 2018 at 8pm

This is the last homework before Midterm 1.

---

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

   (a) $\{\mathtt{0}^m \mathtt{1}^n \mid m > n\}$

   (b) The set of all palindromes in $(\mathtt{0} + \mathtt{1})^*$ whose length is divisible by 7.

   (c) $\left\{\mathtt{0}^a \mathtt{1}^b \;\middle|\; a \neq 2b \text{ and } b \neq 2a\right\}$

   [Hint: You proved that the first two languages are non-regular in HW2.1(a) and HW2.1(c). The language described in HW2.1(b) is not even context-free!]

2. For any string w, let $\mathit{contract}(w)$ denote the string obtained by collapsing each maximal substring of equal symbols to one symbol. For example:

$$\begin{aligned}
\mathit{contract}(\mathtt{010101}) &= \mathtt{010101}\\
\mathit{contract}(\mathtt{001110}) &= \mathtt{010}\\
\mathit{contract}(\mathtt{111111}) &= \mathtt{1}\\
\mathit{contract}(\mathtt{1}) &= \mathtt{1}\\
\mathit{contract}(\varepsilon) &= \varepsilon
\end{aligned}$$

   Prove that for every regular language L over the alphabet $\{\mathtt{0}, \mathtt{1}\}$, the following languages are also regular:

   (a) $\mathit{contract}(L) = \{\mathit{contract}(w) \mid w \in L\}$

   (b) $\mathit{contract}^{-1}(L) = \{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid \mathit{contract}(w) \in L\}$

3. For any string w, let $\mathit{oneswap}(w)$ be the set of all strings obtained by swapping exactly one pair of adjacent symbols in w. For example:

$$\begin{aligned}
\mathit{oneswap}(\mathtt{010101}) &= \{\mathtt{100101}, \mathtt{001101}, \mathtt{011001}, \mathtt{010011}, \mathtt{010110}\}\\
\mathit{oneswap}(\mathtt{001110}) &= \{\mathtt{001110}, \mathtt{010110}, \mathtt{001101}\}\\
\mathit{oneswap}(\mathtt{111111}) &= \{\mathtt{111111}\}\\
\mathit{oneswap}(\mathtt{1}) &= \varnothing\\
\mathit{oneswap}(\varepsilon) &= \varnothing
\end{aligned}$$

   For any language L, define a new language $\mathit{oneswap}(L)$ as follows:

$$\mathit{oneswap}(L) := \bigcup_{w \in L} \mathit{oneswap}(w)$$

   Prove that if L is a regular language over the alphabet $\{\mathtt{0}, \mathtt{1}\}$, the language $\mathit{oneswap}(L)$ is also regular.

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:

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

$$\begin{aligned} \delta'(s', \varepsilon) &= \{(s,h,h) \mid h \in Q\} \\ \delta'(s', a) &= \varnothing \end{aligned}$$

$$\begin{aligned} \delta'((p,h,q), \varepsilon) &= \varnothing \\ \delta'((p,h,q), a) &= \big\{\big(\delta(p,a), h, \delta(q,a)\big)\big\} \end{aligned}$$

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 $\varnothing$, or $\{\varepsilon\}$, or $\mathtt{0}^*$, or Σ^*.

CS/ECE 374 A ✧ Fall 2019

Homework 4

Due Tuesday, October 8, 2019 at 8pm

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

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

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

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

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

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

2. In the lab on Wednesday, we developed an algorithm to compute the median of the union of two sorted arrays size n in $O(\log n)$ time.

   But now suppose we are given *three* sorted arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$. Describe and analyze an algorithm to compute the median of $A \cup B \cup C$ in $O(\log n)$ time. (You can assume the arrays contain $3n$ distinct integers.)

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

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

3. 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 order from left to right, 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 $j \leftarrow 1$ to $n$
      ⟨⟨Find the left target $L[j]$ for hero $j$⟩⟩
      $L[j] \leftarrow$ None
      for $i \leftarrow 1$ to $j-1$
          if $Ht[i] > Ht[j]$
              $L[j] \leftarrow i$
      ⟨⟨Find the right target $R[j]$ for hero $j$⟩⟩
      $R[j] \leftarrow$ None
      for $k \leftarrow n$ down to $j+1$
          if $Ht[k] > Ht[j]$
              $R[j] \leftarrow k$
  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).

