CS/ECE 374 A ✧ Fall 2023

Homework 1

Due Tuesday, August 29, 2023 at 9pm 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, you are welcome to 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.
- **You may use any source at your disposal**—paper, electronic,[1] or human—but you ***must*** cite ***every*** source that you use,[2] you ***must*** write everything yourself in your own words, and you are responsible for any errors in the sources you use.[3] See the academic integrity policies on the course web site for more details.
- Written homework is normally due every Tuesday at 9pm. In addition, guided problem sets on PrairieLearn are normally due every **Monday** at 9pm; each student must do these individually. In particular, Guided Problem Set 1 is due Monday, August 28!
- Both guided problem sets and homework may be submitted up to 24 hours late for 50% partial credit, or for full credit with an approved extension. See the grading policies on the course web site for more details.
- Each homework will include at least one fully solved problem, similar to that week's assigned problems, together with the rubric we would use to grade this problem if it appeared in an actual homework or exam. These model solutions show our recommendations for structure, presentation, and level of detail in your homework solutions. (Obviously, the actual *content* of your solutions won't match the model solutions, because your problems are different!) Homeworks may also include additional practice problems.
- Standard grading rubrics for many problem types can be found on the course web page. For example, the problems in this week's homework will be graded using the standard induction rubric. (Weak induction makes the baby Jesus cry.)

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] Yes, including ChatGPT.
[2] Yes, including ChatGPT.
[3] Yes, including ChatGPT.

1. Consider the following recursively defined function:

$$stutter(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ aa \bullet stutter(x) & \text{if } w = ax \end{cases}$$

For example, $stutter(\texttt{MISSISSIPPI}) = \texttt{MMIISSSSIISSSSIIPPPPII}$.

(a) Prove that $|stutter(w)| = 2|w|$ for every string w.

(b) Prove that $stutter(x \bullet y) = stutter(x) \bullet stutter(y)$ for all strings x and y.

(c) ***Practice only. Do not submit solutions.***

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

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

For example, $\texttt{MISSIPPIPPI}^R = \texttt{IPPIPPISSIM}$.

Prove that $stutter(w)^R = stutter(w^R)$ for every string w.

You may freely use any result proved in lecture, in lab, or in the lecture notes. Otherwise your proofs must be formal and self-contained. In particular, your proofs *must* invoke the formal recursive definitions of string length and concatenation (and for part (c), reversal).

2. For each positive integer n, we define two strings p_n and v_n, respectively called the nth *Piṅgala string* and the nth *Virahāṇka string*. Piṅgala strings are defined by the following recurrence:

$$p_n = \begin{cases} \texttt{1} & \text{if } n = 1 \\ \texttt{0} & \text{if } n = 2 \\ p_{n-2} \bullet p_{n-1} & \text{otherwise} \end{cases}$$

For example:

$$p_7 = \overbrace{\texttt{1001}}^{p_5}\overbrace{\texttt{0}}^{p_4}\overbrace{\texttt{10}}^{p_5}\overbrace{\texttt{0}\texttt{1}\texttt{0}\texttt{1}\texttt{0}}^{p_6}.$$

Virahāṇka strings are defined more indirectly as

$$v_n = \begin{cases} \texttt{1} & \text{if } n = 1 \\ \text{grow}(v_{n-1}) & \text{otherwise} \end{cases}$$

where the string function grow is defined as follows:

$$\text{grow}(w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \texttt{0} \cdot \text{grow}(x) & \text{if } w = \texttt{1}x \\ \texttt{10} \bullet \text{grow}(x) & \text{if } w = \texttt{0}x \end{cases}$$

For example:

$$\text{grow}(\texttt{01010010}) = \texttt{10} \bullet \texttt{0} \bullet \texttt{10} \bullet \texttt{0} \bullet \texttt{10} \bullet \texttt{10} \bullet \texttt{0} \bullet \texttt{10} = \texttt{1001001010010}$$

Finally, recall that the Fibonacci numbers are defined recursively as follows:

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

(a) Prove that $|p_n| = F_n$ for all $n \geq 1$.

(b) Prove that $\text{grow}(w \bullet z) = \text{grow}(w) \bullet \text{grow}(z)$ for all strings w and z.

(c) Prove that $p_n = v_n$ for all $n \geq 1$. *[Hint: Careful!]*

(d) ***Practice only. Do not submit solutions.***
Prove that $|v_n| = F_n$ for all $n \geq 1$.

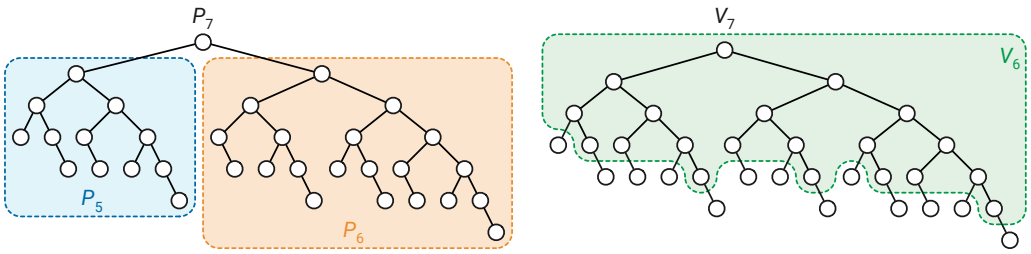
As in problem 1, you may freely use any result that proved in lecture, in lab, or in the lecture notes. Otherwise your proofs must be formal and self-contained. In particular, your proofs *must* invoke the formal recursive definitions of the strings p_n and v_n, the grow function, and the Fibonacci numbers F_n.

*3. ***Practice only. Do not submit solutions.***

For each non-negative integer n, we recursively define two binary trees P_n and V_n, called the nth *Piṅgala tree* and the nth *Virahāṇka tree*, respectively.

- P_0 and V_0 are empty trees, with no nodes.
- P_1 and V_1 each consist of a single node.
- For any integer $n \geq 2$, the tree P_n consists of a root with two subtrees; the left subtree is a copy of P_{n-1}, and the right subtree is a copy of P_{n-2}.
- For any integer $n \geq 2$, the tree L_n is obtained from L_{n-1} by attaching a new right child to every leaf and attaching a new left child to every node that has only a right child.

The following figure shows the recursive construction of these two trees when $n = 7$.



Recall that the Fibonacci numbers are defined recursively as follows:

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

(a) Prove that the tree P_n has exactly F_n leaves.

(b) Prove that the tree V_n has exactly F_n leaves.
[Hint: You need to prove a stronger result.]

(c) Prove that the trees P_n and V_n are identical, for all $n \geq 0$.
[Hint: The hardest part of this proof is developing the right language/notation.]

As in problem 1, you may freely use any result that proved in lecture, in lab, or in the lecture notes. Otherwise your proofs must be formal and self-contained. In particular, your proofs *must* invoke the formal recursive definitions of the trees P_n and V_n and the Fibonacci numbers F_n.

Solved Problems

3. 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 $w = 0$ or $w = 1$, 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 $a, b \in \{0, 1\}$ and $x \in \{0, 1\}^*$, 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.

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

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

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

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

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

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

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 $y^R = y$.
  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).

5. Let $L \subseteq \{0,1\}^*$ be the language defined recursively as follows:

   - The empty string ε is in L.
   - For any string $x \in L$, the strings $0101x$ and $1010x$ are also in L.
   - For all strings x and y such that $xy \in L$, the strings $x00y$ and $x11y$ are also in L. (In other words, inserting two consecutive 0s or two consecutive 1s anywhere in a string in L yields another string in L.)
   - These are the only strings in L.

   Let EE denote the set of all strings $w \in \{0,1\}^*$ such that $\#(0,w)$ and $\#(1,w)$ are both even.

   In the following proofs, you may freely use any result proved in lecture, in lab, in the lecture notes, or earlier in your homework. Otherwise your proofs must be formal and self-contained; in particular, they must invoke the formal recursive definitions of $\#$ and L.

   (a) Prove that $L \subseteq EE$.

   Solution: Let w be an arbitrary string in L. We need to prove that $\#(0,w)$ and $\#(1,w)$ are both even. Here I will prove only that $\#(0,w)$ is even; the proof that $\#(1,w)$ is even is symmetric.

   Assume for every string $x \in L$ such that $|x| < |w|$ that $\#(0,x)$ is even.

   There are several cases to consider, mirroring the definition of L.

   - Suppose $w = \varepsilon$. Then $\#(0,w) = 0$, and 0 is even.
   - Suppose $w = 0101x$ or $w = 1010x$ for some string $x \in L$. The definition of $\#$ (applied four times) implies $\#(0,w) = \#(0,x)+2$. The inductive hypothesis implies $\#(0,x)$ is even. We conclude that $\#(0,w)$ is even.
   - Suppose $w = x00y$ for some strings x and y such that $xy \in L$. Then

   $$\begin{aligned}\#(0,w) &= \#(0,x00y)\\ &= \#(0,x)+\#(0,00)+\#(0,y)\\ &= \#(0,x)+\#(0,y)+\#(0,00)\\ &= \#(0,xy)+2\end{aligned}$$

   The induction hypothesis implies $\#(0,xy)$ is even. We conclude that $\#(0,w) = \#(0,xy)+2$ is also even.
   - Finally, suppose $w = x11y$ for some strings x and y such that $xy \in L$. Then

   $$\begin{aligned}\#(0,w) &= \#(0,x11y)\\ &= \#(0,x)+\#(0,11)+\#(0,y)\\ &= \#(0,x)+\#(0,y)\\ &= \#(0,xy)\end{aligned}$$

   The induction hypothesis implies $\#(0,w) = \#(0,xy)$ is even.

