CS/ECE 374 A ✧ Fall 2021

Homework 1

Due Tuesday, August 31, 2021 at 8pm Central Time

- **Submit your written solutions electronically to Gradescope as PDF files.** Submit a separate PDF file for each numbered problem. If you plan to typeset your solutions, 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).
- Groups of up to three people can submit joint solutions on Gradescope. ***Exactly*** one student in each group should upload the solution and indicate their other group members. All group members must be already registered on Gradescope.
- 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.**
- **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.
- Written homework will be due every Tuesday at 8pm, except in weeks with exams. In addition, guided problems sets on PrairieLearn are due every **Monday** at 8pm; each student must do these individually. In particular, Guided Problem Set 1 is due Monday, August 30! Each Guided Problem Set has the same weight as one numbered homework problem.

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. Consider the following pair of mutually recursive functions on strings:

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

For example, the following derivation shows that $evens(\texttt{PARITY}) = \texttt{AIY}$:

$$\begin{aligned} evens(\texttt{PARITY}) &= odds(\texttt{ARITY}) \\ &= \texttt{A} \cdot evens(\texttt{RITY}) \\ &= \texttt{A} \cdot odds(\texttt{ITY}) \\ &= \texttt{A} \cdot (\texttt{I} \cdot evens(\texttt{TY})) \\ &= \texttt{A} \cdot (\texttt{I} \cdot odds(\texttt{Y})) \\ &= \texttt{A} \cdot (\texttt{I} \cdot (\texttt{Y} \cdot evens(\varepsilon))) \\ &= \texttt{A} \cdot (\texttt{I} \cdot (\texttt{Y} \cdot \varepsilon)) \\ &= \texttt{AIY} \end{aligned}$$

A similar derivation implies that $odds(\texttt{PARITY}) = \texttt{PRT}$.

   (a) Give a self-contained recursive definition for the function *evens* that does not involve the function *odds*.

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

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

You may assume without proof any result proved in class, in lab, or in the lecture notes. Otherwise, your proofs must be formal and self-contained, and they must invoke the *formal* definitions of concatenation $\bullet$, length $|\cdot|$, and the *evens* and *odds* functions. Do not appeal to intuition!

2. Consider the following recursive function that perfectly shuffles two strings together:

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

For example, the following derivation shows that $shuffle(\texttt{PRT}, \texttt{AIY}) = \texttt{PARITY}$:

$$\begin{aligned} shuffle(\texttt{PRT}, \texttt{AIY}) &= \texttt{P} \cdot shuffle(\texttt{AIY}, \texttt{RT}) \\ &= \texttt{P} \cdot (\texttt{A} \cdot shuffle(\texttt{RT}, \texttt{IY})) \\ &= \texttt{P} \cdot (\texttt{A} \cdot (\texttt{R} \cdot shuffle(\texttt{IY}, \texttt{T}))) \\ &= \texttt{P} \cdot (\texttt{A} \cdot (\texttt{R} \cdot (\texttt{I} \cdot shuffle(\texttt{T}, \texttt{Y})))) \\ &= \texttt{P} \cdot (\texttt{A} \cdot (\texttt{R} \cdot (\texttt{I} \cdot (\texttt{T} \cdot shuffle(\texttt{Y}, \varepsilon))))) \\ &= \texttt{P} \cdot (\texttt{A} \cdot (\texttt{R} \cdot (\texttt{I} \cdot (\texttt{T} \cdot (\texttt{Y} \cdot shuffle(\varepsilon, \varepsilon)))))) \\ &= \texttt{P} \cdot (\texttt{A} \cdot (\texttt{R} \cdot (\texttt{I} \cdot (\texttt{T} \cdot (\texttt{Y} \cdot \varepsilon))))) \\ &= \texttt{PARITY} \end{aligned}$$

(a) Prove that $shuffle(odds(w), evens(w)) = w$ for every string w.

(b) Prove $evens(shuffle(w,z)) = z$ for all strings w and z such that $|w| = |z|$.

You may assume without proof any result proved in class, in lab, or in the lecture notes. Otherwise, your proofs must be formal and self-contained, and they must invoke the *formal* definitions of concatenation $\bullet$ and the functions *shuffle*, *evens*, and *odds*. Do not appeal to intuition!

Rubrics

We will announce standard grading rubrics for common question types, which we will apply on all homeworks and exams. However, please remember that some homework and exam questions may fall outside the scope of these standard rubrics.

Standard induction rubric. For problems worth 10 points:

- \+ 1 for explicitly considering an *arbitrary* object.
- \+ 2 for a valid ***strong*** induction hypothesis
  - – **Deadly Sin!** No credit here 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.

