• Submit your written solutions electronically to Gradescope as PDF files. Submit a separate PDF file for each numbered problem. If you plan to typeset your solutions, please use the LaTeX solution template on the course web site. If you must submit scanned handwritten solutions, please use a black pen on blank white paper and a high-quality scanner app (or an actual scanner).

• Groups of up to three people can submit joint solutions on Gradescope. Exactly one student in each group should upload the solution and indicate their other group members. All group members must be already registered on Gradescope.

• You are not required to sign up on Gradescope or Piazza with your real name and your illinois.edu email address; you may use any email address and alias of your choice. However, to give you credit for the homework, we need to know who Gradescope thinks you are. Please fill out the web form linked from the course web page.

• You may use any source at your disposal—paper, electronic, or human—but you must cite every source that you use, and you must write everything yourself in your own words. See the academic integrity policies on the course web site for more details.

• Written homework will be due every Tuesday at 8pm, except in weeks with exams. In addition, guided problems sets on PrairieLearn are due every Monday at 8pm; each student must do these individually. In particular, Guided Problem Set 1 is due Monday, August 30! Each Guided Problem Set has the same weight as one numbered homework problem.

See the course web site for more information.

If you have any questions about these policies, please don’t hesitate to ask in class, in office hours, or on Piazza.
1. Consider the following pair of mutually recursive functions on strings:

\[
\begin{align*}
\text{odds}(w) &:= \begin{cases} 
\epsilon & \text{if } w = \epsilon \\
 a \cdot \text{evens}(x) & \text{if } w = ax
\end{cases} \\
\text{evens}(w) &:= \begin{cases} 
\epsilon & \text{if } w = \epsilon \\
 \text{odds}(x) & \text{if } w = ax
\end{cases}
\end{align*}
\]

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

\[
\begin{align*}
\text{evens}(\text{PARITY}) & = \text{odds}(\text{ARITY}) \\
& = A \cdot \text{evens}(\text{RITY}) \\
& = A \cdot \text{odds}(\text{ITY}) \\
& = A \cdot (I \cdot \text{evens}(\text{TY})) \\
& = A \cdot (I \cdot \text{odds}(\text{Y})) \\
& = A \cdot (I \cdot (Y \cdot \text{evens}(\epsilon))) \\
& = A \cdot (I \cdot (Y \cdot \epsilon)) \\
& = \text{AIY}
\end{align*}
\]

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

(a) Give a self-contained recursive definition for the function \(\text{evens}\) that does not involve the function \(\text{odds}\).

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

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

You may assume without proof any result proved in class, in lab, or in the lecture notes. Otherwise, your proofs must be formal and self-contained, and they must invoke the formal definitions of concatenation \(\cdot\), length \(|\cdot|\), and the \text{evens} and \text{odds} functions. Do not appeal to intuition!
2. Consider the following recursive function that perfectly shuffles two strings together:

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

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

\[
shuffle(PRT, AIY) = P \cdot shuffle(AIY, RT) = P \cdot (A \cdot shuffle(RT, IY)) = P \cdot (A \cdot (R \cdot shuffle(IY, T))) = P \cdot (A \cdot (R \cdot (I \cdot shuffle(T, Y)))) = P \cdot (A \cdot (R \cdot (I \cdot (T \cdot shuffle(Y, \varepsilon)))))) = P \cdot (A \cdot (R \cdot (I \cdot (T \cdot (Y \cdot shuffle(\varepsilon, \varepsilon)))))) = P \cdot (A \cdot (R \cdot (I \cdot (T \cdot (Y \cdot \varepsilon))))) = \text{PARITY}
\]

(a) Prove that \(shuffle(\text{odds}(w), \text{evens}(w)) = w\) for every string \(w\).

(b) Prove \(\text{evens}(shuffle(w, z)) = z\) for all strings \(w\) and \(z\) such that \(|w| = |z|\).