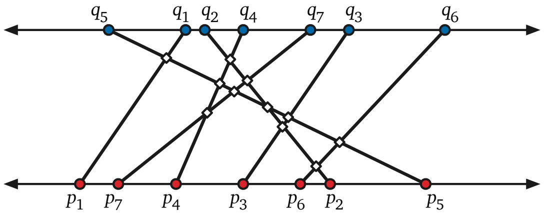
   (c) Alas, Dr. Metaphor's diabolical plan is successful. At the prearranged signal, all the heroes simultaneously shoot their targets, and all targets fall over the cliff, apparently dead. Metaphor repeats his dastardly experiment over and over; after each massacre, he forces the remaining heroes to choose 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.[3]

   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 $\boldsymbol{O(n)}$ time.

[3] In the thrilling final act, Retcon the Squirrel, the last surviving member of the Hero Team/Group/Society (played by Tom Baker, of course), saves everyone by traveling back in time and retroactively replacing the other $n-1$ heroes with lifelike balloon sculptures. So, yeah, it's basically *Avengers: Endgame* meets *Doom Patrol*.

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($A[1..n]$):
    $count \gets 0$
    $total \gets 0$
    for $i \gets 1$ to $n$
        if $A[i]$ is red
            $count \gets count + 1$
        else
            $total \gets total + count$
    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($A[1..n], m$):
    $i \gets 1$;  $j \gets m + 1$;  $count \gets 0$;  $total \gets 0$
    for $k \gets 1$ to $n$
        if $j > n$
            $B[k] \gets A[i]$;  $i \gets i + 1$;  $total \gets total + count$
        else if $i > m$
            $B[k] \gets A[j]$;  $j \gets j + 1$;  $count \gets count + 1$
        else if $A[i] < A[j]$
            $B[k] \gets A[i]$;  $i \gets i + 1$;  $total \gets total + count$
        else
            $B[k] \gets A[j]$;  $j \gets j + 1$;  $count \gets count + 1$
    for $k \gets 1$ to $n$
        $A[k] \gets B[k]$
    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):
  $i \leftarrow 1$; $j \leftarrow m+1$; $total \leftarrow 0$
  for $k \leftarrow 1$ to $n$
      if $j > n$
          $B[k] \leftarrow A[i]$; $i \leftarrow i+1$; $total \leftarrow total + \mathbf{j - m - 1}$
      else if $i > m$
          $B[k] \leftarrow A[j]$; $j \leftarrow j+1$
      else if $A[i] < A[j]$
          $B[k] \leftarrow A[i]$; $i \leftarrow i+1$; $total \leftarrow total + \mathbf{j - m - 1}$
      else
          $B[k] \leftarrow A[j]$; $j \leftarrow j+1$
  for $k \leftarrow 1$ to $n$
      $A[k] \leftarrow B[k]$
  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 ✧ Fall 2019

Homework 5

Due Tuesday, October 15, 2019 at 8pm

1. Farmers Boggis, Bunce, and Bean have set up an obstacle course for Mr. Fox. The course consists of a long row of booths, each with a number painted on the front with bright red paint. Formally, Mr. Fox is given an array $A[1..n]$, where $A[i]$ is the number painted on the front of the ith booth. Each number $A[i]$ could be positive, negative, or zero. Everyone agrees with the following rules:

   - Mr. Fox must visit all the booths in order from 1 to n.
   - At each booth, Mr. Fox must say one word: either "Ring!" or "Ding!"
   - If Mr. Fox says "Ring!" at the ith booth, he earns a reward of $A[i]$ chickens. (If $A[i] < 0$, Mr. Fox pays a penalty of $-A[i]$ chickens.)
   - If Mr. Fox says "Ding!" at the ith booth, he pays a penalty of $A[i]$ chickens. (If $A[i] < 0$, Mr. Fox earns a reward of $-A[i]$ chickens.)
   - Mr. Fox is forbidden to say the same word more than three times in a row. For example, if he says "Ring!" at booths 6, 7, and 8, then he must say "Ding!" at booth 9.
   - All accounts will be settled at the end, after Mr. Fox visits every booth and the umpire calls "Hot box!" Mr. Fox does not actually have to carry chickens (or anti-chickens) through the obstacle course.
   - Finally, if Mr. Fox violates any of the rules, or if he ends the obstacle course owing the farmers chickens, the farmers will shoot him.

   Describe and analyze an algorithm to compute the largest number of chickens that Mr. Fox can earn by running the obstacle course, given the array $A[1..n]$ of numbers as input. *[Hint: Watch out for the burning pine cone!]*

2. Recall that a *supersequence* of a string w is any string obtained from w by inserting zero or more symbols. For example, the strings STRING, STIRRING, and MISTERFINNIGAN are all supersequences of the string STRING.

   (a) Recall that a *palindrome* is a string that is equal to its reversal, like the empty string, A, HANNAH, or AMANAPLANACATACANALPANAMA. Describe an algorithm to compute the length of the shortest palindrome supersequence of a given string.