Solved Problems

Each homework assignment will include at least one fully 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. For any string $w \in \{0,1\}^*$, let $swap(w)$ denote the string obtained from w by swapping the first and second symbols, the third and fourth symbols, and so on. For any string $w \in \{0,1\}^*$, let $swap(w)$ denote the string obtained from w by swapping the first and second symbols, the third and fourth symbols, and so on. For example:

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

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

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

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

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

- If $w = \varepsilon$, then

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

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

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

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

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

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

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

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

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

- If $w = \varepsilon$, then

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

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

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

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

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

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

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

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

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

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

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

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

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

■

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

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

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

Solution: Let w be an arbitrary palindrome.

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

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

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

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

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

Rubric: 4 points: standard induction rubric (scaled)

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

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

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

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

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

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

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

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

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

  Because $a = b$, we have $w = ay^R a$ and $w^R = aya$.
  The recursive definition of string equality implies $y^R a = ya$.
  Right cancellation implies 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).

CS/ECE 374 A ✧ Fall 2021

Homework 2

Due Tuesday, September 7, 2021 at 8pm

- **Submit your written solutions electronically to Gradescope as PDF files.** Submit a separate PDF file for each numbered problem. If you plan to typeset your solutions, 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 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.

---

1. Let L be the set of all strings w in $\{\mathtt{A},\mathtt{B}\}^*$ for which $\#(\mathtt{ABBA}, w) \geq 2$. Here $\#(x, w)$ denotes the number of occurences of the substring x in the string w.

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

   (b) Describe a DFA over the alphabet $\Sigma = \{\mathtt{A},\mathtt{B}\}$ 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 be heavily penalized, even if they are perfectly correct.

   [Hint: The shortest string in L has length 7.]

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

   - The number of times the substring `01` appears in w is *not* divisible by 3[1].
   - The length of w is even.
   - The binary value of w equals 2 (mod 3).

   For example: The string `0101` satisfies all three conditions, so `0101` 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 36 states, which is *far* 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 be heavily penalized, even if they are perfectly 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 could be as problematic as imprecision and handwaving.

[1] Recall that a is divisible by b if and only if $a \equiv 0 \pmod{b}$.

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.
  - 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.

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.
    - Use doubled circles to indicate accepting states A.
    - If $A = \emptyset$, you must say so explicitly.
    - If your drawing omits a junk/trash/reject state, you must 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. But you must still give an explicit description of the states Q, the start state s, and the accepting states A.
  - **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.
  - In particular, each state must have a mnemonic name.
  - For product constructions, explaining the states in the factor DFAs is both necessary and sufficient.
  - Yes, we mean it: A perfectly correct drawing of a perfectly correct DFA with no state explanation is worth at most 6 points.
- 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.

Solved problem

3. ***C comments*** are the set of strings over alphabet $\Sigma = \{*, /, A, \diamond, \downarrow\}$ that form a proper comment in the C program language and its descendants, like C++ and Java. Here $\downarrow$ represents the newline character, $\diamond$ represents any other whitespace character (like the space and tab characters), and A represents any non-whitespace character other than * or /.[2] There are two types of C comments:

   - Line comments: Strings of the form //$\cdots\downarrow$
   - 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 $\downarrow$ 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:

   /***/ //◇//◇$\downarrow$ /*///◇*◇$\downarrow$**/ /*◇//◇$\downarrow$◇*/

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

   /*/ //◇//◇$\downarrow$◇$\downarrow$ /*◇/*◇*/◇*/

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

[2]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, pretend that the characters ', ", and \ do not exist.

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

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

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

Solution:

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

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

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

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

Solution:

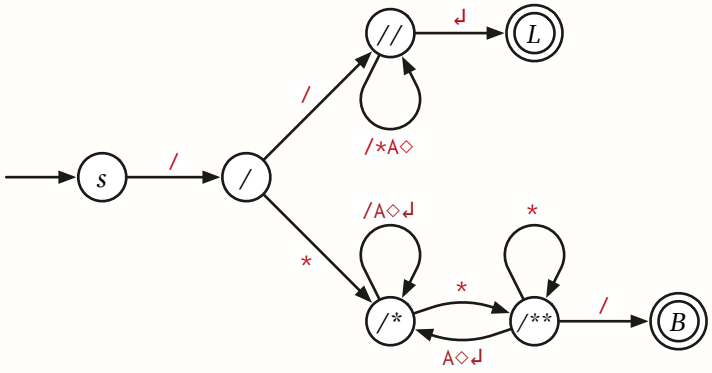
$$\left(\diamond + \text{↲} + //(/ + * + \mathtt{A} + \diamond)^*\text{↲} + /*(/ + \mathtt{A} + \diamond + \text{↲} + **^*(\mathtt{A} + \diamond + \text{↲}))^* *^*/\right)^*$$