In all cases, we have shown that $\#(0, w)$ is even. Symmetric arguments imply that $\#(1, w)$ is even. We conclude that $w \in EE$. ■

Rubric: 5 points: standard induction rubric (scaled). Yes, this is enough detail for $\#(1, w)$. If explicit proofs are given for both $\#(0, w)$ and $\#(1, w)$, grade them independently, each for 2½ points.

(b) Prove that $EE \subseteq L$.

Solution: Let w be an arbitrary string in EE. We need to prove that $w \in L$.

Assume that for every string $x \in EE$ such that $|x| < |w|$, we have $x \in L$.

There are four (overlapping) cases to consider, depending on the first four symbols in w.

- Suppose $|w| < 4$. Then w must be one of the strings ε, 00, or 11; brute force inspection implies that every other string of length at most 3 (0, 1, 01, 10, 000, 001, 010, 011, 100, 101, 110, 111) has an odd number of 0s or an odd number of 1s (or both). All three strings ε, 00, and 11 are in L. In all other cases, we can assume that $|w| \geq 4$, so the "first four symbols of w" are well-defined.
- Suppose the first four symbols of w are 0000 or 0001 or 0010 or 0011 or 0100 or 1000 or 1001 or 1100. Then $w = x00y$ for some (possibly empty) strings x and y. Arguments in part (a) imply that $\#(0, xy) = \#(0, w) - 2$ and $\#(1, xy) = \#(1, w)$ are both even. Thus $xy \in EE$ by definition of EE. So the induction hypothesis implies $xy \in L$. We conclude that $w = x00y \in L$ by definition of L.
- Suppose the first four symbols of w are 0011 or 0110 or 0111 or 1011 or 1100 or 1101 or 1110 or 1111.) After swapping 0s and 1s, the argument in the previous case implies that $w \in L$.
- Finally, suppose the first four symbols of w are 0101 or 1010; in other words, suppose $w = 0101x$ or $w = 1010x$ for some (possibly empty) string x. Then $\#(0, x) = \#(0, w) - 2$ and $\#(1, x) = \#(1, w) - 2$ are both even, so $x \in EE$ by definition. The induction hypothesis implies $x \in L$. We conclude that $w \in L$ by definition of L.

Each of the 16 possible choices for the first four symbols of w is considered in at least one of the last three cases.

In all cases, we conclude that $w \in L$. ■

Rubric: 5 points: standard induction rubric (scaled). This is not the only correct proof. This is not the only correct way to express this particular case analysis.

CS/ECE 374 A ✧ Fall 2023

Homework 2

Due Wednesday, September 6, 2023 at 9pm Central Time

(One day later than usual because of Labor Day)

- Starting with this homework, please start your solution to each *lettered subproblem* ((a), (b), (c), etc.) on a new page. Yes, even if the previous subproblem is only one line long. Please also remember to tell Gradescope which page(s) are relevant for which subproblems.

---

1. For each of the following languages over the alphabet $\{0,1\}^*$, describe an equivalent regular expression, and briefly explain why your regular expression is correct. There are infinitely many correct answers for each language.

   (a) All strings in 1^*01^* whose length is a multiple of 3.

   (b) All strings that begin with the prefix 001, end with the suffix 100, and contain an odd number of 1s.

   (c) All strings that contain both 0011 and 1100 as substrings.

   (d) All strings that contain the substring 01 an odd number of times.

   (e) $\{0^a1^b0^c \mid a \geq 0 \text{ and } b \geq 0 \text{ and } c \geq 0 \text{ and } a \equiv b + c \pmod 2\}$.

2. For each of the following languages over the alphabet $\Sigma = \{0,1\}$, describe a DFA that accepts the language, and briefly describe the purpose of each state. You can describe your DFA using a drawing, or using formal mathematical notation, or using a product construction; see the standard DFA rubric.

   (a) All strings in 1^*01^* whose length is a multiple of 3.

   (b) All strings that represent a multiple of 5 in base 3. For example, this language contains the string 10100, because $10100_3 = 90_{10}$ is a multiple of 5. (Yes, base 3 allows the digits 0, 1, and 2, but your input string will never contain a 2.)

   (c) All strings containing the substring 01010010. (The required substring is $p_6 = v_6$ from Homework 1.)

   (d) All strings whose ninth-to-last symbol is 0, or equivalently, the set

   $$\left\{x0z \mid x \in \Sigma^* \text{ and } z \in \Sigma^8\right\}.$$

   (e) All strings w such that $(\#(0,w) \bmod 3) + (\#(1,w) \bmod 7) = (|w| \bmod 4)$.

   [Hint: Don't try to draw the last two.]

3. ***Practice only. Do not submit solutions.***

This question asks about strings over the set of *pairs* of bits, which we will write vertically. Let Σ_2 denote the set of all bit-pairs:

$$\Sigma_2 = \left\{\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\right\}$$

We can interpret any string w of bit-pairs as a $2 \times |w|$ matrix of bits; each row of this matrix is the binary representation of some non-negative integer, possibly with leading 0s. Let $hi(w)$ and $lo(w)$ respectively denote the *numerical values* of the top and bottom row of this matrix. For example, $hi(\varepsilon) = lo(\varepsilon) = 0$, and if

$$w = \begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}0011\\0101\end{bmatrix}$$

then $hi(w) = 3$ and $lo(w) = 5$.

(a) Describe a DFA that accepts the language $L_{+1} = \{w \in \Sigma_2^* \mid hi(w) = lo(w) + 1\}$.
For example, $w = \begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1100\\1011\end{bmatrix} \in L_{+1}$, because $hi(w) = 12$ and $lo(w) = 11$.

(b) Describe a regular expression for L_{+1}.

(c) Describe a DFA that accepts the language $L_{\times 3} = \{w \in \Sigma_2^* \mid hi(w) = 3 \cdot lo(w)\}$.
For example, $w = \begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}1001\\0011\end{bmatrix} \in L_3$, because $hi(w) = 9$ and $lo(w) = 3$.

(d) Describe a regular expression for $L_{\times 3}$.

*(e) Describe a DFA that accepts the language $L_{\times 3/2} = \{w \in \Sigma_2^* \mid 2 \cdot hi(w) = 3 \cdot lo(w)\}$.
For example, $w = \begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}1001\\0110\end{bmatrix} \in L_{\times 3/2}$, because $hi(w) = 9$ and $lo(w) = 6$.

(Don't bother with the regular expression for this one.)

Solved problem

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

   - Line comments: Strings of the form //⋯↲
   - Block comments: Strings of the form /*⋯*/

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

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

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

   /*/ //◇//◇↲◇↲ /*◇/*◇*/◇*/

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

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

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

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

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:

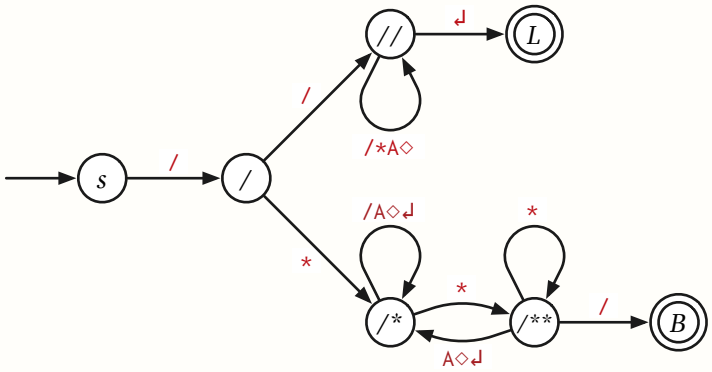
$$\left(\diamond + \text{↲} + //(/ + * + \mathtt{A} + \diamond)^*\text{↲} + /*(/ + \mathtt{A} + \diamond + \text{↲} + **^*(\mathtt{A} + \diamond + \text{↲}))^* **^*/\right)^*$$

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:

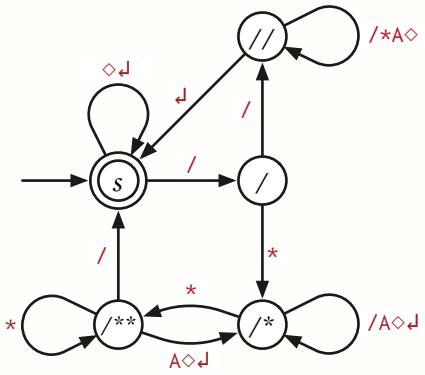
- s — We have not read anything.
- / — We just read the initial /.
- // — We are reading a line comment.
- L — We have just read a complete line comment.
- /* — We are reading a block comment, and we did not just read a * after the opening /*.
- /** — We are reading a block comment, and we just read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

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

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

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

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

Prove that the reversal of every regular language is regular.

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

Assume for every regular expression s smaller than r that there is a regular expression s' such that $L(s') = L(s)^R$.

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

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

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

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

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

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

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

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

Set $r' := s' + t'$; then we have

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

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

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

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

Rubric: Standard induction rubric!!

CS/ECE 374 A ✧ Fall 2023

Homework 3

Due Tuesday, September 12, 2023 at 9pm Central Time

- Please start your solution to each *lettered subproblem* ((a), (b), (c), etc.) on a new page. Please also remember to tell Gradescope which page(s) are relevant for which subproblems.

---

1. Prove that the following languages over the alphabet $\Sigma = \{0, 1\}$ are *not* regular.

   (a) $\left\{0^a 1 0^b 1 0^c \;\middle|\; 2b = a + c\right\}$.

   (b) The set of all palindromes in Σ^* whose lengths are divisible by 7.

   (c) $\left\{1^m 0^n \;\middle|\; m + n > 0 \text{ and } \gcd(m, n) = 1\right\}$

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

2. For each of the following 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.

   (a) Strings in which the substrings 01 and 10 appear the same number of times. For example, $1100011 \in L$ because both substrings appear once, but $01000011 \notin L$.

   (b) Strings in which the substrings 00 and 11 appear the same number of times. For example, $1100011 \in L$ because both substrings appear twice, but $01000011 \notin L$.