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

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

**Standard induction rubric.** For problems worth 10 points:

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

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

Solved Problems

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

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

$$\text{swap}(101100101) = 0111001011.$$
The swap function can be formally defined as follows:

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

(a) Prove that \(|\text{swap}(w)| = |w|\) for every string \(w\).

**Solution:** Let \(w\) be an arbitrary string.

Assume \(|\text{swap}(x)| = |x|\) for every string \(x\) that is shorter than \(w\).

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

- If \(w = \varepsilon\), then

  \[
  |\text{swap}(w)| = |\text{swap}(\varepsilon)| \\
  = |\varepsilon| \quad \text{because } w = \varepsilon \\
  = |w| \quad \text{by definition of } \text{swap}
  \]

- If \(w = 0 \text{ or } w = 1\), then

  \[
  |\text{swap}(w)| = |w| \quad \text{by definition of } \text{swap}
  \]

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

  \[
  |\text{swap}(w)| = |\text{swap}(abx)| \quad \text{because } w = abx \\
  = |ba \cdot \text{swap}(x)| \quad \text{by definition of } \text{swap} \\
  = |ba| + |\text{swap}(x)| \quad \text{because } |y \cdot z| = |y| + |z| \\
  = |ba| + |x| \quad \text{by the induction hypothesis} \\
  = 2 + |x| \quad \text{by definition of } |\cdot| \\
  = |ab| + |x| \quad \text{by definition of } |\cdot| \\
  = |ab \cdot x| \quad \text{because } |y \cdot z| = |y| + |z| \\
  = |abx| \quad \text{by definition of } \cdot \\
  = |w| \quad \text{because } w = abx
  \]

In all cases, we conclude that \(|\text{swap}(w)| = |w|\). ■

**Rubric:** 5 points: Standard induction rubric (scaled). This is more detail than necessary for full credit.
(b) Prove that \( \text{swap}(\text{swap}(w)) = w \) for every string \( w \).

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

- If \( w = \varepsilon \), then
  \[
  \text{swap}(\text{swap}(w)) = \text{swap}(\text{swap}(\varepsilon)) = \text{swap}(\varepsilon) = \varepsilon = w.
  \]
  by definition of \( \text{swap} \)

- If \( w = \emptyset \) or \( w = 1 \), then
  \[
  \text{swap}(\text{swap}(w)) = \text{swap}(w) = w.
  \]
  by definition of \( \text{swap} \)

- Finally, if \( w = abx \) for some \( a, b \in \{0, 1\} \) and \( x \in \{0, 1\}^* \), then
  \[
  \text{swap}(\text{swap}(w)) = \text{swap}(\text{swap}(abx)) = \text{swap}(ba \cdot \text{swap}(x)) = \text{swap}(ba \cdot z) = ab \cdot \text{swap}(z) = ab \cdot \text{swap}(\text{swap}(x)) = ab \cdot x = w.
  \]
  by the induction hypothesis

In all cases, we conclude that \( \text{swap}(\text{swap}(w)) = w \). ■

**Rubric:** 5 points: Standard induction rubric (scaled). This is more detail than necessary for full credit.
5. The **reversal** $w^R$ of a string $w$ is defined recursively as follows:

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

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 $\Sigma$.

**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 = $\frac{1}{2}$ 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: $(x \cdot y)^R = y^R \cdot x^R$
- Right cancellation: If $x \cdot z = y \cdot z$, 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 $x \in \Sigma^*$, then

$$w^R = (a \cdot x \cdot a)^R$$

$$= (x \cdot a)^R \cdot a \quad \text{by definition of reversal}$$

$$= a^R \cdot x^R \cdot a \quad \text{by concatenation reversal}$$

$$= a \cdot x^R \cdot a \quad \text{by definition of reversal}$$

$$= a \cdot x \cdot a \quad \text{by the inductive hypothesis}$$

$$= w \quad \text{by assumption}$$

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 \( \varepsilon \):

- Reversal reversal: \( (x^R)^R = x \)
- Concatenation reversal: \( (x \cdot y)^R = y^R \cdot x^R \)
- Right cancellation: If \( x \cdot z = y \cdot z \), 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 \( w^R = ay a \).
  The recursive definition of string equality implies \( y^R a = ya \).
  Right cancellation implies that \( y^R = y \).
  The inductive hypothesis now implies that \( y \) is a palindrome.