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

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

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

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



The states are labeled mnemonically as follows:

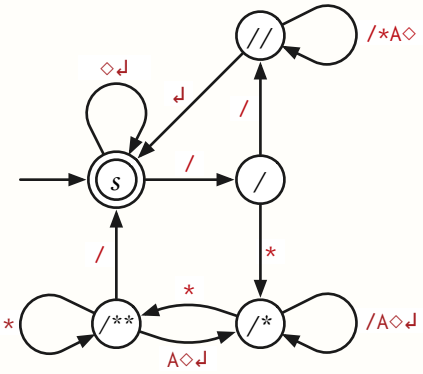
- *s* — We have not read anything.
- */* — We just read the initial /.
- *//* — We are reading a line comment.
- *L* — We have just read a complete line comment.
- */** — We are reading a block comment, and we did not just read a * after the opening /*.
- */*** — We are reading a block comment, and we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

■

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

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

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



The states are labeled mnemonically as follows:

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

■

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

CS/ECE 374 A ✧ Fall 2021

☙ Homework 3 ❧

Due Tuesday, September 14, 2021 at 8pm

---

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

   (a) $\{0^m 1^n \mid m \text{ and } n \text{ are relatively prime}\}$

   (b) $\{w \in (0+1)^* \mid 10^n 1^n \text{ for } n > 0 \text{ is a suffix of } w\}$

   (c) The set of all palindromes in $(0+1)^*$ whose length is divisible by 3.

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

   (a) $\{0^a 1 w 1 0^c \mid w \in \Sigma^*, (a \le |w| + c) \text{ and } (|w| \le a + c \text{ or } c \le a + |w|)\}$

   (b) $\left\{0^a w 0^a \;\middle|\; w \in \Sigma^+, a > 0, |w| \ge 0\right\}$

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

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

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

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

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

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

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

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

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

   Solution: Not regular. Call this language L_b.

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

   Let x and y be arbitrary distinct strings in F.

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

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

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

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

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

   Because F is infinite, L_b cannot be regular. ■

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

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

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

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

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

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

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

Solution: Regular. Call this language L_d.

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

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

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

Solution: Regular. Call this language L_d.

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

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

■

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

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

Standard fooling set rubric. For problems worth 5 points:

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

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

CS/ECE 374 A ✧ Fall 2021

Homework 4

Due Tuesday, September 21, 2021 at 8pm

This is the last homework before Midterm 1.

---

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

(see next page for Question 2)

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

   (a) $\mathsf{thirds}(L) := \{\mathsf{thirds}(w) | w \in L\}$,
   where $\mathsf{thirds}(w)$ is the subsequence of w containing every third symbol.
   For example, $\mathsf{thirds}(011000110) = 100$.
   (notice, we picked the third, sixth, and ninth symbols in 011000110)

   (b) $\mathsf{thirds}^{-1}(L) := \{w \in \Sigma^* | \mathsf{thirds}(w) \in L\}$.

Standard langage transformation rubric. For problems worth 10 points:

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

Solved problem

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

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

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

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

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

■

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

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

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

Solution: Consider the regular language $L = \mathtt{0}^*\mathtt{1}$.

Expanding the regular expression lets us rewrite $L = \{\mathtt{0}^n\mathtt{1} \mid n \geq 0\}$. It follows that $\mathit{double}(L) = \{\mathtt{0}^n\mathtt{1}\mathtt{0}^n\mathtt{1} \mid n \geq 0\}$. I claim that this language is not regular.

Let x and y be arbitrary distinct strings in L.

Then $x = \mathtt{0}^i\mathtt{1}$ and $y = \mathtt{0}^j\mathtt{1}$ for some integers $i \neq j$.

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

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

We conclude that L is a fooling set for $\mathit{double}(L)$.

Because L is infinite, $\mathit{double}(L)$ cannot be regular. ■

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

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

Let F be the infinite language $\mathtt{0}\mathtt{1}^*\mathtt{0}$.

Let x and y be arbitrary distinct strings in F.

Then $x = \mathtt{0}\mathtt{1}^i\mathtt{0}$ and $y = \mathtt{0}\mathtt{1}^j\mathtt{0}$ for some integers $i \neq j$.