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

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

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

*3. ***Practice only. Do not submit solutions.***

A ***Moore machine*** is a variant of a finite-state automaton that produces output; Moore machines are sometimes called finite-state *transducers*. For purposes of this problem, a Moore machine formally consists of six components:

- A finite set Σ called the input alphabet
- A finite set Γ called the output alphabet
- A finite set Q whose elements are called states
- A start state $s \in Q$
- A transition function $\delta : Q \times \Sigma \to Q$
- An output function $\omega : Q \to \Gamma$

More intuitively, a Moore machine is a graph with a special start vertex, where every node (state) has one outgoing edge labeled with each symbol from the input alphabet, and each node (state) is additionally labeled with a symbol from the output alphabet.

The Moore machine reads an input string $w \in \Sigma^*$ one symbol at a time. For each symbol, the machine changes its state according to the transition function δ, and then outputs the symbol $\omega(q)$, where q is the new state. Formally, we recursively define a *transducer* function $\omega^* : Q \times \Sigma^* \to \Gamma^*$ as follows:

$$\omega^*(q, w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \omega(\delta(q,a)) \cdot \omega^*(\delta(q,a), x) & \text{if } w = ax \end{cases}$$

Given input string $w \in \Sigma^*$, the machine outputs the string $\omega^*(w, s) \in \Gamma^*$. The ***output language $L^\circ(M)$*** of a Moore machine M is the set of all strings that the machine can output:

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

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

(b) Let M be an arbitrary Moore machine whose input alphabet Σ and output alphabet Γ are identical. Prove that the language

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

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

[Hint: These problems are easier than they look!]

Solved problems

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.

   Consider the set $F = (012)^+$.

   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 $yz \in L_b$.

   Thus, z is a distinguishing suffix for x and y.

   We conclude that F is a fooling set for L_b.

   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.

Consider the set $F = (001101)^+$.

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 and 00, because $i < j$, so $yz \in L_b$.

Thus, z is a distinguishing suffix for x and y.

We conclude that F is a fooling set for L_c.

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

CS/ECE 374 A ✧ Fall 2023

☙ Homework 4 ❧

Due Tuesday, September 19, 2023 at 9pm Central Time

This is the last homework before Midterm 1.

---

1. Recall the following string functions from Homework 1:

$$stutter(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ aa \bullet stutter(x) & \text{if } w = ax \end{cases} \qquad \mathsf{grow}(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \mathtt{0} \cdot \mathsf{grow}(x) & \text{if } w = \mathtt{1}x \\ \mathtt{10} \bullet \mathsf{grow}(x) & \text{if } w = \mathtt{0}x \end{cases}$$

For example, $stutter(\mathtt{1001}) = \mathtt{11000011}$, and $\mathsf{grow}(\mathtt{1001}) = \mathtt{0} \bullet \mathtt{10} \bullet \mathtt{10} \bullet \mathtt{0} = \mathtt{010100}$.

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

(a) $\textsc{Stutter}(L) = \{stutter(w) \mid w \in L\}$

(b) $\textsc{Unstutter}(L) = \{w \mid stutter(w) \in L\}$

(c) $\textsc{Grow}(L) = \{grow(w) \mid w \in L\}$

(d) $\textsc{Ungrow}(L) = \{w \mid grow(w) \in L\}$

2. Give context-free grammars for the following languages, and clearly explain how they work and the set of strings generated by each nonterminal. Grammars with unclear or missing explanations may receive little or no credit. On the other hand, we do *not* want formal proofs of correctness.

(a) $\left\{\mathtt{0}^a\mathtt{1}\mathtt{0}^b\mathtt{1}\mathtt{0}^c \;\middle|\; b = 2a + 2c\right\}$.

(b) $\left\{\mathtt{0}^a\mathtt{1}\mathtt{0}^b\mathtt{1}\mathtt{0}^c \;\middle|\; 2b = a + c\right\}$.

(c) The set of all palindromes in Σ^* whose lengths are divisible by 7.

*(d) ***Practice only. Do not submit solutions.***

Strings in which the substrings $\mathtt{00}$ and $\mathtt{11}$ appear the same number of times. For example, $\mathtt{1100011} \in L$ because both substrings appear twice, but $\mathtt{01000011} \notin L$.

Yes, you've seen most of these languages before.

*3. ***Practice only. Do not submit solutions.***

Let L_1 and L_2 be arbitrary regular languages over the alphabet $\Sigma = \{0, 1\}$. Prove that the following languages are also regular.

(a) $\textsc{Faro}(L_1, L_2) := \{faro(x, z) \mid x \in L_1 \text{ and } z \in L_2 \text{ with } |x| = |z|\}$, where

$$faro(x, z) := \begin{cases} z & \text{if } x = \varepsilon \\ a \cdot faro(z, y) & \text{if } x = ay \end{cases}$$

For example, $faro(0011, 0101) = 00011011$ and $\textsc{Faro}(0^*, 1^*) = (01)^*$.

(b) $\textsc{Shuffles}(L_1, L_2) := \bigcup_{w \in L_1, y \in L_2} shuffles(w, y)$, where $shuffles(w, y)$ is the set of all strings obtained by shuffling w and y, or equivalently, all strings in which w and y are complementary subsequences. Formally:

$$shuffles(w, y) = \begin{cases} \{y\} & \text{if } w = \varepsilon \\ \{w\} & \text{if } y = \varepsilon \\ \{a\} \bullet shuffles(x, y) \cup \{b\} \bullet shuffles(w, z) & \text{if } w = ax \text{ and } y = bz \end{cases}$$

For example, $shuffles(001, 1) = \{0011, 0101, 1001\}$ and $shuffles(00, 11) = \{0011, 0101, 0110, 1001, 1010, 1100\}$. Finally, $\textsc{Shuffles}(0^*, 1^*) = (0 + 1)^*$.

Both of these names are taken from methods of mixing a deck of playing cards. A *shuffle* divides the deck into two smaller stacks, and then interleaves those two stacks arbitrarily. A *Faro shuffle* or *perfect shuffle* divides the pack of cards exactly in half, and then interleaves them perfectly; the final deck alternates between cards from one half and cards from the other half. Faro shuffles are the basis of several card tricks.

Solved problems

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

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

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

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

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

■

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

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

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

Solution: Consider the regular language $L = 0^*1$.

Expanding the regular expression lets us rewrite $L = \{0^n1 \mid n \geq 0\}$. It follows that $double(L) = \{0^n10^n1 \mid n \geq 0\}$. I claim that this language is not regular.

Let x and y be arbitrary distinct strings in L.

Then $x = 0^i1$ and $y = 0^j1$ for some integers $i \neq j$.

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

- $xx \in double(L)$ by definition, but
- $yx = 0^i10^j1 \notin double(L)$ because $i \neq j$.

We conclude that L is a fooling set for $double(L)$.

Because L is infinite, $double(L)$ cannot be regular. ■

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

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

Let F be the infinite language 01^*0.

Let x and y be arbitrary distinct strings in F.

Then $x = 01^i0$ and $y = 01^j0$ for some integers $i \neq j$.

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

- $xz = 01^i01^i = ww$ where $w = 01^i$, so $xz \in double(\Sigma^*)$, but
- $yx = 01^j01^i \notin double(\Sigma^*)$ because $i \neq j$.

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

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

Rubric: 5 points:

- 2 points for describing a regular language L such that $double(L)$ is not regular.
- 3 point for the fooling set proof (standard fooling set rubric, scaled and rounded)

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

5. Give context-free grammars for the following languages over the alphabet $\Sigma = \{0, 1\}$. Clearly explain how they work and the set of strings generated by each nonterminal. Grammars with unclear or missing explanations may receive little or no credit; on the other hand, we do *not* want formal proofs of correctness.

   (a) In any string, a ***run*** is a maximal non-empty substring of identical symbols. For example, the string $0111000011001 = 0^1 1^3 0^4 1^2 0^2 1^1$ consists of six runs.

   Let L_a be the set of all strings in Σ^* that contain two runs of 0s of equal length. For example, L_a contains the strings 01101111 and 01001011100010 (because each of those strings contains more than one run of 0s of length 1) but L_a does not contain the strings 000110011011 and 00000000111.

Solution:

$S \to ACB$	strings with two blocks of 0s of same length
$A \to \varepsilon \mid X1$	empty or ends with 1
$B \to \varepsilon \mid 1X$	empty or starts with 1
$C \to 0C0 \mid 0D0$	$0^n y 0^n$, where y starts and ends with 1
$D \to 1 \mid 1X1$	starts and ends with 1
$X \to \varepsilon \mid 1X \mid 0X$	all strings: $(0+1)^*$

Every string in L has the form $x0^n y0^n z$, where x is either empty or ends with 1, y starts and ends with 1, and z is either empty or begins with 1. Nonterminal A generates the prefix x; non-terminal B generates the suffix z; nonterminal C generates the matching runs of 0s, and nonterminal D generates the interior string y.

The same decomposition can be expressed more compactly as follows:

$S \to B \mid B1A \mid A1B \mid A1B1A$	strings with two blocks of 0s of same length
$A \to 1A \mid 0A \mid \varepsilon$	all strings: $(0+1)^*$
$B \to 0B0 \mid 010 \mid 01A10$	$0^n y 0^n$, where y starts and ends with 1

■

Rubric: 5 points = 3 for clearly correct grammar + 2 for clear explanation. These are not the only correct solutions.

(b) $L_b = \{w \in \Sigma^* \mid w \text{ is } \textit{not} \text{ a palindrome}\}$.