  We conclude that \( w \) is a palindrome by definition.

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

**Rubric:** 4 points: standard induction rubric (scaled).
1. Let $L$ be the set of all strings $w$ in $\{A, B\}^*$ for which $\#(ABBA, w) \geq 2$. Here $\#(x, w)$ denotes the number of occurrences of the substring $x$ in the string $w$.

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

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

      You may either draw the DFA or describe it formally, but the states $Q$, the start state $s$, the accepting states $A$, and the transition function $\delta$ must be clearly specified. (See the standard DFA rubric for more details.)

      Argue that your DFA is correct by explaining what each state in your DFA means. Drawings or formal descriptions without English explanations will be heavily penalized, even if they are perfectly correct.

      [Hint: The shortest string in $L$ has length 7.]
2. Let $L$ denote the set of all strings $w \in \{0,1\}^*$ that satisfy at most two of the following conditions:

- The number of times the substring $01$ appears in $w$ is not divisible by $3^1$.
- The length of $w$ is even.
- The binary value of $w$ equals $2 \pmod{3}$.

For example: The string $0101$ satisfies all three conditions, so $0101$ is not in $L$, and the empty string $\epsilon$ satisfies only the second condition, so $\epsilon \in L$. ($01$ appears in $\epsilon$ zero times, and the binary value of $\epsilon$ is 0, because what else could it be?)

Formally describe a DFA with input alphabet $\Sigma = \{0,1\}$ that accepts the language $L$, by explicitly describing the states $Q$, the start state $s$, the accepting states $A$, and the transition function $\delta$. Do not attempt to draw your DFA; the smallest DFA for this language has 36 states, which is far too many for a drawing to be understandable.

Argue that your machine is correct by explaining what each state in your DFA means. Formal descriptions without English explanations will be heavily penalized, even if they are perfectly correct. (See the standard DFA rubric for more details.)

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

---

1Recall that $a$ is divisible by $b$ if and only if $a \equiv 0 \pmod{b}$. 
Standard regular expression rubric. For problems worth 10 points:

- 2 points for a syntactically correct regular expression.
- **Homework only:** 4 points for a brief English explanation of your regular expression. This is how you argue that your regular expression is correct.
  - For longer expressions, you should explain each of the major components of your expression, and separately explain how those components fit together.
  - We do not want a transcription; don't just translate the regular-expression notation into English.
- 4 points for correctness. (8 points on exams, with all penalties doubled)
  - $-1$ for a single mistake: one typo, excluding exactly one string in the target language, or including exactly one string not in the target language.
  - $-2$ for incorrectly including/excluding more than one but a finite number of strings.
  - $-4$ for incorrectly including/excluding an infinite number of strings.
- Regular expressions that are more complex than necessary may be penalized. Regular expressions that are significantly too complex may get no credit at all. On the other hand, minimal regular expressions are not required for full credit.
Standard DFA design rubric. For problems worth 10 points:

- 2 points for an unambiguous description of a DFA, including the states set $Q$, the start state $s$, the accepting states $A$, and the transition function $\delta$.
  
  - **Drawings:**
    - Use an arrow from nowhere to indicate $s$.
    - Use doubled circles to indicate accepting states $A$.
    - If $A = \emptyset$, you must say so explicitly.
    - If your drawing omits a junk/trash/reject state, you must say so explicitly.
    - **Draw neatly!** If we can’t read your solution, we can’t give you credit for it.
  