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

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

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

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

Rubric: 5 points:

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

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

Standard langage transformation rubric. For problems worth 10 points:

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

CS/ECE 374 A ✧ Fall 2021

Homework 5

Due Tuesday, October 5, 2021 at 8pm Central Time

---

1. Consider the following cruel and unusual sorting algorithm, proposed by Gary Miller:

```
CRUEL(A[1..n]):
  if n > 1
      CRUEL(A[1..n/2])
      CRUEL(A[n/2+1..n])
      UNUSUAL(A[1..n])
```

```
UNUSUAL(A[1..n]):
  if n = 2
      if A[1] > A[2]                          ⟨⟨the only comparison!⟩⟩
          swap A[1] ↔ A[2]
  else
      for i ← 1 to n/4                        ⟨⟨swap 2nd and 3rd quarters⟩⟩
          swap A[i + n/4] ↔ A[i + n/2]
      UNUSUAL(A[1..n/2])                      ⟨⟨recurse on left half⟩⟩
      UNUSUAL(A[n/2+1..n])                    ⟨⟨recurse on right half⟩⟩
      UNUSUAL(A[n/4+1..3n/4])                 ⟨⟨recurse on middle half⟩⟩
```

The comparisons performed by Miller's algorithm do not depend at all on the values in the input array; such a sorting algorithm is called ***oblivious***. Assume for this problem that the input size n is always a power of 2.

   (a) Prove by induction that CRUEL correctly sorts any input array. *[Hint: Follow the smallest $n/4$ elements. Follow the largest $n/4$ elements. Follow the middle $n/2$ elements. What does UNUSUAL actually **do**??]*

   (b) Prove that CRUEL would *not* correctly sort if we removed the for-loop from UNUSUAL.

   (c) Prove that CRUEL would *not* correctly sort if we swapped the last two lines of UNUSUAL.

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

   (e) What is the running time of CRUEL? Justify your answer.

2. Dakshita is putting together a list of famous cryptographers, each with their dates of birth and death: al-Kindi (801–873), Giovanni Fontana (1395–1455), Leon Alberti (1404–1472), Charles Babbage (1791–1871), Alan Turing (1912–1954), Joan Clarke (1917–1996), Ann Caracristi (1921–2016), and so on. She wonders which two cryptographers on her list had the longest overlap between their lifetimes. For example, among the seven example cryptographers, Clarke and Caracristi had the longest overlap of 45 years (1921–1966).

Dakshita formalizes her problem as follows. The input is an array $A[1..n]$ of records, each with two numerical fields $A[i].birth$ and $A[i].death$ and a string field $A[i].name$. The desired output is the maximum, over all indices $i \neq j$, of the overlap length

$$\min\{A[i].death,\ A[j].death\} - \max\{A[i].birth,\ A[j].birth\}.$$

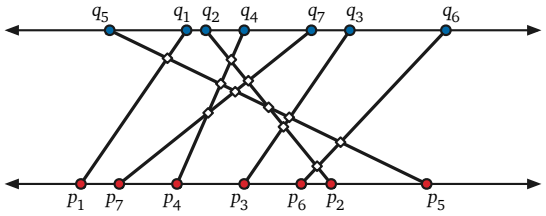
Describe and analyze an efficient algorithm to solve Dakshita's problem.

[Hint: Start by splitting the list in half by birth date. Do not *assume that cryptographers always die in the same order they are born. Assume that birth and death dates are distinct and accurate to the nanosecond.]*

Rubrics

Solved Problems

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



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

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

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

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

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

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

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

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

Rubric: This is enough for full credit.

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

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

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

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

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

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

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

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

CS/ECE 374 A ✧ Fall 2021

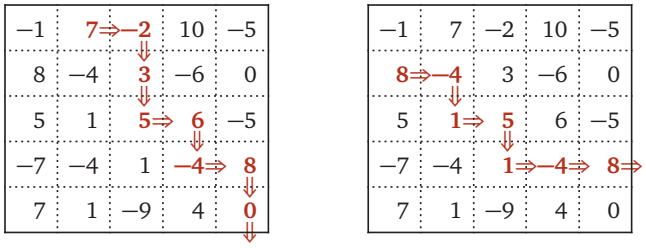
Homework 6

Due Tuesday, October 12, 2021 at 8pm Central Time

---

1. *Vankin's Mile* is an American solitaire game played on an $n \times n$ square grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his score. The object of the game is to score as many points as possible.

   For example, given the grid shown below, the player can score $7-2+3+5+6-4+8+0 = 23$ points by following the path on the left, or they can score $8-4+1+5+1-4+8 = 15$ points by following the path on the right.



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

   (b) A variant called *Vankin's Niknav* adds an additional constraint to Vankin's Mile: *The sequence of values that the token touches must be a* ***palindrome****.* Thus, the example path on the right is valid, but the example path on the left is not. Describe and analyze an efficient algorithm to compute the maximum possible score for an instance of Vankin's Niknav, given the $n \times n$ array of values as input.