Solution:

$S \to 0S0 \mid 0S1 \mid 1S0 \mid 1S1 \mid A$	non-palindromes
$A \to 0B1 \mid 1B0$	start and end with different symbols
$B \to 0B \mid 1B \mid \varepsilon$	all strings

Every non-palindrome w can be decomposed as either $w = x0y1z$ or $w = x1y0z$, for some substrings x, y, z such that $|x| = |z|$. Non-terminal S generates the prefix x and matching-length suffix z; non-terminal A generates the distinct symbols, and non-terminal B generates the interior substring y. ■

Solution:

$S \to 0S0 \mid 1S1 \mid A$	non-palindromes
$A \to 0B1 \mid 1B0$	start and end with different symbols
$B \to 0B \mid 1B \mid \varepsilon$	all strings

Every non-palindrome w must have a prefix x and a substring y such that either $w = x0y1x^R$ or $w = x1y0x^R$. Specifically, x is the longest common prefix of w and w^R. In the first case, the grammar generates w as follows:

$$S \rightsquigarrow^* xAx^R \rightsquigarrow x0B1x^R \rightsquigarrow^* x0y1x^R = w$$

The derivation for $w = x1y0x^R$ is similar. ■

Rubric: 5 points = 3 for clearly correct grammar + 2 for clear explanation. These are not the only correct solutions.

CS/ECE 374 A ✧ Fall 2023

Homework 5

Due Tuesday, October 3, 2023 at 9pm Central Time

---

1. In the lab on Wednesday, you'll see an algorithm that finds a local minimum in a one-dimensional array in $O(\log n)$ time. This question asks you to consider two higher-dimensional versions of this problem.

   (a) Suppose we are given a two-dimensional array $A[1..n, 1..n]$ of distinct integers. An array element $A[i, j]$ is called a ***local minimum*** if it is smaller than its four immediate neighbors:

   $$A[i, j] < \min\{A[i-1, j],\ A[i+1, j],\ A[i, j-1],\ A[i, j+1]\}$$

   To avoid edge cases, we assume all cells in row 1, row n, column 1, and column n have value $+\infty$.

   Describe and analyze an algorithm to find a local minimum in A as quickly as possible. (Remember that faster algorithms are worth more points, but only if they are correct.)

   [Hint: Suppose $A[i, j]$ is the smallest element in row i. If $A[i, j]$ is smaller than both of its vertical neighbors $A[i-1, j]$ and $A[i+1, j]$, we are clearly done. But what if $A[i, j] > A[i+1, j]$?]

   [Hint: This problem is more subtle than it appears at first glance; many published solutions for this problem on the internet are incorrect. The main issue is that a local minimum in a rectangular subarray is not necessarily a local minimum in the original array. Design a recursive algorithm for the following more general problem: Given a two-dimensional array that contains a local minimum ***whose value is less than the value of every border cell****, find such a local minimum.]*

   (b) Now suppose we are given a three-dimensional array $A[1..n, 1..n, 1..n]$ of distinct integers. An array element $A[i, j, k]$ is called a ***local minimum*** if it is smaller than its six immediate neighbors:

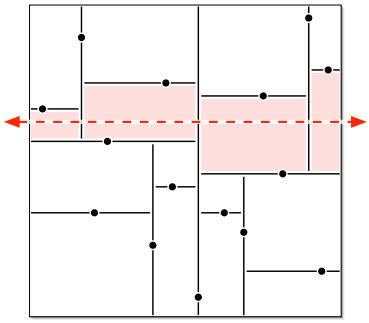
   $$A[i, j] < \min\left\{\begin{array}{l} A[i-1, j, k],\ A[i+1, j, k], \\ A[i, j-1, k],\ A[i, j+1, k], \\ A[i, j, k-1],\ A[i, j, k+1] \end{array}\right\}$$

   To avoid edge cases, we assume all cells on the boundary of the array have value $+\infty$.

   Describe and analyze an algorithm to find a local minimum in A as quickly as possible.

(Remember that faster algorithms are worth more points, but only if they are correct.)

2. Suppose we have n points scattered inside a two-dimensional box. A *kd-tree* recursively subdivides the points as follows. First we split the box into two smaller boxes with a *vertical* line, then we split each of those boxes with *horizontal* lines, and so on, always alternating between horizontal and vertical splits. Each time we split a box, the splitting line partitions the rest of the interior points *as evenly as possible* by passing through a median point in the interior of the box (*not* on its boundary). If a box doesn't contain any points, we don't split it any more; these final empty boxes are called *cells*.



A kd-tree for 15 points. The dashed line crosses the four shaded cells.

(a) How many cells does the kd-tree have, as a function of n? Prove that your answer is correct.

(b) In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n? Prove that your answer is correct. Assume that $n = 2^k - 1$ for some integer k. *[Hint: There is more than one function f such that $f(15) = 4$.]*

(c) Suppose we have n points stored in a kd-tree. Describe and analyze an algorithm that counts the number of points above a given horizontal line (such as the dashed line in the figure) as quickly as possible. *[Hint: Use part (b).]*

I should have specified that the following information is stored in each internal node v in the kd-tree:

- $v.x$ and $v.y$: The coordinates of the point defining the cut at v
- $v.dir \in \{vertical, horizontal\}$: The direction of the cut at v.
- $v.left$ and $v.right$: The children of v if $v.dir = vertical$
- $v.up$ and $v.down$: The children of v if $v.dir = horizontal$
- $v.size$: the number of points=cuts in the subtree rooted at v.

Instead I allowed arbitrary information to be computed in preprocessing; that freedom allows a much simpler and more efficient query algorithm!

(d) Describe and analyze an efficient algorithm that counts, given a kd-tree storing n points, the number of points that lie inside a given rectangle R with horizontal and vertical sides. *[Hint: Use part (c).]*

Assume that all x-coordinates and y-coordinates are distinct; that is, no two points lie on the same horizontal line or the same vertical line, no point lies on the query line in part (c), and no point lies on the boundary of the query rectangle in part (d).

*3. ***Practice only. Do not submit solutions.***

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

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

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

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

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

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

(e) Forty years later, 15th Doctor Ncuti Gatwa discovered the following optimization to WhoSort, which uses the standard Merge subroutine from mergesort, which merges two sorted arrays into one sorted array.

```
NuWhoSort(A[1..n]):
  if n < 13
      sort A by brute force
  else
      k = ⌈n/5⌉
      NuWhoSort(A[1..3k])                ⟨⟨Grant⟩⟩
      NuWhoSort(A[2k+1..n])              ⟨⟨Whittaker⟩⟩
      Merge(A[1..2k], A[2k+1..4k])       ⟨⟨Tennant⟩⟩
```

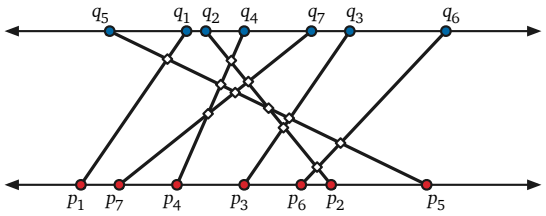
What is the running time of NuWhoSort?

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

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

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 half-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 2023

Homework 6

Due Tuesday, October 10, 2023 at 9pm Central Time

Please make sure that you read and understand the standard dynamic programming rubric.

---

1. Satya is in charge of establishing a new testing center for the Standardized Awesomeness Test (SAT), and found an old conference hall that is perfect. The conference hall has n rooms of various sizes along a single long hallway, numbered in order from 1 through n. Satya knows exactly how many students fit into each room, and he wants to use a subset of the rooms to host as many students as possible for testing.

   Unfortunately, there have been several incidents of students cheating at other testing centers by tapping secret codes through walls. To prevent this type of cheating, Satya can use two adjacent rooms only if he demolishes the wall between them. The city's chief architect has determined that demolishing the walls on both sides of the same room would threaten the building's structural integrity. For this reason, Satya can never host students in three consecutive rooms.

   Describe an efficient algorithm that computes the largest number of students that Satya can host for testing without using three consecutive rooms. The input to your algorithm is an array $S[1..n]$, where each $S[i]$ is the (non-negative integer) number of students that can fit in room i.

2. As a typical overworked college student, you occasionally pull all-nighters to get more work done. Painful experience has taught you that the longer you stay awake, the less productive you are.

   Suppose there are n days left in the semester. For each of the next n days, you can either stay awake and work, or you can sleep. You have an array $Score[1..n]$, where $Score[i]$ is the (always positive) number of points you will earn on day i *if you are awake and well-rested*.

   However, staying awake for several days in a row has a price: Each consecutive day you stay awake cuts the quality of your work in half. Thus, if you are awake on day i, and you most recently slept on day $i-k$, then you will actually earn $Score[i]/2^{k-1}$ points on day i. (You've already decided to sleep on day 0.)

   For example, suppose $n = 6$ and $Score = [3, 7, 4, 3, 9, 1]$.

   - If you work on all six days, you will earn $3 + \frac{7}{2} + \frac{4}{4} + \frac{3}{8} + \frac{9}{16} + \frac{1}{32} = 8.46875$ points.
   - If you work only on days 1, 3, and 5, you will earn $3 + 4 + 9 = 16$ points.
   - If you work only on days 2, 3, 5, and 6, you will earn $7 + \frac{4}{2} + 9 + \frac{1}{2} = 18.5$ points.

   Design and analyze an algorithm that computes the maximum number of points you can earn, given the array $Score[1..n]$ as input. For example, given the input array $[3, 7, 4, 3, 9, 1]$, your algorithm should return the number 18.5.

   VERY IMPORTANT: Do not actually do this in real life!

3. ***Practice only. Do not submit solutions.***

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

BUB • BASEESAB • ANANA

B • U • BB • ASEESA • B • ANANA

BUB • B • A • SEES • ABA • N • ANA

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

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

Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string BUBBASEESABANANA, your algorithm should return 3.

(b) A ***metapalindrome*** is a decomposition of a string into a sequence of palindromes, such that the sequence of palindrome lengths is itself a palindrome. For example, the string BOBSMAMASEESAUKULELE ("Bob's mama sees a ukulele") has the following metapalindromes (among others):

BOB • S • MAM • ASEESA • UKU • L • ELE

B • O • B • S • M • A • M • A • S • E • E • S • A • U • K • U • L • E • L • E

The length sequences of these metapalindromes are $(3, 1, 3, 6, 3, 1, 3)$ and $(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$; notice that both of these sequences are themselves palindromes.