  - **Text descriptions:** You can describe the transition function either using a 2d array, using mathematical notation, or using an algorithm. But you must still give an explicit description of the states $Q$, the start state $s$, and the accepting states $A$.
  
  - **Product constructions:** You must give a complete description of each the DFAs you are combining (as either drawings, text, or recursive products), together with the accepting states of the product DFA.

- **Homework only:** 4 points for briefly explaining the purpose of each state in English. This is how you argue that your DFA is correct.
  
  - In particular, each state must have a mnemonic name.
  
  - For product constructions, explaining the states in the factor DFAs is both necessary and sufficient.
  
  - Yes, we mean it: A perfectly correct drawing of a perfectly correct DFA with no state explanation is worth at most 6 points.

- 4 points for correctness. (8 points on exams, with all penalties doubled)
  
  - $-1$ for a single mistake: a single misdirected transition, a single missing or extra accepting state, rejecting exactly one string that should be accepted, or accepting exactly one string that should be accepted.
  
  - $-2$ for incorrectly accepting/rejecting more than one but a finite number of strings.
  
  - $-4$ for incorrectly accepting/rejecting an infinite number of strings.

- DFAs that are more complex than necessary may be penalized. DFAs that are significantly more complex than necessary may get no credit at all. On the other hand, minimal DFAs are not required for full credit, unless the problem explicitly asks for them.

- Half credit for describing an NFA when the problem asks for a DFA.
Solved problem

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

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

Following the C99 standard, we explicitly disallow nesting comments of the same type. A line comment starts with $//$ and ends at the first $\downarrow$ after the opening $//$. A block comment starts with $/*$ and ends at the 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:

- $/*\star\star\star*/$  
- $//\diamond//\diamond$  
- $/*//\diamond\diamond\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\star\sta...
(a) Describe a regular expression for the set of all C comments.

**Solution:**

```
//((/ + * + A + *)*[^] + /*((/ + A + ^ + [^]*A + [^]))*[^]*/)*/
```

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 (A + ^ + [^]) 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 ( ), newlines ( ^ ), and C comments.

**Solution:**

```
( ) + //((/ + * + A + *)*[^] + /*((/ + A + ^ + [^]*A + [^]))*[^]*/)*/
```

This regular expression has the form ((whitespace) + (comment))*, where (whitespace) is the regular expression  + and (comment) 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.
1. Prove that the following languages are not regular.

(a) \( \{ \theta^m \lambda^n \mid m \text{ and } n \text{ are relatively prime} \} \)
(b) \( \{ w \in (\theta + 1)^* \mid 10^n \lambda^n \text{ for } n > 0 \text{ is a suffix of } w \} \)
(c) The set of all palindromes in \((\theta + 1)^*\) whose length is divisible by 3.

2. For each of the following languages over the alphabet \( \Sigma = \{\theta, 1\} \), 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 \( \Sigma^+ \) denotes the set of all nonempty strings over \( \Sigma \). Watch those parentheses!

(a) \( \{ \theta^a \lambda \lambda^c \mid w \in \Sigma^+, (a \leq |w| + c) \text{ and } (|w| \leq a + c \text{ or } c \leq a + |w|) \} \)
(b) \( \{ \theta^a \lambda \lambda^c \mid w \in \Sigma^+, a > 0, |w| \geq 0 \} \)
(c) \( \{ x \lambda \lambda^y \mid w, x, y \in \Sigma^+ \} \)
(d) \( \{ \lambda \lambda^x y \mid w, x, y \in \Sigma^+ \} \)

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

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

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

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

**Solution:** Regular: \( \epsilon + 01^* + 10^* \). Call this language \( L_a \).

Let \( w \) be an arbitrary non-empty string in \((\emptyset + 1)^*\). 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 \( n \leq |x| \). Every palindrome that starts with \( 0 \) also ends with \( 0 \), so the only palindrome prefixes of \( w \) are \( \epsilon \) 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, \( \epsilon \in L_a \). □

**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 \((\emptyset + 1 + 2)^*\) in which no prefix of length at least 2 is a palindrome.