2. A ***snowball*** is a poem or sentence that starts with a one-letter word, where each later word is one letter longer than its predecessor. For example:

   I am the fire demon, moving castles: Calcifer!

   Snowballs, sometimes also known as ***chaterisms*** or ***rhopalisms***, are one of many styles of constrained writing practiced by OuLiPo, a loose gathering of writers and mathematicians, founded in France in 1960 but still active today.

   Describe and analyze an algorithm to extract the longest snowball hidden in a given string of text. You are given an array $T[1..n]$ of English letters as input. Your goal is to find the longest possible sequence of disjoint substrings of T, where the ith substring is an English word of length i. Your algorithm should return the number of words in this sequence.

   Your algorithm will call the library function IsWord, which takes a string w as input and returns True if and only if w is an English word. IsWord(w) runs in $O(|w|)$ time.

   For example, given the input string

   ```
   EVENIFYOUAMTHEAREMYFIRELEASTDEMONFAVORITEMOVINGCASTLESVEGETABLECALCIFER
   ```

   your algorithm should return the integer 8:

   ```
   EVENIFYOUAMTHEAREMYFIRELEASTDEMONFAVORITEMOVINGCASTLESVEGETABLECALCIFER
   ```

Standard dynamic programming rubric. For problems worth 10 points:

- 3 points for a clear and correct English description of the recursive function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.)
  - No credit if the description is inconsistent with the recurrence.
  - No credit if the description does not explicitly describe how the function value depends on the named input parameters.
  - No credit if the description refers to internal states of the eventual dynamic programming algorithm, like "the current index" or "the best score so far". The function must have a well-defined value that depends *only* on its input parameters (and constant global variables).
  - An English explanation of the *recurrence* or *algorithm* does not qualify. We want a description of *what* your function returns, not (here) an explanation of *how* that value is computed.

  — 1 for naming the function "OPT" or "DP" or any single letter.
- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.

  + 1 for base case(s). —½ for one *minor* bug, like a typo or an off-by-one error.

  + 3 for recursive case(s). —1 for each *minor* bug, like a typo or an off-by-one error.

  — 2 for greedy optimizations without proof, even if they are correct.

  – No credit for the rest of the problem if the recursive case(s) are incorrect.
- 3 points for iterative details

  + 1 for describing an appropriate memoization data structure

  + 1 for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested for loops, be sure to specify the nesting order.

  + 1 for correct 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 and English description of the underlying recursive function (or equivalently, the contents of the memoization structure). ***Perfectly correct iterative pseudocode, with no explanation or time analysis, is worth at most 6 points out of 10.***
- Official solutions will provide target time bounds. Faster algorithms are worth more points, and slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n in either direction. Partial credit is scaled to the new maximum score, and all points above 10 (for algorithms that are faster than our target time bound) are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we do include them, significantly more students submit incorrect algorithms with the target running time (earning 0/10) instead of correct algorithms that are slower than the target (earning 7/10).
- Partial credit for incomplete solutions depends on the running time of the ***best possible*** completion (up to the target running time). For example, consider a solution that contains *only* a clear English description of a function, with no recurrence or iterative details. If the described function *can* be developed into an algorithm with the target running time, the solution is worth 3 points; however, if the function leads to an algorithm that is slower than the target time by a factor of n, the solution could be worth only 2 points (= 70% of 3, rounded).

Solved Problem

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

   BANANAANANAS BANANAANANAS BANANAANANAS

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

   PRODGYRNAMAMMIINCG DYPRONGARMAMMICING

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

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

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

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

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

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

Rubric: 5 points, standard dynamic programming rubric. 3 points for a slower polynomial-time algorithm; scale partial credit accordingly.

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

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

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

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

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

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

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

Rubric: 5 points, standard dynamic programming rubric. 3 points for a slower polynomial-time algorithm; scale partial credit accordingly.

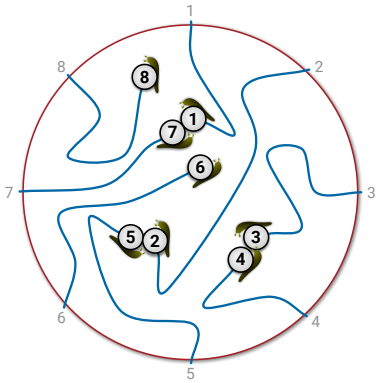
CS/ECE 374 A ✧ Fall 2021

Homework 7

Due Tuesday, October 19, 2021 at 8pm Central Time

---

1. Every year, as part of its annual meeting, the Antarctican Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to n. During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.