Describe and analyze an efficient algorithm to find the smallest number of palindromes in any metapalindrome for a given string. For example, given the input string BOBSMAMASEESAUKULELE, your algorithm should return 7.

Solved Problems

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 **BAN**ANA**ANA**NAS B**A**N**AN**A**A**NA**NA**S

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

PRO**D**G**Y**R**NAM**AMMI**I**N**C**G **DY**PRO**N**G**A**R**M**AMM**IC**ING

(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 above and immediately to the left: $Shuf[i-1,j]$ and $Shuf[i,j-1]$. Thus, we can fill the array in standard row-major order in ***O(mn) time***. ■

Solution: The following algorithm runs in ***O(mn) time***.

```
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]
```

Here $Shuf(i,j) = \text{TRUE}$ if and only if the prefix $C[1..i+j]$ is a shuffle of the

prefixes $A[1..i]$ and $B[1..j]$. ■

Rubric: 5 points, standard dynamic programming rubric. **Each of these solutions is separately worth full credit.** These are not the only correct solutions. −½ for reporting running time as $O(n^2)$. 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.

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

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

Solution: The following algorithm runs in ***O(mn) time***:

```
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]
```

Here $\#Shuf[i, j]$ stores the number of different ways that the prefixes $A[1..i]$ and $B[1..j]$ can be shuffled to obtain the prefix $C[1..i+j]$. ■

Rubric: 5 points, standard dynamic programming rubric. **Again, each of these solutions is separately worth full credit.** These are not the only correct solutions. −½ for reporting running time as $O(n^2)$. 3 points for a slower polynomial-time algorithm; scale partial credit accordingly.

CS/ECE 374 A ✧ Fall 2023

Homework 7

Due Tuesday, October 17, 2023 at 9pm Central Time

---

1. The City Council of Sham-Poobanana needs to partition Purple Street into voting districts. A total of n people live on Purple Street, at consecutive addresses $1, 2, \ldots, n$. Each voting district must be a contiguous interval of addresses $i, i+1, \ldots, j$ for some $1 \le i < j \le n$. By law, each Purple Street address must lie in exactly one district, and the number of addresses in each district must be between k and $2k$, where k is a positive integer parameter.

   Every election in Sham-Poobanana is between two rival factions: Oceania and Eurasia. A majority of the current City Council are from Oceania, so they consider a district to be *good* if more than half the residents of that district voted for Oceania in the previous election. Naturally, the City Council has complete voting records for all n residents.

   For example, the figure below shows a legal partition of 22 addresses (of which 9 are good and 13 are bad) into 4 good districts and 3 bad districts, where $k = 2$ (so each district contains either 2, 3, or 4 addresses). Each **O** indicates a vote for Oceania, and each **X** indicates a vote for Eurasia.

   Describe an algorithm to find the largest possible number of *good* districts in a legal partition. Your input consists of the integer k and a boolean array GoodVote$[1..n]$ indicating which residents previously voted for Oceania (True) or Eurasia (False). You can assume that a legal partition exists. Analyze the running time of your algorithm in terms of the parameters n and k. (In particular, do **not** assume that k is a constant.)

2. The StupidScript language includes a binary operator `@` that computes the *average* of its two arguments. For example, the StupidScript code `print(3 @ 6)` would print `4.5`, because $(3+6)/2 = 4.5$.

   Expressions like 3 @ 7 @ 4 that use the @ operator more than once yield different results when they are evaluated in different orders:

$$(3 \mathbin{@} 7) \mathbin{@} 4 = 5 \mathbin{@} 4 = 4.5 \qquad \text{but} \qquad 3 \mathbin{@} (7 \mathbin{@} 4) = 3 \mathbin{@} 5.5 = 4.25$$

   Here is a larger example:

$$\begin{aligned}((((8 \mathbin{@} 6) \mathbin{@} 7) \mathbin{@} 5) \mathbin{@} 3) \mathbin{@} (0 \mathbin{@} 9) &= 4.5\\ ((8 \mathbin{@} 6) \mathbin{@} (7 \mathbin{@} 5)) \mathbin{@} ((3 \mathbin{@} 0) \mathbin{@} 9) &= 5.875\\ (8 \mathbin{@} (6 \mathbin{@} (7 \mathbin{@} (5 \mathbin{@} (3 \mathbin{@} 0))))) \mathbin{@} 9 &= 7.890625\end{aligned}$$

   Your goal for this problem is to describe and analyze an algorithm to compute, given a sequence of integers separated by @ signs, the ***largest possible*** value the expression can take by adding parentheses. Your input is an array $A[1..n]$ listing the sequence of integers.

   For example, if your input sequence is $[3, 7, 4]$, your algorithm should return 4.5, and if your input sequence is $[8, 6, 7, 5, 3, 0, 9]$, your algorithm should return 7.890625. Assume all arithmetic operations (including @) can be performed exactly in $O(1)$ time.

   (a) Tommy Tutone suggests the following natural greedy algorithm: Merge the adjacent pair of numbers with the smallest average (breaking ties arbitrarily), replace them with their average, and recurse. For example:

$$\begin{aligned}&8 \mathbin{@} 6 \mathbin{@} 7 \mathbin{@} 5 \mathbin{@} \underline{3 \mathbin{@} 0} \mathbin{@} 9\\ &8 \mathbin{@} 6 \mathbin{@} 7 \mathbin{@} \underline{5 \mathbin{@} 1.5} \mathbin{@} 9\\ &8 \mathbin{@} 6 \mathbin{@} \underline{7 \mathbin{@} 3.25} \mathbin{@} 9\\ &8 \mathbin{@} \underline{6 \mathbin{@} 5.125} \mathbin{@} 9\\ &\underline{8 \mathbin{@} 5.5625} \mathbin{@} 9\\ &\underline{6.78125 \mathbin{@} 9}\\ &7.890625\end{aligned}$$

   Tommy reasons that with an efficient priority queue, this algorithm will run in $O(n \log n)$ time, which is *way* faster than any dynamic programming algorithm.

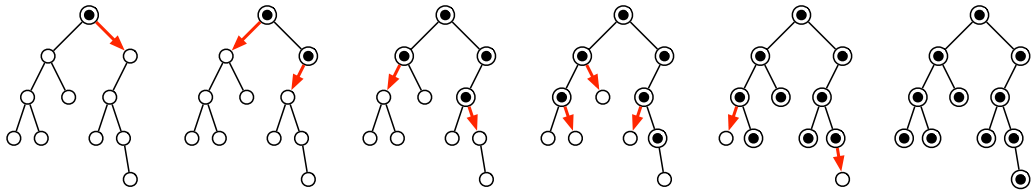
   Prove that Tommy's algorithm is incorrect, by describing a specific input array and proving that his algorithm does not yield the largest possible value for that array.

   (b) Describe and analyze a correct algorithm for this problem. Poor, poor Tommy.

3. ***Practice only. Do not submit solutions.***

Suppose we need to broadcast a message to all the nodes in a rooted binary tree. Initially, only the root node knows the message. In a single round, any node that knows the message can forward it to at most one of its children. See the figure below for an example.

Design an algorithm to compute the minimum number of rounds required to broadcast the message to every node.



A message being distributed through a binary tree in five rounds.

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 $w = \texttt{([()][]())[()()]()}$ 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:

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

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 $LBS[i,k]$ depends only on entries in later rows or earlier columns (or both), we can fill this array row-by-row from bottom up (decreasing i) in the outer loop, scanning each row from left to right (increasing k) in the inner loop.

We can compute each entry $LBS[i,k]$ in $O(n)$ time, so the resulting algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ■

Solution (pseudocode): The following 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] = ( and A[k] = )) or (A[i] = [ and 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]
```

Here $LBS[i, k[$ stores the length of the longest balanced subsequence of the substring $A[i..k]$. ■

Rubric: 10 points, standard dynamic programming rubric. Yes, each of these solutions is independently worth full credit.

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 $\sum \varnothing = 0$.[a]

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.

The resulting algorithm spends $O(1)$ time at each node of T, and therefore runs in ***O(n) time***. ■

[a] A naïve recursive implementation of these recurrences 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 occurs when T is a single path.

Solution (two functions, pseudocode): The following algorithm runs in ***O(n) time***.