   (b) A *dromedrome* is an even-length string whose first half is equal to its second half, like the empty string, AA, ACKACK, or AMANAPLANAMANAPLAN. Describe an algorithm to compute the length of the shortest dromedrome supersequence of a given string.

   For example, given the string SUPERSEQUENCE as input, your algorithm for part (a) should return 21 (the length of SUPECNRSEQUQESRNCEPUS), and your algorithm for part (b) should return 20 (the length of SEQUPERNCESEQUPERNCE). The input to both algorithms is an array $A[1..n]$ representing a string.

3. Suppose you are given a DFA M with k states for Jeff's favorite regular language $L \subseteq (\mathtt{0}+\mathtt{1})^*$.

   (a) Describe and analyze an algorithm that decides whether a given bit-string belongs to the language L^*.

   (b) Describe and analyze an algorithm that partitions a given bit-string into as many substrings as possible, such that L contains every substring in the partition. Your algorithm should return only the number of substrings, not their actual positions. (In light of your algorithm from part (a), you can assume that an appropriate partition exists.)

   For example, suppose L is the set of all bit-strings that start and end with `1` and whose length is *not* divisible by 3.[1] Then given the input string `10111110000110101101111101`, your algorithm for part (a) should return TRUE, and your algorithm for part (b) should return the integer 5, which is the length of the following partition:

   $$\mathtt{1011} \bullet \mathtt{1} \bullet \mathtt{10000110101} \bullet \mathtt{1011} \bullet \mathtt{1101}$$

   The input to both algorithms consists of (some reasonable representation of) the DFA M and an array $A[1..n]$ of bits. Express the running time of your algorithms as functions of both k (the number of states in M) and n (the length of the input string).

   *[Hint: Do **not** try to build a DFA for L^*.]*

[1] This is not actually Jeff's favorite regular language.

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.

   BANANAANANAS BANANAANANAS BANANAANANAS

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

   PRODGYRNAMAMMIINCG DYPRONGARMAMMICING

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

Solution: We define a boolean function $\mathit{Shuf}(i,j)$, 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:

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

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

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

```
Shuffle?($A[1..m]$, $B[1..n]$, $C[1..m+n]$):
    $Shuf[0,0] \gets$ True
    for $j \gets 1$ to $n$
        $Shuf[0,j] \gets Shuf[0,j-1] \wedge (B[j] = C[j])$
    for $i \gets 1$ to $n$
        $Shuf[i,0] \gets Shuf[i-1,0] \wedge (A[i] = B[i])$
        for $j \gets 1$ to $n$
            $Shuf[i,j] \gets$ False
            if $A[i] = C[i+j]$
                $Shuf[i,j] \gets Shuf[i,j] \vee Shuf[i-1,j]$
            if $B[i] = C[i+j]$
                $Shuf[i,j] \gets Shuf[i,j] \vee Shuf[i,j-1]$
    return $Shuf[m,n]$
```

The algorithm runs in $\boldsymbol{O(mn)}$ ***time***. ■

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

Standard dynamic programming rubric. For problems worth 10 points:

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 point for a clear English description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) **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 specifically 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 ***do*** 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 ✧ Fall 2019

Homework 6

Due Tuesday, October 22, 2019 at 8pm

---

1. A non-empty sequence $S[1\,..\,\ell]$ of positive integers is called a ***perfect ruler sequence*** if it satisfies the following conditions:

   - The length of S is one less than a power of 2; that is, $\ell = 2^k - 1$ for some integer k.
   - Let $m = \lceil \ell/2 \rceil = 2^{k-1}$. Then $S[m]$ is the unique maximum element of S.
   - If $\ell > 1$, then the prefix $S[1\,..\,m-1]$ is a perfect ruler sequence.
   - If $\ell > 1$, then the suffix $S[m+1\,..\,\ell]$ is a perfect ruler sequence.

   For example, the following sequence is a perfect ruler sequence:

   $$\langle 2, 7, 6, 9, 5, 8, 5, 12, 1, 9, 4, 10, 7, 8, 3 \rangle$$

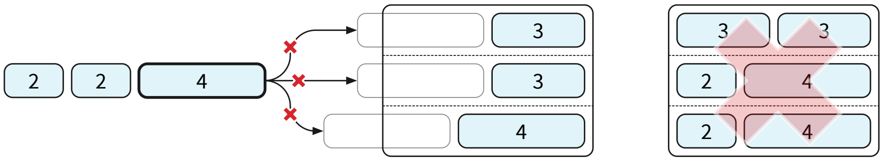
   Describe and analyze an efficient algorithm to compute the longest perfect ruler subsequence of a given array $A[1\,..\,n]$ of integers.