The end of a typical Antarctican SLUG race. Snails 6 and 8 never find mates.
The organizers must pay $M[3,4] + M[2,5] + M[1,7]$.

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

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array M as input.

2. Suppose you are given a NFA $M = (\{0,1\}, Q, s, A, \delta)$ *without* ε-transitions and a binary string $w \in \{0,1\}^*$. Describe and analyze an efficient algorithm to determine whether M accepts w. Concretely, the input NFA M is represented as follows:

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

   Your input consists of the integer k, the array $Acc[1..k]$, the array $inDelta[1..k, 0..1, 1..k]$, and the input string $w[1..n]$. Your algorithm should return $\textsc{True}$ if M accepts w, and $\textsc{False}$ if M does not accept w. Report the running time of your algorithm as a function of k (the number of states in M) and n (the length of w). *[Hint: Do not convert M to a DFA!!]*

Solved Problems

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

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

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

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

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

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

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

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

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

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

■

Rubric: 10 points, standard dynamic programming rubric

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The function $MaxFun$ obeys the following recurrence:

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

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

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

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

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

The algorithm spends $O(1)$ time at each node (because each node has exactly one parent and one grandparent) and therefore runs in $\boldsymbol{O(n)}$ ***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 2021

Homework 8

Due Tuesday, October 26, 2021 at 8pm Central Time

---

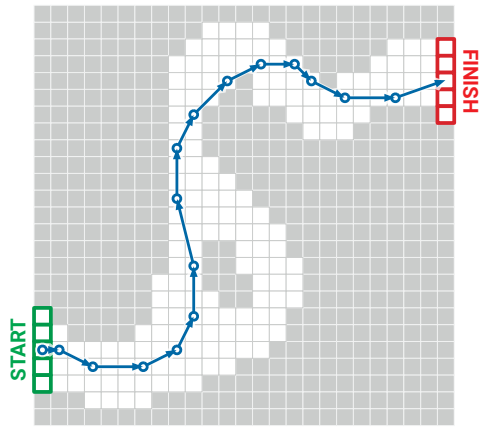
1. ***Racetrack*** (also known by several other names, including *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.[1] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid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

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

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

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

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



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

---

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

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

2. Jeff likes to go on a long bike ride every Sunday, but because he is lazy, he absolutely refuses to ever ride into the wind.

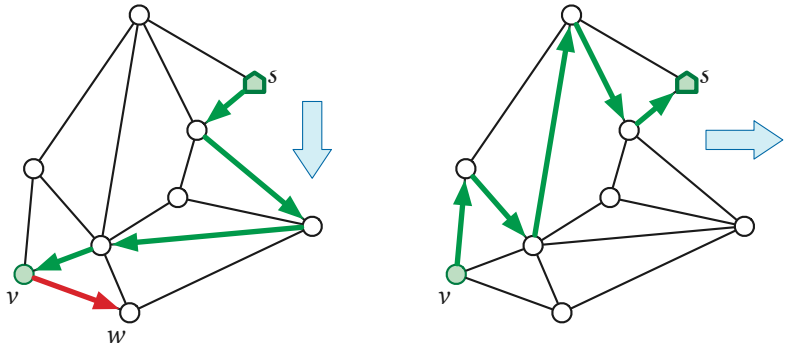
   Jeff has encoded a map of all bike-safe roads in Champaign-Urbana into an undirected graph $G = (V, E)$ whose vertices represent intersections and sharp corners, and whose edges represent straight road segments. Jeff's home is represented in G by a special vertex s. Every edge of G is labeled with its length and orientation.

   (a) One Sunday the weather forecast predicts wind from due north all day long, which means Jeff can only ride along each road segment in the direction that tends south. Describe and analyze an algorithm to determine the longest total distance Jeff can ride, without ever riding into the wind, before he has to call his wife to come pick him up in the car.

   (b) The following Sunday's weather forecast predicts wind from due north in the morning, followed by wind from due west in the afternoon. Describe and analyze an algorithm to find the longest total distance Jeff can ride if he starts at home, rides out to some destination in the morning, eats lunch at noon (while the wind shifts), and then rides home in the afternoon, all without ever riding into the wind.

   In both cases, your input consists of the graph G and the start vertex s. Despite overpowering evidence to the contrary, assume that Jeff can ride infinitely fast, and that no roads in Champaign-Urbana are oriented exactly north-south or exactly east-west.