```
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}
```

We are storing two pieces of information in each node v of the tree:

- $v.yes$ is the maximum total "fun" of a legal party among the descendants of v, assuming v is invited.
- $v.no$ is the maximum total "fun" of a legal party among the descendants of v, assuming v is not invited.

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

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.

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. ■

Solution (one function, pseudocode):

```
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}
```

Here $v.maxFun$ stores the maximum total "fun" of a legal party among the descendants of v, where v may or may not be invited.

Each value $v.maxFun$ is read at most three times during the algorithm's execution: Once in ComputeMaxFun($v.parent$), and once in ComputeMaxFun($v.parent.parent$), and at most once in the non-recursive part of BestParty. Thus, the entire algorithm runs in $O(n)$ *time*. ■

Rubric: 10 points: standard dynamic programming rubric. These are not the only correct solutions. Yes, each of these solutions is independently worth full credit.

CS/ECE 374 A ✧ Fall 2023

Homework 8

Due Tuesday, October 24, 2023 at 9pm Central Time

---

1. A *six-sided die* (plural *dice*) is a cube with each side marked with a different number of dots (called *pips*) from 1 to 6. On a *standard* die, numbers on opposite sides always add up to 7.

   A *rolling die maze* is a puzzle involving a standard six-sided die and a grid of squares. You should imagine the grid lying on a table; the die always rests on and exactly covers one square of the grid. In a single step, you can *roll* the die 90 degrees around one of its bottom edges, moving it to an adjacent square one step north, south, east, or west.

   Some squares in the grid may be *blocked*; the die can never rest on a blocked square. Other squares may be *labeled* with a number; whenever the die rests on a labeled square, the number on the *top* face of the die must equal the label. Squares that are neither labeled nor marked are called *free*. You may not roll the die off the edges of the grid. A rolling die maze is *solvable* if it is possible to place a die on the lower left square and roll it to the upper right square under these constraints.

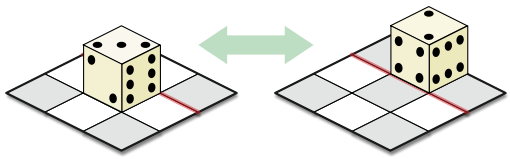


Figure 1. Rolling a (right-handed) die

   Figure 2 shows five rolling die mazes. The first two mazes are solvable using any standard die. Specifically, the first maze can be solved by placing the die on the lower left square with 1 on the top face, and then rolling the die east, north, north, east; the second maze can be solved in 12 moves. The third maze is only solvable using a *right-handed* die, where faces 1, 2, 3 appear in counterclockwise order around a common corner.[1] The last two mazes cannot be solved even with non-standard dice.

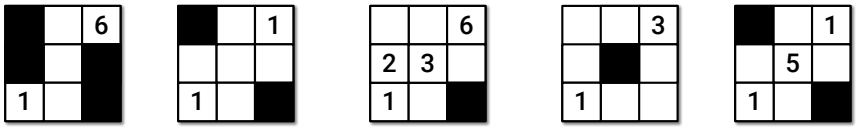


Figure 2. Five rolling die mazes.

   Describe and analyze an algorithm that determines whether a given rolling die maze can be solved with a right-handed standard die. Your input is a two-dimensional array $Label[1..n, 1..n]$, where each entry $Label[i, j]$ stores the label of the square in the ith row and jth column, where 0 means the square is free and -1 means the square is blocked.

   [Hint: You have some freedom in how to place the initial die. There are rolling die mazes that can be solved only if the initial placement is chosen correctly. Describe your solution in high-level language; don't get bogged down in grungy case analysis.]

[1] Right-handed dice are more common in the Western hemisphere; left-handed dice are more common in east Asia.

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

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

   The children of Cheery Hells have published a secret web site listing the amount of candy that each house in Cheery Hells will give to each visitor.

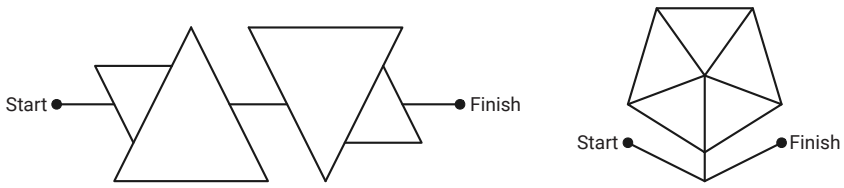
   Describe and analyze an algorithm to compute the maximum amount of candy that a single child can obtain in a walk through Cheery Hells, starting at the entrance node s. The input to your algorithm is the directed graph G, along with a non-negative integer $v.candy$ for each vertex v, describing the amount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

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

3. ***Practice only. Do not submit solutions.***

   One of my daughter's elementary-school math workbooks[2] contains several puzzles of the following type:

   > Complete each angle maze below by tracing a path from Start to Finish that has only acute angles.



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

   Your input is a connected undirected graph G, whose vertices are points in the plane and whose edges are straight line segments. Edges do not intersect, except at their common endpoints. For example, a drawing of the letter X would have five vertices and four edges, and the first maze above has 18 vertices and 21 edges. You are also given two vertices Start and Finish.

   Your algorithm should return TRUE if G contains a walk from Start to Finish that has only acute angles, and FALSE otherwise. Formally, a walk through G is valid if, for any two

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

consecutive edges $u \to v \to w$ in the walk, either $\angle uvw = \pi$ (straight) or $0 < \angle uvw < \pi/2$ (acute). Assume you have a subroutine that can determine in $O(1)$ time whether the angle between two given segments is straight, obtuse, right, or acute.

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 change the first configuration into 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

- $\left.\begin{cases}(0, a+b, c) & \text{if } a+b \le B \\ (a+b-B, B, c) & \text{if } a+b \ge B\end{cases}\right\}$ — pouring from jar 1 into jar 2
- $\left.\begin{cases}(0, b, a+c) & \text{if } a+c \le C \\ (a+c-C, b, C) & \text{if } a+c \ge C\end{cases}\right\}$ — pouring from jar 1 into jar 3
- $\left.\begin{cases}(a+b, 0, c) & \text{if } a+b \le A \\ (A, a+b-A, c) & \text{if } a+b \ge A\end{cases}\right\}$ — pouring from jar 2 into jar 1
- $\left.\begin{cases}(a, 0, b+c) & \text{if } b+c \le C \\ (a, b+c-C, C) & \text{if } b+c \ge C\end{cases}\right\}$ — pouring from jar 2 into jar 3
- $\left.\begin{cases}(a+c, b, 0) & \text{if } a+c \le A \\ (A, b, a+c-A) & \text{if } a+c \ge A\end{cases}\right\}$ — pouring from jar 3 into jar 1
- $\left.\begin{cases}(a, b+c, 0) & \text{if } b+c \le B \\ (a, B, b+c-B) & \text{if } b+c \ge B\end{cases}\right\}$ — 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 speed up this algorithm by observing that every move leaves at least one jar either completely empty or completely 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 2023

Homework 9

Due Tuesday, October 31, 2023 at 9pm

This is the last homework before Midterm 2.

---

1. 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. 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 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 to *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?

2. You are driving through the back-country roads of Tenkucky, desperately trying to leave the state before the state's annual Halloween Purge begins. Every road in the state is patrolled by a Driving Posse who will let you exercise your god-given right to drive as fast as you damn well please, provided you pay the appropriate speed tax. The faster you traverse any road, the more you have to pay. What's the fastest way to escape the state?

   You have an accurate map of the state, in the form of a directed graph $G = (V, E)$, whose vertices V represent small towns and whose edges E represent one-lane dirt roads between towns.[1] One vertex s is marked as your starting location; a subset $X \subset V$ of vertices are marked as exits. Each edge e has an associated value $\$(e)$ with the following interpretation.

   - If you drive from one end of road e to the other in m minutes, for any positive real number m, then you must pay road e's Driving Posse a speed tax of $\lceil \$(e)/m \rceil$ dollars.

[1] Paved roads are far too expensive!

- Equivalently, if you pay road e's Driving Posse a speed tax of d dollars, for any positive integer d, you are allowed to drive the entire length of road e in $\$(e)/d$ minutes, but no less.

In particular, any road you drive on *at all* will cost you *at least* one dollar. Anyone who violates this rule (for example, by running out of money) will be thrown in jail, which means almost certain death in the Purge.