2. Suppose you are running a ferry across Lake Michigan.[1] The vehicle hold in your ferry is L meters long and three lanes wide. As each vehicle drives up to your ferry, you direct it to one of the three lanes; the vehicle then parks as far forward in that lane as possible. Vehicles must enter the ferry in the order they arrived; if the vehicle at the front of the queue doesn't fit into any of the lanes, then no more vehicles are allowed to board.

   Because your uncle runs the concession stand at the ferry terminal, you want to load as *few* vehicles onto your ferry as possible for each trip. But you don't want to be *obvious* about it, if the vehicle at the front of the queue fits anywhere, you must assign it to a lane where it fits. You can see the lengths of all vehicles in the queue on your security camera.

   Describe and analyze an algorithm to load the ferry. The input to your algorithm is the integer L and an array $len[1\,..\,n]$ containing the (integer) lengths of all vehicles in the queue. (You can assume that $1 \le len[i] \le L$ for all i.) Your output should be the *smallest* integer k such that you can put vehicles 1 through k onto the ferry, in such a way that vehicle $k+1$ does not fit. Express the running time of your algorithm as a function of both n (the number of vehicles) and L (the length of the ferry).

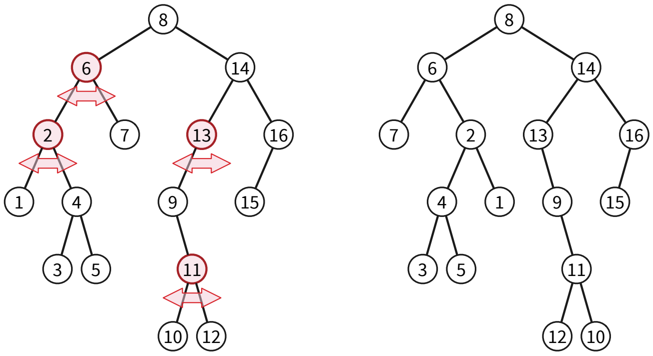
   For example, suppose $L = 6$, and the first six vehicles in the queue have lengths 3, 3, 4, 4, 2, and 2. Your algorithm should return the integer 3, because if you assign the first three vehicles to three different lanes, the fourth vehicle won't fit. (A different lane assignment gets all six vehicles on board, but that would rob your uncle of three customers.)



[1] Welcome aboard the Recursion Ferry! How we get across the water is none of your business!

3. CS 125 students Chef Gallon and Fade Waygone wrote some inorder traversal code for their MP on binary search trees. To keep things simple, they wisely chose the integers 1 through n as their search keys. Unfortunately, their code contained a subtle bug (which was nearly impossible to track down, thanks to version inconsistencies between Fade's laptop, the submission/grading server, and Oracle's ridiculous licencing terms) that would sporadically swap left and right child pointers in some binary tree nodes. As a result, their traversal code rarely returned the search keys in sorted order.

   For example, given the binary search tree below, if the four marked nodes had their left and right pointers swapped, Chef and Fade's traversal code would return the garbled "inorder" sequence $7, 6, 3, 4, 5, 2, 1, 8, 13, 9, 12, 11, 10, 14, 15, 16$.



   Chef and Fade submitted the output of several garbled traversals, but before they could submit the actual traversal code, Fade's laptop was infested with bees. After receiving a grade of 0 on their MP, Chef and Fade argued with their instructor that they should get *some* partial credit, because the sequences their code produced were at least consistent with correct binary search trees, and anyway the bees weren't their fault.

   Design and analyze an efficient algorithm to verify or refute Chef and Fade's claim (about the binary search trees, not the bees). The input to your algorithm is an array $A[1 .. n]$ containing a permutation of the integers 1 through n. Your algorithm should output TRUE if this array is the inorder traversal of an actual binary search tree with keys 1 through n, possibly with some left and right child pointers swapped, and FALSE otherwise. For example, if the input array contains $[5, 2, 3, 4, 1]$, your algorithm should return TRUE, and if the input array contains $[2, 5, 3, 1, 4]$, your algorithm should return FALSE.

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:

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

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 $i \leftarrow n$ down to 1
      $LBS[i,i] \leftarrow 0$
      for $k \leftarrow i+1$ to $n$
          if $A[i] \sim A[k]$
              $LBS[i,k] \leftarrow LBS[i+1,k-1] + 2$
          else
              $LBS[i,k] \leftarrow 0$
          for $j \leftarrow i$ to $k-1$
              $LBS[i,k] \leftarrow \max\{LBS[i,k],\ LBS[i,j] + LBS[j+1,k]\}$
  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.