   For example, suppose Jeff has the graph G shown below. On the first Sunday, Jeff can ride from s to w along the path shown on the left, including the red edge from v to w. On the second Sunday, Jeff can ride from s to v along the green path on the left in the morning, and then from v back to s along the green path on the right in the afternoon; however, he cannot ride to w, because every path from w to s requires riding into the wind at least once, and Jeff's wife is tired of driving out to the middle of nowhere to rescue him.



3. This problem is intended as a practice run for future homeworks, the second midterm, the final exam. **Each student must submit individually.**

   On the course Gradescope site, you will find an assignment called "Homework 8.3". Do not open this Gradescope assignment until you have the following items:

   - Two blank white sheets of paper. (In particular, *not* lined notebook paper.)
   - A pen with dark ink, preferably blue or black. (In particular, *not* a pencil.)
   - A fully-charged and working cell phone with a scanning app installed. (Gradescope recommends Scannable for iOS devices and Genius Scan for Android devices.)
   - A well-lit environment for scanning.

   The assignment will ask you to write/draw something, scan the paper, convert your scan to a PDF file, and upload the PDF to Gradescope. (Gradescope will automatically assign pages of your uploaded PDF to corresponding subproblems.)

   Alternatively, you can write/draw on a tablet and a note-taking app, export your note as a PDF file, upload the PDF to Gradescope.

   Once you open the Gradescope assignment, you will have 15 minutes to complete the submission process.

   The precise content to be written/drawn will be revealed in the Gradescope assignment. (Don't worry, we won't ask for anything technical. The actual writing/drawing should take less than 60 seconds.) If you are not used to your scanning app, we strongly recommend practicing the entire scanning process before starting the assignment.

   Rubric: 10 points = 1 for using blank white paper + 2 for using a dark pen + 2 for submitting a scan instead of a raw photo + 3 for a *good* scan (in focus, high contrast, properly aligned, no keystone effect, no shadows, no background) + 2 for following content instructions. Yes, this actually counts.

Rubrics

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** (For the solved problem below: "shortest path in G from $(0, 0, 0)$ to any target vertex")
  - — "Breadth-first search" is not a problem; it's an algorithm!
- \+ 1 for correctly applying the correct **algorithm**. (For the solved problem below, "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 to the construction algorithm. For example, for an algorithm that uses dynamic programming to build the graph quickly, apply the standard dynamic programming rubric.

Solved Problem

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

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

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

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

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

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

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

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

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

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

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

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

Rubric: 10 points: standard graph reduction rubric

- Brute force construction is fine.

— 1 for calling Dijkstra instead of BFS

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

CS/ECE 374 A 💀 Fall 2021

🦇🦇 Homework 9 🦇🦇

Due Tuesday, November 2, 2021 at 8pm Central Time

This is the last homework before Midterm 2.

1. 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. As usual, do not assume that edge weights are integers.

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

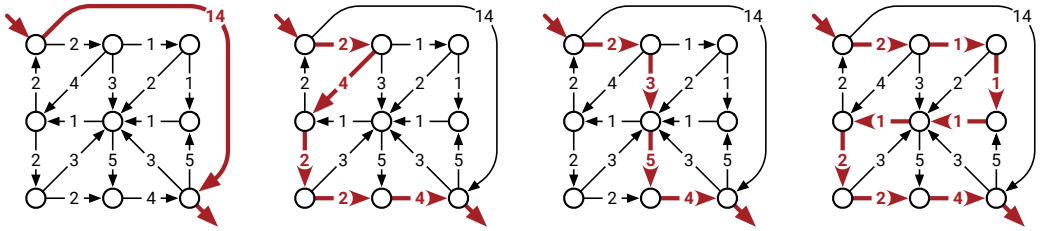
   You have converted the trail map into a directed graph $G = (V, E)$, whose vertices represent trail intersections, and whose edges represent trail segments. A subset $R \subseteq V$ of the vertices indicate the locations of the *R*epellent *R*efilling stations, and a subset $H \subseteq E$ of the edges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

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

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

Solved Problem

3. 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)

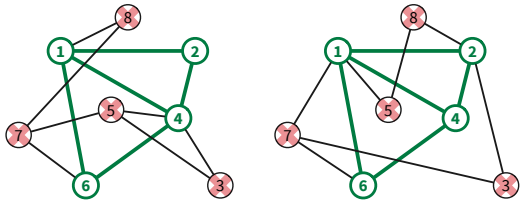
CS/ECE 374 A ✧ Fall 2021

Homework 10

Due Tuesday, November 16, 2021 at 8pm

---

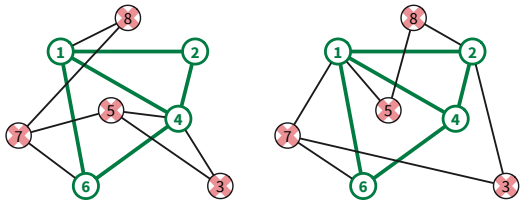
1. Suppose we are given two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ with the same set of vertices $V = \{1, 2, \ldots, n\}$. You are given the following problem: find the smallest subset $S \subseteq V$ of vertices whose deletion leaves identical subgraphs $G_1 \setminus S = G_2 \setminus S$. For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