The Driving Posses do not accept coins, credit cards, Venmo, Zelle, or any other mobile payment app—only cold hard American paper currency—and they do not give change. Fortunately, you are starting your journey with a pile of D crisp new $1 bills.

Describe and analyze and algorithm to compute the fastest possible driving route from s to any exit node in X. The input to your algorithm consists of the map $G = (V, E)$, the start vertex s, the exit vertices X, and the positive integer D. Report the running time of your algorithm as a function of the parameters V, E, and D.

3. ***Practice only. Do not submit solutions.***

After a grueling midterm at the See-Bull Center for Commuter Silence, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Sham-Poobanana. Unfortunately, no single bus visits both the See-Bull Center and your home; you must change buses at least once. There are exactly b different buses. Each bus starts at 12:00:01AM, makes exactly n stops, and finally stops running at 11:59:59PM. Buses always run exactly on schedule, and you have an accurate watch. Finally, you are far too tired to walk between bus stops.

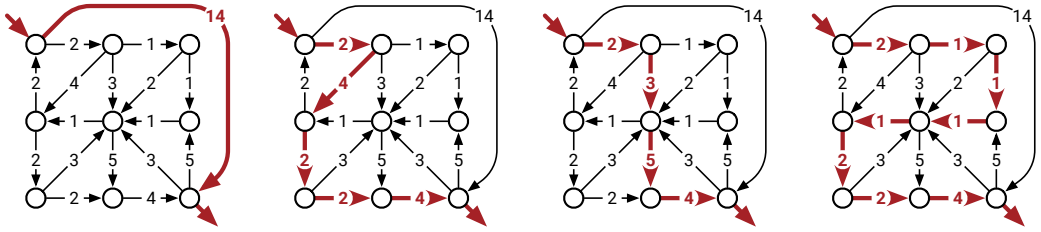
(a) Describe and analyze an algorithm to determine a sequence of bus rides that gets you home as early as possible. Your goal is to minimize your *arrival time*, not the time you spend traveling.

(b) Oh, no! The midterm was held on Halloween, and the streets are infested with zombies! Describe how to modify your algorithm from part (a) to minimize *the total time you spend waiting at bus stops*; you don't care how late you get home or how much time you spend on buses. (Assume you can wait inside the See-Bull Center until your first bus is just about to leave.)

For both questions, your input consists of the exact time when the midterm ends See-Bull and two arrays $Time[1..b, 1..n]$ and $Stop[1..b, 1..n]$, where $Time[i, j]$ is the scheduled time of the ith bus's jth stop, and $Stop[i, j]$ is the location of that stop. Report the running times of your algorithms as functions of the parameters n and b.

Solved Problems

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



Four (of many) equal-length shortest paths.

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

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

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

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

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

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

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

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

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

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

5. After moving to a new city, you decide to choose a walking route from your home to your new office. Your route must consist of an uphill path (for exercise) followed by a downhill path (to cool down), or just an uphill path, or just a downhill path. But you also want the *shortest* path that satisfies these conditions, so that you actually get to work on time.

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

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

Solution: We define a new directed graph $G' = (V', E')$ as follows:

- $V' = \{v^\uparrow, v^\downarrow \mid V \in V\}$. Vertex $v^\uparrow$ indicates that we are at intersection v moving uphill, and vertex $v^\downarrow$ indicates that we are at intersection v moving downhill.
- E' is the union of three sets:
  - Uphill edges: $\{u^\uparrow \to v^\uparrow \mid uv \in E \text{ and } h(u) < h(v)\}$. Each uphill edge $u^\uparrow \to v^\uparrow$ has weight $\ell(uv)$.
  - Downhill edges: $\{u^\downarrow \to v^\downarrow \mid uv \in E \text{ and } h(u) > h(v)\}$. Each downhill edge $u^\downarrow \to v^\downarrow$ has weight $\ell(uv)$.
  - Switch edges: $\{v^\uparrow \to v^\downarrow \mid v \in V\}$; each switch edge has weight 0.

We need to compute three shortest paths in this graph:

- The shortest path from $s^\uparrow$ to $t^\downarrow$ gives us the best uphill-then-downhill route.
- The shortest path from $s^\uparrow$ to $t^\uparrow$ gives us the best uphill-only route.
- The shortest path from $s^\downarrow$ to $t^\downarrow$ gives us the best downhill-only route.

G' is a directed ***acyclic*** graph; we can get a topological ordering by listing the up vertices $v^\uparrow$, sorted by increasing height, followed by the down vertices $v^\downarrow$, sorted by decreasing height. Thus, we can compute the shortest path in G' from any vertex to any other in $O(V' + E') = O(V + E)$ time by dynamic programming. (The algorithm is the same as the longest-path algorithm in the notes, except we use "min" instead of "max" in the recurrence, and define $\min \emptyset = \infty$.)

Our overall algorithm runs in $\boldsymbol{O(V + E)}$ ***time***. ■

Rubric: 5 points = 1 for vertices + 1 for edges + 1 for arguing G' is a dag + 1 for algorithm + 1 for running time. This is not the only correct solution. Max 4 points for a correct reduction to Dijkstra's algorithm that runs in $O(E \log V)$ time.

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

Solution (Dijkstra, 5/5): Let $L = 1 + \sum_{u \to v} \ell(u \to v)$. Define a new graph $G' = (V', E')$ as follows:

- $V' = \{v^\uparrow, v^\downarrow \mid V \in V\} \cup \{s, t\}$. Vertex $v^\uparrow$ indicates that we are at intersection v moving uphill, and vertex $v^\downarrow$ indicates that we are at intersection v moving downhill.
- E' contains four types of edges:
  - Uphill edges: $\{u^\uparrow \to v^\uparrow \mid uv \in E \text{ and } h(u) < h(v)\}$. Each uphill edge $u^\uparrow \to v^\uparrow$ has weight $\ell(uv)$.
  - Downhill edges: $\{u^\downarrow \to v^\downarrow \mid uv \in E \text{ and } h(u) > h(v)\}$. Each downhill edge $u^\downarrow \to v^\downarrow$ has weight $\ell(uv)$.
  - Switch edges: $\{v^\uparrow \to v^\downarrow \mid v \in V\} \cup \{v^\downarrow \to v^\uparrow \mid v \in V\}$. Each switch edge has weight L.
  - Start and end edges $s \to s^\uparrow$, $s \to s^\downarrow$, $t^\uparrow \to t$, and $t^\downarrow \to t$, each with weight 0,

We need to compute the shortest path from s to t in G'; the large weight L on the switch edges guarantees that this path with have the minimum number of switches, and the minimum length among all paths with that number of switches. Dijkstra's algorithm finds this shortest path in $O(E' \log V') = \boldsymbol{O(E \log V)}$ ***time***.

(Because G' includes switch edges in both directions, G' is not a dag, so we can't use dynamic programming directly.) ■

Rubric: 5 points, standard graph-reduction rubric. This is not the only correct solution with running time $O(E \log V)$.

Solution (clever, extra credit): Our algorithm works in two phases: First we determine the minimum number of switches required to reach every vertex, and then we compute the shortest path from s to t with the minimum number of switches. The first phase is can be solved in $O(V + E)$ time by a modification of breadth-first search; the second by computing shortest paths in a dag.

For the first phase, we define a new graph $G' = (V', E')$ as follows:

- $V' = \{v^\uparrow, v^\downarrow \mid V \in V\} \cup \{s, t\}$. Vertex $v^\uparrow$ indicates that we are at intersection v moving uphill, and vertex $v^\downarrow$ indicates that we are at intersection v moving downhill.
- E' contains four types of edges:
  - Uphill edges: $\{u^\uparrow \to v^\uparrow \mid uv \in E \text{ and } h(u) < h(v)\}$. Each uphill edge has weight 0.
  - Downhill edges: $\{u^\downarrow \to v^\downarrow \mid uv \in E \text{ and } h(u) > h(v)\}$. Each downhill

edge has weight 0.

- Switch edges: $\{v^{\uparrow}\to v^{\downarrow} \mid v \in V\} \cup \{v^{\downarrow}\to v^{\uparrow} \mid v \in V\}$. Each switch edge has weight 1.
- Start and end edges $s\to s^{\uparrow}$, $s\to s^{\downarrow}$, $t^{\uparrow}\to t$, and $t^{\downarrow}\to t$, each with weight 0.

Now we compute the shortest path distance from s to every other vertex in G'. We could use Dijkstra's algorithm in $O(E \log V)$ time, but the structure of the graph supports a faster algorithm.

Intuitively, we break the shortest-path computation into phases, where in the kth phase, we mark all vertices at distance k from the source vertex s. During the kth phase, we may also discover vertices at distance $k+1$, but no further. So instead of using a binary heap for the priority queue, it suffices to use two bags: one for vertices at distance k, and one for vertices at distance $k+1$.