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

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

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

(These recurrences do not require separate base cases, because $\sum \varnothing = 0$.) We can memoize these functions by adding two additional fields $v.\mathit{yes}$ and $v.\mathit{no}$ 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):
    $v.yes \gets v.fun$
    $v.no \gets 0$
    for all children $w$ of $v$
        ComputeMaxFun($w$)
        $v.yes \gets v.yes + w.no$
        $v.no \gets v.no + \max\{w.yes, w.no\}$
```

(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 ***$O(n)$ 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:

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

(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 $\boldsymbol{O(n)}$ ***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 ✧ Fall 2019

Homework 7

Due Tuesday, October 29, 2019 at 8pm

1. You new job at Object Oriented Parcel Service is to help direct delivery drivers through the city of Gridville. You are given a complete street map, in the form of a graph G, whose vertices are intersections, and whose edges represent streets between those intersections. Every street in Gridville runs in a straight line either north-south or east-west, and there are no one-way streets. One specific vertex s of G represents the OOPS warehouse.

   To increase fuel economy, decrease delivery times, and reduce accidents, OOPS imposes the following strict policies on its drivers.[1]

   - U-turns are forbidden, except at dead ends, where they are obviously required.
   - Left turns are forbidden, except where the road turns left, with no option to continue either straight or right.
   - Drivers must stop at every intersection.
   - Drivers must must park as close as possible to their destination address.

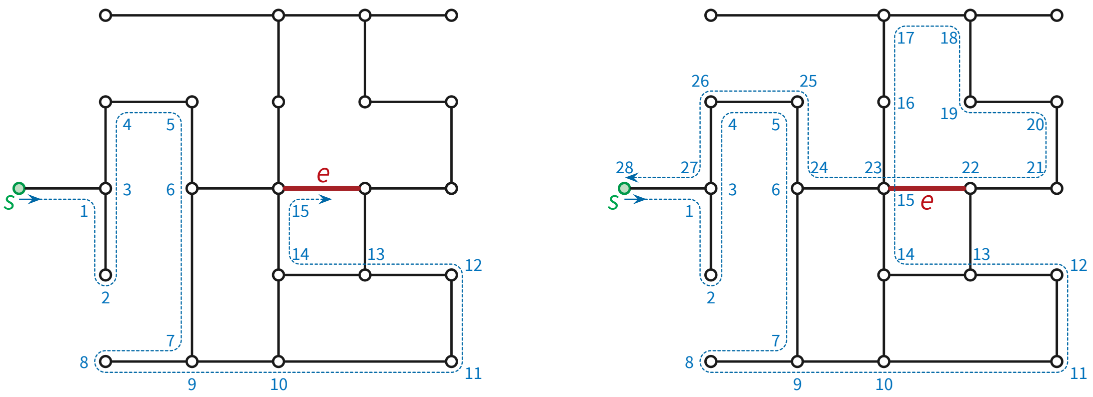
   Your job is to find routes from the OOPS warehouse to other locations in Gridville, with the smallest possible number of stops, that satisfy OOPS's driving policies. A destination is specified by an *edge* of G.

   (a) Describe and analyze an algorithm to find a legal route with the minimum number of stops from the OOPS warehouse to an arbitrary destination address. The input to your algorithm is the graph G, the start vertex s, and the destination edge; the output is the number of stops on the best legal route (or ∞ if there is no legal route).

   (b) After submitting your fancy new algorithm to your boss, you gently remind her that trucks have to return to the warehouse after making each delivery. Describe and analyze an algorithm to find a legal route with the minimum number of stops, from the OOPS warehouse, to an arbitrary destination address, and then back to the warehouse. The input to your algorithm is the graph G, the start vertex s, and the destination edge; the output is the number of stops on the best legal route (or ∞ if there is no legal route).

   For example, given the map on the next page, your algorithm for part (a) should return 15, and your algorithm for part (b) should return 28. Both optimal routes start with a forced right turn, followed by a forced U-turn, because turning left at a T intersection is forbidden. Notice that the optimal route to the destination is *not* a prefix of the optimal route to the destination and back.

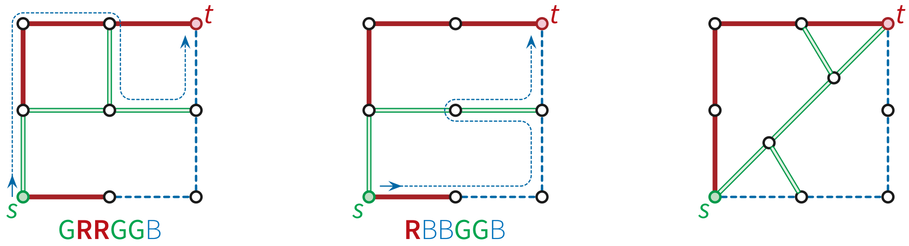
[1] OOPS maintains GPS trackers on every truck. If a driver ever breaks any of these rules, the tracker immediately shuts down and locks the truck, trapping the driver until a manager arrives, unlocks the truck, fires the driver, and installs a new replacement driver. OOPS managers are notoriously lazy, so most drivers keep enough food and water in their trucks to last several days.



Optimal delivery routes for OOPS drivers.

2. Suppose you are given an undirected graph G in which every edge is either red, green, or blue, along with two vertices s and t. Call a walk from s to t *awesome* if the walk does not contain three consecutive edges with the same color.