Provide a polynomial-time reduction for this problem from *any one of the following three* problems:

- MaxIndependentSet: MaxIndependentSet(G, m) returns 1 if the size of the largest independent set in graph G is m, otherwise returns 0.
- MaxClique: MaxClique(G, m) returns 1 if the size of the largest clique in G is m, otherwise returns 0.
- MinVertexCover: MinVertexCover(G, m) returns 1 if the size of the smallest vertex cover in G is m, otherwise returns 0.

Hint: There exists a reduction to all three problems; you may pick whichever one is most convenient for you.



2. 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?

Solved Problem

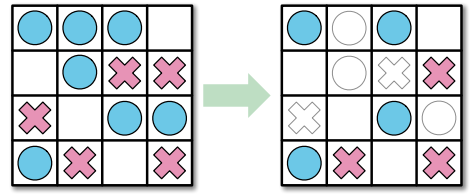
3. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions:

   (1) Every row contains at least one stone.

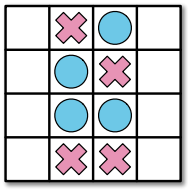
   (2) No column contains stones of both colors.

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

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



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

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

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

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

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

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

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

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

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

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

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

This reduction clearly requires only polynomial time. ■

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

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

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

CS/ECE 374 A ✧ Fall 2021

Homework 11

Due Tuesday, November 30, 2021 at 8pm

---

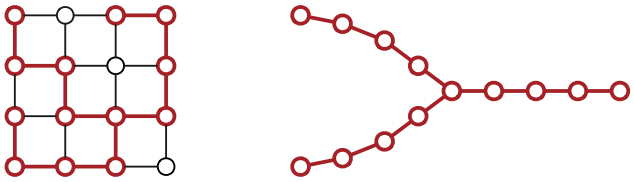
1. (a) A **quasi-satisfying assignment** for a 3CNF boolean formula Φ is an assignment of truth values to the variables such that at most one clause in Φ does not contain a true literal.

   Prove that it is NP-hard to determine whether a given 3CNF boolean formula has a quasi-satisfying assignment.

   (b) A **near-clique** in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ where adding a single edge between two vertices in S results in the set S becoming a clique.

   Prove that it is NP-hard to find the size of the largest near-clique in a graph $G = (V, E)$.

2. 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*.

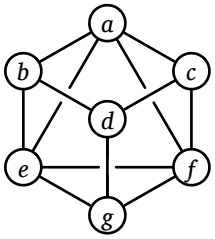


This grid graph contains a wye whose paths have length 4.

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.

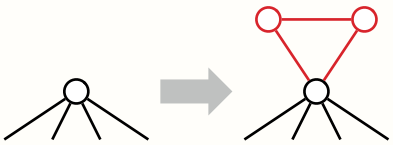
Solved Problem

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



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

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



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

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

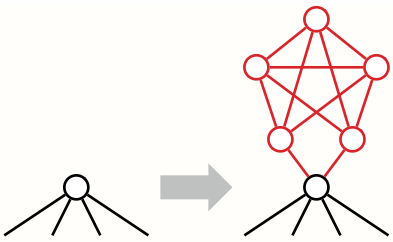
$\Longrightarrow$ Suppose G has a Hamiltonian cycle $v_1 \to v_2 \to \cdots \to v_n \to v_1$. We can construct a double-Hamiltonian tour of H by replacing each vertex v_i with the following walk:

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

$\Longleftarrow$ 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.

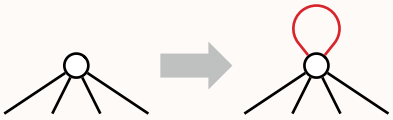


A vertex in G, and the corresponding modified vertex gadget in H.

■

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.

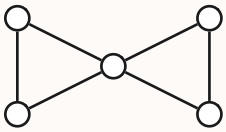


An incorrect vertex gadget.

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.

♣

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?

CS/ECE 374 A ✧ Fall 2021

☙ "Homework" 12 ❧

"Due" Monday, December 6, 2021

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

---

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

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

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

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

   Prove that the following language is undecidable:

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

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

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

3. Prove that the following language is undecidable:

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

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