```
ZeroOneDijkstra(G, ℓ, s):
  s.dist ← 0
  for all vertices v ≠ s
      v.dist ← ∞
  curr ← new empty bag
  add s to curr
  for k ← 0 to V
      next ← new empty bag
      while curr is not empty
          take v from curr            ⟨⟨v.dist = k⟩⟩
          for all edges v→w
              if w.dist > v.dist + ℓ(v→w)
                  w.dist ← v.dist + ℓ(v→w)
                  if ℓ(v→w) = 0
                      add w to curr
                  else ⟨⟨if ℓ(v→w) = 1⟩⟩
                      add w to next
      curr ← next
```

This phase of the algorithm runs in $O(V' + E') = O(V + E)$ time.

Once we have computed distances in G', we construct a second graph $G'' = (V', E'')$ with the same vertices as G', but only a subset of the edges:

$$E'' = \left\{u'\to v' \in E' \;\middle|\; u'.dist + \ell(u'\to v') = v'.dist\right\}$$

Equivalently, an edge $u'\to v'$ belongs to E'' if and only if that edge is part of at least one shortest path in G' from s to another vertex. It follows (by induction, of course), that every path in G'' from s to another vertex v' is a shortest path in G', and therefore a minimum-switch path in G.

We also reassign the edge weights in G''. Specifically, we assign each uphill edge $u^{\uparrow}\to v^{\uparrow}$ and downhill edge $u^{\downarrow}\to v^{\downarrow}$ in G'' weight $\ell(uv)$, and we assign every switch edge, start edge, and end edge weight 0. **Now we need to compute the shortest path from s to t in G'', with respect to these new edge weights.**

We can expand the definition of E'' in terms of the original input graph as follows:

$$\begin{aligned} E'' = &\left\{u^{\uparrow} \to v^{\uparrow} \;\middle|\; uv \in E \text{ and } h(u) < h(v) \text{ and } u^{\uparrow}.dist = v^{\uparrow}.dist\right\} \\ &\cup \left\{u^{\downarrow} \to v^{\downarrow} \;\middle|\; uv \in E \text{ and } h(u) > h(v) \text{ and } u^{\downarrow}.dist = v^{\downarrow}.dist\right\} \\ &\cup \left\{v^{\uparrow} \to v^{\downarrow} \;\middle|\; v \in V \text{ and } v^{\uparrow}.dist < v^{\downarrow}.dist\right\} \\ &\cup \left\{v^{\downarrow} \to v^{\uparrow} \;\middle|\; v \in V \text{ and } v^{\downarrow}.dist < v^{\uparrow}.dist\right\} \end{aligned}$$

We can topologically sort G'' by first sorting the vertices by increasing $v'.dist$, and then within each subset of vertices with equal $v'.dist$, listing the up-vertices by increasing height, followed by the down vertices by decreasing height. It follows that G'' is a dag! Thus, we can compute shortest paths in G'' in $O(V'' + E'') = O(V + E)$ time, using the same dynamic programming algorithm that we used in part (a).

The overall algorithm runs in $\boldsymbol{O(V+E)}$ ***time***. ■

Rubric: max 10 points =

- 5 for computing minimum-switch paths = 1 for vertices + 1 for edges (including weights) + 2 for 0/1 shortest path algorithm + 1 for running time.
- 5 for computing shortest minimum-switch paths = 1 for vertices + 1 for edges (including weights) + 1 for proving dag + 1 for dynamic programming algorithm + 1 for running time

CS/ECE 374 A ✧ Fall 2023

Homework 10

Due Tuesday, November 14, 2019 at 9pm

---

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

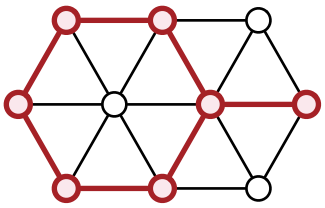
   - SUBSETSUM: Given a set S of positive integers and a target integer T, is there a subset of S whose sum is T?
   - PARTITION: Given a set S of positive integers, is there a way to partition S into two subsets S_1 and S_2 that have the same sum?

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

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

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

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



A triangle-free subset of 7 vertices and its induced edges.
This is ***not*** the largest triangle-free subset in this graph.

Solved Problem

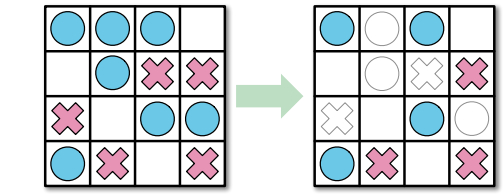
4. ***RedBlue*** is a puzzle that consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions:

   (1) Every row contains at least one stone.

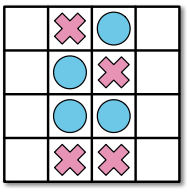
   (2) No column contains stones of both colors.

   For some RedBlue puzzles, reaching this goal is impossible; see the example below.

   Prove that it is NP-hard to determine whether a given RedBlue puzzle has a solution.



A solvable RedBlue puzzle and one of its many solutions.



An unsolvable RedBlue puzzle.

Solution: We show that RedBlue 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 RedBlue instance X 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.

To prove that RedBlue is NP-hard, it suffices to prove the following claim:

Φ is satisfiable if and only if RedBlue puzzle X is solvable.

$\Longrightarrow$ 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 RedBlue puzzle X is solvable.

⟸ On the other hand, suppose RedBlue puzzle X 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. ■

Standard NP-hardness rubric. 10 points =

+ 1 point for choosing a reasonable NP-hard problem X to reduce from.
  - The Cook-Levin theorem implies that *in principle* one can prove NP-hardness by reduction from *any* NP-complete problem. What we're looking for here is a problem where a simple and direct NP-hardness proof seems likely.
  - You can use any of the NP-hard problems listed on the next page or in the textbook (except the one you are trying to prove NP-hard, of course).
+ 2 points for a *structurally sound* polynomial-time reduction. Specifically, the reduction must:
  - take an *arbitrary* instance of the declared problem X ***and nothing else*** as input,
  - transform that input into a corresponding instance of Y (the problem we're trying to prove NP-hard),
  - transform the output of the magic algorithm for Y into a reasonable output for X, and
  - run in polynomial time.

  (The output transformation is usually trivial.) This is strictly about the structure of the reduction algorithm, not about its correctness. **No credit for the rest of the problem if this is wrong.**
+ 2 points for a *correct* polynomial-time reduction. That is, assuming a black-box algorithm that solves Y in polynomial time, the proposed reduction actually solves problem X in polynomial time.
+ 2 points for the "if" proof of correctness. (Every good instance of X is transformed into a good instance of Y.)
+ 2 points for the "only if" proof of correctness. (Every bad instance of X is transformed into a bad instance of Y.)
+ 1 point for writing "polynomial time"

- An incorrect but structurally sound polynomial-time reduction that still satisfies half of the correctness proof is worth at most 6/10.
- A reduction in the wrong direction is worth at most 1/10.

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

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

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

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

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

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

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

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

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

ChromaticNumber: Given an undirected graph G, what is the minimum number of colors required to color its vertices, 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 2023

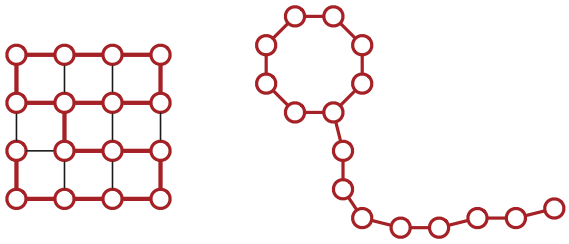
Homework 11

Due Tuesday, November 28, 2023 at 9pm

This is the last graded homework before the final exam.

---

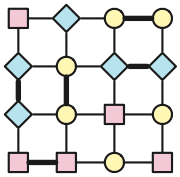
1. A *balloon* of size ℓ is an undirected graph consisting of a (simple) cycle of length ℓ and a (simple) path of length ℓ, where one endpoint of the path lies on the cycle, and otherwise the cycle and the path are disjoint. Every balloon of size ℓ has exactly 2ℓ vertices and 2ℓ edges. For example, the 4×4 grid graph shown below contains a balloon subgraph of size 8.



Prove that it is NP-hard to find the size of the the largest balloon subgraph of a given undirected graph.

2. Recall that a *3-coloring* of a graph assigns each vertex one of three colors, say red, yellow, and blue. A 3-coloring is *proper* if every edge has endpoints with different colors. The 3Color problem asks, given an arbitrary undirected graph G, whether G has a proper 3-coloring.

   Call a 3-coloring of a graph G *slightly improper* if each vertex has *at most one neighbor* with the same color. The SlightlyImproper3Color problem asks, given an arbitrary undirected graph G, whether G has a slightly improper 3-coloring.



   (a) Consider the following attempt to prove that SlightlyImproper3Color is NP-hard, using a reduction from 3Color.

   Non-solution: We reduce from 3Color. Given an arbitrary input graph G, we construct a new graph H by attaching a clique of 4 vertices to every vertex of G. Specifically, for each vertex v in G, the graph H contains three new vertices v_1, v_2, v_3, along with edges $vv_1, vv_2, vv_3, v_1v_2, v_1v_3, v_2v_3$. I claim that

G has a proper 3-coloring
if and only if
H has a slightly improper 3-coloring.

$\Longrightarrow$ Suppose G has a proper 3-coloring, using the colors red, yellow, and blue. Extend this color assignment to the vertices of H by coloring each vertex v_1 red, each vertex v_2 yellow, and each vertex v_3 blue. With this assignment, each vertex of H has at most one neighbor with the same color. Specifically, each vertex of G has the same color as one of the vertices in its gadget, and the other two vertices in v's gadget have no neighbors with the same color.

$\Longleftarrow\!\!\!\!\!\!\times$ Now suppose H has a slightly improper 3-coloring. Then G must have a proper 3-coloring because. . . um. . .

♣

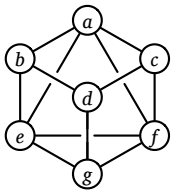
Describe a graph G that does not have a proper 3-coloring, such that the graph H constructed by this reduction does have a slightly improper 3-coloring.

(b) Describe a small graph X with the following property: In every slightly improper 3-coloring of X, every vertex of X has *exactly* one neighbor with the same color.

(c) Describe a correct polynomial-time reduction from 3Color to SlightlyImproper-3Color. *[Hint: Use your graph from part (b) as a gadget.]* This reduction will prove that SlightlyImproper3Color is indeed NP-hard.

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.

Now I claim that

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

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

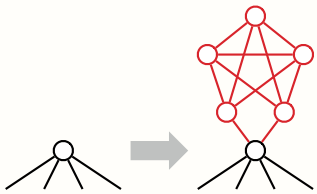
$$\cdots \to v_i \to v_i^\flat \to v_i^\sharp \to v_i^\flat \to v_i^\sharp \to v_i \to \cdots$$

$\Longleftarrow$ 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 remove the vertices and edges in $H \setminus G$ from D, we obtain a closed walk in G that visits every vertex in G exactly once.

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

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.

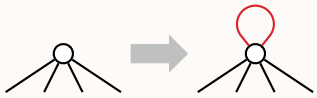


A vertex in G, and the corresponding modified vertex gadget in H.

■

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.

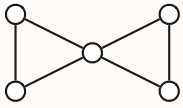
Now I claim that

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 alternating between edges of the Hamiltonian cycle and self-loops: $v_1 \to v_1 \to v_2 \to v_2 \to v_3 \to \cdots \to v_n \to v_n \to v_1$.

$\not\Longleftarrow$ Um. . .

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?

ChromaticNumber: Given an undirected graph G, what is the minimum number of colors required to color its vertices, 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 2023

☙ "Homework" 12 ❧

"Due" Monday, December 4, 2023

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.

Solved Problem

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

   (a) Prove that SOMETIMESHALT is undecidable.

Solution (Rice): Let $\mathcal{L}$ be the family of all non-empty languages. Let N be any Turing machine that never halts, so $\text{HALT}(N) = \emptyset \notin \mathcal{L}$. Let Y be any Turing machine that always halts, so $\text{HALT}(Y) = \Sigma^* \in \mathcal{L}$. Rice's Halting Theorem immediately implies that SOMETIMESHALT $=$ HALTIN$(\mathcal{L})$ is undecidable. ■

Solution (closure): Let ENCODINGS be the language of all Turing machine encodings (for some fixed universal Turing machine); this language is decidable. We immediately have ENCODINGS $=$ NEVERHALT $\cup$ SOMETIMESHALT, or equivalently, NEVERHALT $=$ ENCODINGS $\setminus$ SOMETIMESHALT.

The lectures notes include a proof that NEVERHALT is undecidable. On the other hand, the existence of a universal Turing machine implies that ENCODINGS is decidable. So Corollary 3(d) in the undecidability notes implies that SOMETIMESHALT is undecidable. ■

Solution (reduction from HALT): We can reduce the standard halting problem to SOMETIMESHALT as follows:

```
DECIDEHALT(⟨M, w⟩):
  Encode the following Turing machine M′:
    M′(x):
      (ignore x)
      run M on input w
  return DECIDESOMETIMESHALT(⟨M′⟩)
```

We prove this reduction correct as follows:

$\Longrightarrow$ Suppose M halts on input w.
Then M' halts on *every* input string x.
So DECIDESOMETIMESHALT must accept the encoding $\langle M' \rangle$.
We conclude that DECIDEHALT correctly accepts the encoding $\langle M, w \rangle$.

$\Longleftarrow$ Suppose M does not halt on input w.
Then M' diverges on *every* input string x.
So DECIDESOMETIMESHALT must reject the encoding $\langle M' \rangle$.
We conclude that DECIDEHALT correctly rejects the encoding $\langle M, w \rangle$.

■