**Solution:** Not regular. Call this language \( L_b \).

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

Let \( x \) and \( y \) be arbitrary distinct strings in \( F \).

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

Without loss of generality, assume \( i < j \).

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

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

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

Because \( F \) is infinite, \( L_b \) cannot be regular. □

**Rubric:** 2½ points = ½ for “not regular” + 2 for fooling set proof (standard rubric, scaled).
(c) Strings in \((0 + 1)^*\) in which no prefix of length at least 3 is a palindrome.

**Solution: Not regular.** Call this language \(L_c\).

I claim that the infinite language \(F = (001101)^*\) is a fooling set for \(L_c\).

Let \(x\) and \(y\) be arbitrary distinct strings in \(F\).

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

Without loss of generality, assume \(i < j\).

Let \(z\) be the suffix \((101100)^j\).

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

We conclude that \(F\) is a fooling set for \(L_c\), as claimed.

Because \(F\) is infinite, \(L_c\) cannot be regular. ■

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

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

**Solution: Regular.** Call this language \(L_d\).

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

\[
(0 + 1)^*(000 + 010 + 101 + 111 + 0110 + 1001)(0 + 1)^*
\]

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

**Solution: Regular.** Call this language \(L_d\).

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

\[
\epsilon + 0 + 1 + 00 + 01 + 10 + 11 + 001 + 011 + 100 + 110 + 0011 + 1100
\]

**Rubric:** 2½ points = ½ for “regular” + 2 for proof:
- 1 for expression for \(\overline{L_d}\) + 1 for applying closure
- 1 for regular expression + 1 for justification
Standard fooling set rubric. For problems worth 5 points:

• 2 points for the fooling set:
  + 1 for explicitly describing the proposed fooling set $F$.
  + 1 if the proposed set $F$ is actually a fooling set for the target language.
    — No credit for the proof if the proposed set is not a fooling set.
    — No credit for the problem if the proposed set is finite.

• 3 points for the proof:
  ◦ The proof must correctly consider arbitrary strings $x, y \in F$.
    — No credit for the proof unless both $x$ and $y$ are always in $F$.
    — No credit for the proof unless $x$ and $y$ can be any strings in $F$.
  + 1 for correctly describing a suffix $z$ that distinguishes $x$ and $y$.
  + 1 for proving either $xz \in L$ or $yz \in L$.
  + 1 for proving either $yz \notin L$ or $xz \notin L$, respectively.

As usual, scale partial credit (rounded to nearest ½) for problems worth fewer points.
This is the last homework before Midterm 1.

1. For each of the following regular expressions, describe or draw two finite-state machines:
   
   • An NFA that accepts the same language, constructed from the given regular expression using Thompson’s algorithm (described in class and in the notes).
   
   • An equivalent DFA, constructed from your NFA using the incremental subset algorithm (described in class and in the notes). For each state in your DFA, identify the corresponding subset of states in your NFA. Your DFA should have no unreachable states.

   (a) \((0 + 11)^*(00 + 1)^*\)
   
   (b) \(((0^* + 1)^* + 0)^* + 1)^*\)

   (see next page for Question 2)
2. Let $L$ be any regular language over the alphabet $\Sigma = \{0, 1\}$. Prove that the following languages are also regular:

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

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

**Standard language transformation rubric.** For problems worth 10 points:

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

Solved problem

3. (a) Fix an arbitrary regular language $L$. Prove that the language $\text{half}(L) := \{w \mid ww \in L\}$ 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 $\varepsilon$-transitions that accepts $\text{half}(L)$, as follows:

$$Q' = (Q \times Q \times Q) \cup \{s'\}$$
$s'$ is an explicit state in $Q'$

$$A' = \{(h, h, q) \mid h \in Q \text{ and } q \in A\}$$

$$\delta'(s', \varepsilon) = \{(s, h, h) \mid h \in Q\}$$

$$\delta'(s', a) = \emptyset$$

$$\delta'((p, h, q), \varepsilon) = \emptyset$$

$$\delta'((p, h, q), a) = \{(\delta(p, a), h, \delta(q, a))\}$$

$M'$ 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 language transformation rubric (scaled). Yes, the smartass solution would be worth full credit.
(b) Describe a regular language $L$ such that the language $\text{double}(L) := \{ww \mid w \in L\}$ is not regular. Prove your answer is correct.

**Solution:** Consider the regular language $L = \emptyset^*1$.
Expanding the regular expression lets us rewrite $L = \{\emptyset^n1 \mid n \geq 0\}$. It follows that $\text{double}(L) = \{\emptyset^n1\emptyset^n1 \mid n \geq 0\}$. I claim that this language is not regular.
Let $x$ and $y$ be arbitrary distinct strings in $L$.
Then $x = \emptyset^i1$ and $y = \emptyset^j1$ for some integers $i \neq j$.
Then $x$ is a distinguishing suffix of these two strings, because
- $xx \in \text{double}(L)$ by definition, but
- $yx = \emptyset^i1\emptyset^j1 \notin \text{double}(L)$ because $i \neq j$.
We conclude that $L$ is a fooling set for $\text{double}(L)$.
Because $L$ is infinite, $\text{double}(L)$ cannot be regular.

**Solution:** Consider the regular language $L = \Sigma^* = (\emptyset + 1)^*$.
I claim that the language $\text{double}(\Sigma^*) = \{ww \mid w \in \Sigma^*\}$ 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 $xz \in \text{double}(\Sigma^*)$, but
- $yx = 01^i01^j \notin \text{double}(\Sigma^*)$ because $i \neq j$.
We conclude that $F$ is a fooling set for $\text{double}(\Sigma^*)$.
Because $F$ is infinite, $\text{double}(\Sigma^*)$ cannot be regular.

**Rubric:** 5 points:
- 2 points for describing a regular language $L$ such that $\text{double}(L)$ is not regular.
- 1 point for describing an infinite fooling set for $\text{double}(L)$:
  - $\frac{1}{2}$ for explicitly describing the proposed fooling set $F$.
  - $\frac{1}{2}$ if the proposed set $F$ is actually a fooling set.
    - No credit for the proof if the proposed set is not a fooling set.
    - No credit for the problem if the proposed set is finite.
- 2 points for the proof:
  - $\frac{1}{2}$ for correctly considering arbitrary strings $x$ and $y$.
    - No credit for the proof unless both $x$ and $y$ are always in $F$.
    - No credit for the proof unless both $x$ and $y$ can be any string in $F$.
  - $\frac{1}{2}$ for correctly stating a suffix $z$ that distinguishes $x$ and $y$.
  - $\frac{1}{2}$ for proving either $xz \in L$ or $yz \in L$.
  - $\frac{1}{2}$ for proving either $yz \notin L$ or $xz \notin L$, respectively.
These are not the only correct solutions. These are not the only fooling sets for these languages.
Standard language transformation rubric. For problems worth 10 points:

+ 2 for a formal, complete, and unambiguous description of the output automaton, including the states, the start state, the accepting states, and the transition function, as functions of an arbitrary input DFA. The description must state whether the output automaton is a DFA, an NFA without $\epsilon$-transitions, or an NFA with $\epsilon$-transitions.
  • No points for the rest of the problem if this is missing.

+ 2 for a brief English explanation of the output automaton. We explicitly do not want a formal proof of correctness, or an English transcription, but a few sentences explaining how your machine works and justifying its correctness. What is the overall idea? What do the states represent? What is the transition function doing? Why these accepting states?
  • Deadly Sin: No points for the rest of the problem if this is missing.