   Describe and analyze an algorithm to find the length of the shortest awesome walk from s to t. For example, given either the left or middle input below, your algorithm should return the integer 6, and given the input on the right, your algorithm should return ∞.



3. You are helping a group of ethnographers analyze some oral history data. The ethnographers have collected information about the lifespans of n different people, all now deceased, arbitrarily labeled with the integers 1 through n. Specifically, for some pairs (i, j), the ethnographers have learned one of the following facts:

   (a) Person i died before person j was born.

   (b) Person i and person j were both alive at some moment.

   The ethnographers are not sure that their facts are correct; after all, this information was passed down by word of mouth over generations, and memory is notoriously unreliable. So they would like you to determine whether the data they have collected is at least internally consistent, meaning there could have been people whose births and deaths consistent with their data.

   Describe and analyze an algorithm to answer the ethnographers' problem. Your algorithm should either output possible dates of birth and death that are consistent with all the stated facts, or it should report correctly that no such dates exist.

Solved Problem

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

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

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

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

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

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

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

Solution: Let A, B, C denote the capacities of the three jars. We reduce the problem to breadth-first search in 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
  - $\left\{\begin{matrix}(0,a+b,c) & \text{if } a+b \le B\\ (a+b-B,B,c) & \text{if } a+b \ge B\end{matrix}\right\}$ — pouring from jar 1 into jar 2
  - $\left\{\begin{matrix}(0,b,a+c) & \text{if } a+c \le C\\ (a+c-C,b,C) & \text{if } a+c \ge C\end{matrix}\right\}$ — pouring from jar 1 into jar 3

- $\begin{Bmatrix} (a+b,0,c) & \text{if } a+b \le A \\ (A,a+b-A,c) & \text{if } a+b \ge A \end{Bmatrix}$ — pouring from jar 2 into jar 1
- $\begin{Bmatrix} (a,0,b+c) & \text{if } b+c \le C \\ (a,b+c-C,C) & \text{if } b+c \ge C \end{Bmatrix}$ — pouring from jar 2 into jar 3
- $\begin{Bmatrix} (a+c,b,0) & \text{if } a+c \le A \\ (A,b,a+c-A) & \text{if } a+c \ge A \end{Bmatrix}$ — pouring from jar 3 into Jar 1
- $\begin{Bmatrix} (a,b+c,0) & \text{if } b+c \le B \\ (a,B,b+c-B) & \text{if } b+c \ge B \end{Bmatrix}$ — 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")
  - − ½ for using a slower or more specific algorithm than necessary
- + 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 ✧ Fall 2019

Homework 8

Due Tuesday, November 5, 2019 at 8pm

This is the last homework before Midterm 2.

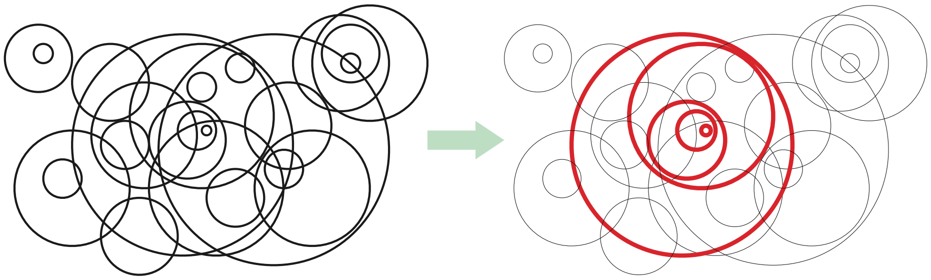
---

1. Over the summer, you pick up a part-time consulting gig at a gun range, designing targets for their customers to shoot at. A *target* consists of a properly nested set of circles, meaning for any two circles in the set, the smaller circle lies entirely inside the larger circle. The score for each shot is equal to the number of circles that contain the bullet hole.

   The shooters at the range aren't very good; their shots tend to be distributed uniformly at random on the target sheet. As a result, the expected score for any shot is proportional to the sum of the areas of the circles that make up the target. You'd like to make this expected score as large as possible.

   Now suppose your boss hands you a target sheet with n circles drawn on it. Describe and analyze an efficient algorithm to find a properly nested subset of these circles that maximizes the sum of the circle areas. You cannot *move* the circles; you must keep them exactly where your boss has drawn them.