+ 6 for correctness
  • 3 for accepting all strings in the target language
  • 3 for accepting only strings in the target language
  • 1 for a single mistake in the formal description (for example a typo)
  • Double-check correctness when the input language is $\emptyset$, or $\{\epsilon\}$, or $\theta^*$, or $\Sigma^*$.
1. Consider the following cruel and unusual sorting algorithm, proposed by Gary Miller:

```
Cruel(A[1..n]):
  if n > 1
    Cruel(A[1..n/2])
    Cruel(A[n/2 + 1..n])
    Unusual(A[1..n])
```

```
Unusual(A[1..n]):
  if n = 2
    else
      for i ← 1 to n/4 (swap 2nd and 3rd quarters)
        swap A[i + n/4] ↔ A[i + n/2]
      Unusual(A[1..n/2]) (recurse on left half)
      Unusual(A[n/2 + 1..n]) (recurse on right half)
      Unusual(A[n/4 + 1..3n/4]) (recurse on middle half)
```

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

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

(b) Prove that Cruel would not correctly sort if we removed the for-loop from Unusual.

(c) Prove that Cruel would not correctly sort if we swapped the last two lines of Unusual.

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

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

2. Dakshita is putting together a list of famous cryptographers, each with their dates of birth and death: al-Kindi (801–873), Giovanni Fontana (1395–1455), Leon Alberti (1404–1472), Charles Babbage (1791–1871), Alan Turing (1912–1954), Joan Clarke (1917–1996), Ann Caracristi (1921–2016), and so on. She wonders which two cryptographers on her list had the longest overlap between their lifetimes. For example, among the seven example cryptographers, Clarke and Caracristi had the longest overlap of 45 years (1921–1966).
Dakshita formalizes her problem as follows. The input is an array $A[1..n]$ of records, each with two numerical fields $A[i].birth$ and $A[i].death$ and a string field $A[i].name$. The desired output is the maximum, over all indices $i \neq j$, of the overlap length


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

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

Solved Problems

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

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 \( Q[1..[n/2]] \) blue.
- Color the elements in the Right half \( Q[[n/2]+1..n] \) red.
- Recursively count inversions in (and sort) the blue subarray \( Q[1..[n/2]] \).
- Recursively count inversions in (and sort) the red subarray \( Q[[n/2]+1..n] \).
- 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:
Merge and \texttt{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 \texttt{Merge} algorithm, without any need for “colors”. Here changes to the standard \texttt{Merge} algorithm are indicated in red.

\begin{verbatim}
\texttt{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
\end{verbatim}

We can further optimize \texttt{MergeAndCount} by observing that \texttt{count} is always equal to $j - m - 1$, so we don’t need an additional variable. (Proof: Initially, $j = m + 1$ and \texttt{count} = 0, and we always increment $j$ and \texttt{count} together.)
MergeAndCount2(A[1 .. n], m):
  \( i \leftarrow 1; j \leftarrow m + 1; \ total \leftarrow 0 \)
  for \( k \leftarrow 1 \) to \( n \)
    if \( j > n \)
      \( B[k] \leftarrow A[i]; \ i \leftarrow i + 1; \ total \leftarrow total + j - m - 1 \)
    else if \( i > m \)
      \( B[k] \leftarrow A[j]; \ j \leftarrow j + 1 \)
      \( B[k] \leftarrow A[i]; \ i \leftarrow i + 1; \ total \leftarrow total + j - m - 1 \)
    else
      \( B[k] \leftarrow A[j]; \ j \leftarrow j + 1 \)
  for \( k \leftarrow 1 \) to \( n \)
    \( A[k] \leftarrow B[k] \)
return \( total \)

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

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

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

Notice that each boxed algorithm is preceded by a clear English description of the task that algorithm performs—not how the algorithm works, but the relationship between its input and its output. Each English description is worth 25% of the credit for that algorithm (rounding to the nearest point). For example, the COUNTREDBLUE algorithm is worth 4 points (“conquer”); the English description alone (“For each blue element \( Q[i] \) of the now-sorted array \( Q[1 .. n] \), count the number of smaller red elements \( Q[j] \).”) is worth 1 point.
1. **Vankin’s Mile** is an American solitaire game played on an $n \times n$ square grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his score. The object of the game is to score as many points as possible.