   The input to your algorithm consists of three arrays $R[1..n]$, $X[1..n]$, and $Y[1..n]$, specifying the radius of each circle and the x- and y-coordinates of its center. The output is the sum of the circle areas in the best target.



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

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

Unknown to the Neighborhood Association, the children of Cheery Hells have published a secret web site, accessible only through a link embedded in yet another TikTok cover of “Spooky Scary Skeletons”, listing the amount of candy that each house in Cheery Hells will give to each visitor. (The web site also asks visitors to say “Gimme some Skittles, but I don’t wanna pay for them” instead of “Trick or treat”, just to mess with the grownups.)

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

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

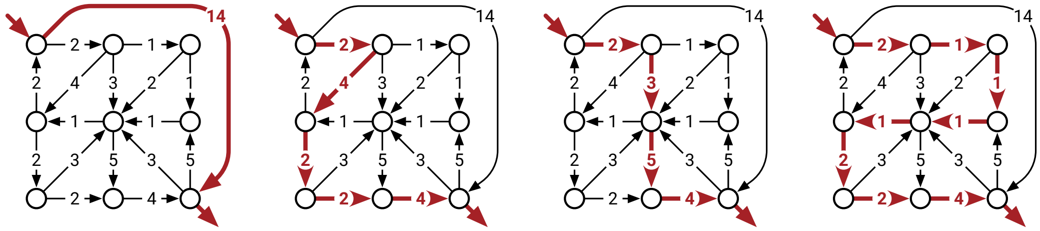
3. Morty needs to retrieve a stabilized plumbus from the Clackspire Labyrinth. He must enter the labyrinth using Rick’s interdimensional portal gun, traverse the Labyrinth to a plumbus, then take that plumbus through the Labyrinth to a fleeb to be stabilized, and finally take the stabilized plumbus back to the original portal to return home. Plumbuses are stabilized by fleeb juice, which any fleeb will release immediately after being removed from its fleebhole. An unstabilized plumbus will explode if it is carried more than 137 flinks from its original storage unit. The Clackspire Labyrinth smells like farts, so Morty wants to spend as little time there as possible.

   Rick has given Morty a detailed map of the Clackspire Labyrinth, which consist of a directed graph $G = (V, E)$ with non-negative edge weights (indicating distance in flinks), along with two disjoint subsets $P \subset V$ and $F \subset V$, indicating the plumbus storage units and fleebholes, respectively. Morty needs to identify a start vertex s, a plumbus storage unit $p \in P$, and a fleebhole $f \in F$, such that the shortest-path distance from p to f is at most 137 flinks long, and the length of the shortest walk $s \rightsquigarrow p \rightsquigarrow f \rightsquigarrow s$ is as short as possible.

   Describe and analyze an algo(burp)rithm to so(burp)olve Morty’s problem. You can assume that it is in fact possible for Morty to succeed.

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

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

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

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

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

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

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

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

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

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

CS/ECE 374 A ✧ Fall 2019

Homework 9

Due Tuesday, November 19, 2019 at 8pm

---

1. This problem asks you to develop polynomial-time algorithms for two (apparently) minor variants of 3Sat.

   (a) The input to **2Sat** is a boolean formula Φ in conjunctive normal form, with exactly **two** literals per clause, and the 2Sat problem asks whether there is an assignment to the variables of Φ such that every clause contains at least one True literal.

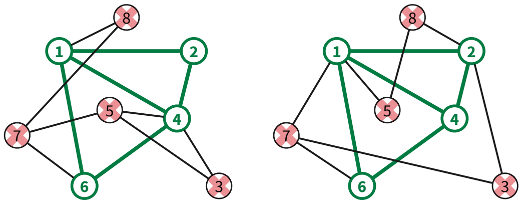
   Describe a polynomial-time algorithm for 2Sat. *[Hint: This problem is strongly connected to topics covered earlier in the semester.]*

   (b) The input to **Majority3Sat** is a boolean formula Φ in conjunctive normal form, with exactly three literals per clause. Majority3Sat asks whether there is an assignment to the variables of Φ such that every clause contains *at least two* True literals.

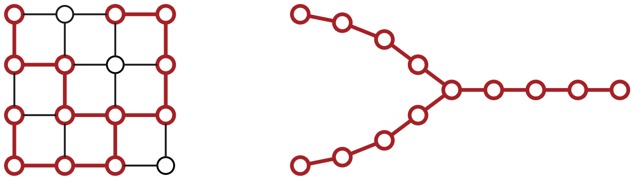
   Describe and analyze a polynomial-time reduction from Majority3Sat to 2Sat. Don't forget to prove that your reduction is correct.

   (c) Combining parts (a) and (b) gives us an algorithm for Majority3Sat. What is the running time of this algorithm?

2. Suppose we are given two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ with the same set of vertices $V = \{1, 2, \ldots, n\}$. Prove that is it NP-hard to find the smallest subset $S \subseteq V$ of vertices whose deletion leaves identical subgraphs $G_1 \setminus S = G_2 \setminus S$. For example, given the graphs below, the smallest subset has size 4.



3. 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.

Solved Problem

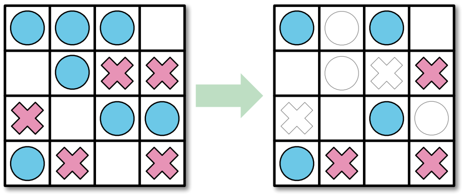
4. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions:

   (1) Every row contains at least one stone.

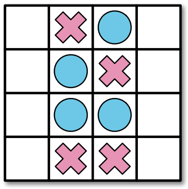
   (2) No column contains stones of both colors.

   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 $x_j =$ TRUE, remove all red stones from column j.
- If $x_j =$ 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.

$\Longleftarrow$ 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 ✧ Fall 2019

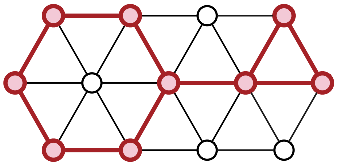
Homework 10

Due Tuesday, December 3, 2019 at 8pm

This is the last graded homework before the final exam.

This brings the total number of graded homework problems to 33,
at most 24 of which will count toward your final course grade.

1. A subset S of vertices in an undirected graph G is called ***square-free*** if, for every four distinct vertices $u, v, w, x \in S$, at least one of the four edges uv, vw, wx, xu is *absent* from G. That is, the subgraph of G induced by S has no cycles of length 4. Prove that finding the size of the largest square-free subset of vertices in a given undirected graph is NP-hard.



A square-free subset of 9 vertices, and all edges between them.
This is ***not*** the largest square-free subset in this graph.

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 $L(M) \neq \Sigma^*$?

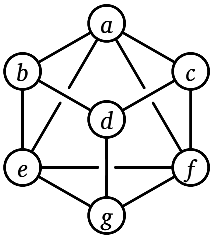
   *[Hint: Encode all the **bad** choices for some problem into a regular expression R, so that if **all** choices are bad, then $L(R) = \Sigma^*$.]*

3. ***This problem has been removed.*** — *We are deferring all discussion of undecidability until after Thanksgiving break. This problem will reappear on "Homework 11".*

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

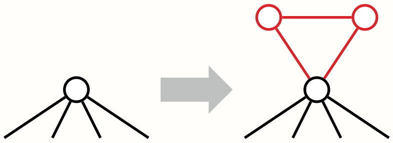
Solved Problem

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.

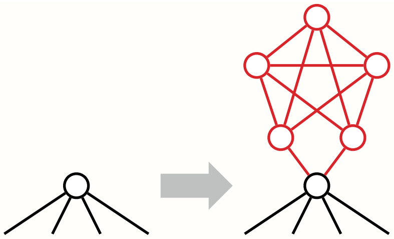
$\Longrightarrow$ Suppose G has a Hamiltonian cycle $v_1 \to v_2 \to \cdots \to v_n \to v_1$. We can construct a double-Hamiltonian tour of H by replacing each vertex v_i with the following walk:

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

$\Longleftarrow$ 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.

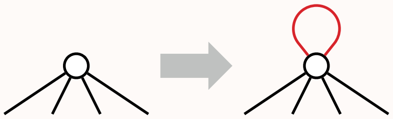


A vertex in G, and the corresponding modified vertex gadget in H.

■

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

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

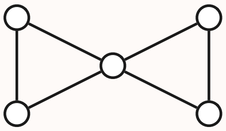


An incorrect vertex gadget.

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:

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

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



This graph has a double-Hamiltonian tour.

♣

CS/ECE 374 A ✧ Fall 2019

“Homework” 11

“Due” Monday, December 9, 2019

This homework is *not* for submission. However, undecidability questions are in scope for the final exam, so we still strongly recommend treating at least those questions as regular homework. Solutions will be released next Monday.

---

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

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

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

2. Let M be a Turing machine, let w be 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) Prove that the following language is undecidable:

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

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

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

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

   [Hint: This question is only for people who really want to get down in the Turing-machine weeds. Nothing like this will appear on the final exam.]

3. 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.