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

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

(b) A variant called **Vankin’s Niknav** adds an additional constraint to Vankin’s Mile: *The sequence of values that the token touches must be a palindrome*. Thus, the example path on the right is valid, but the example path on the left is not. Describe and analyze an efficient algorithm to compute the maximum possible score for an instance of Vankin’s Niknav, given the $n \times n$ array of values as input.
2. A **snowball** is a poem or sentence that starts with a one-letter word, where each later word is one letter longer than its predecessor. For example:

   I am the fire demon, moving castles: Calcifer!

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

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

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

For example, given the input string

```
EVENIFYOUAMTHEAREMYFIRELEASTDEMONFAVORITEMOVINGCASTLESVEGETABLECALCIFER
```

your algorithm should return the integer 8:

```
EVENIFYOUAMTHEAREMYFIRELEASTDEMONFAVORITEMOVINGCASTLESVEGETABLECALCIFER
```
Standard dynamic programming rubric. For problems worth 10 points:

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

- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  + 1 for base case(s). —½ for one minor bug, like a typo or an off-by-one error.
  + 3 for recursive case(s). —1 for each minor bug, like a typo or an off-by-one error.
  - 2 for greedy optimizations without proof, even if they are correct.
  - No credit for the rest of the problem if the recursive case(s) are incorrect.

- 3 points for iterative details
  + 1 for describing an appropriate memoization data structure
  + 1 for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested for loops, be sure to specify the nesting order.
  + 1 for correct time analysis. (It is not necessary to state a space bound.)

- For problems that ask for an algorithm that computes an optimal structure—such as a subset, partition, subsequence, or tree—an algorithm that computes only the value or cost of the optimal structure is sufficient for full credit, unless the problem specifically says otherwise.

- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, but iterative pseudocode is not required for full credit. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. But you do still need and English description of the underlying recursive function (or equivalently, the contents of the memoization structure). **Perfectly correct iterative pseudocode, with no explanation or time analysis, is worth at most 6 points out of 10.**

- Official solutions will provide target time bounds. Faster algorithms are worth more points, and slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of \( n \) in either direction. Partial credit is scaled to the new maximum score, and all points above 10 (for algorithms that are faster than our target time bound) are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we do include them, significantly more students submit incorrect algorithms with the target running time (earning 0/10) instead of correct algorithms that are slower than the target (earning 7/10).

- Partial credit for incomplete solutions depends on the running time of the best possible completion (up to the target running time). For example, consider a solution that contains only a clear English description of a function, with no recurrence or iterative details. If the described function can be developed into an algorithm with the target running time, the solution is worth 3 points; however, if the function leads to an algorithm that is slower than the target time by a factor of \( n \), the solution could be worth only 2 points (≈ 70% of 3, rounded).
Solved Problem

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

Similarly, the strings PRODGYRNAMMIIINC and DYPRONGARMMICING are both shuffles of the strings DYNAMIC and PROGRAMMING:

(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 $\text{Shuf}(i, j)$, 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 $\text{Shuf}(m, n)$. The function $\text{Shuf}$ satisfies the following recurrence:

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

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

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

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

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

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

The algorithm runs in $O(mn)$ time. □

**Rubric:** 5 points, standard dynamic programming rubric. 3 points for a slower polynomial-time algorithm; scale partial credit accordingly.
1. Every year, as part of its annual meeting, the Antarctican Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to \( n \). During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.


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

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array \( M \) as input.
2. Suppose you are given a NFA $M = (\{0, 1\}, Q, s, A, \delta)$ without $\epsilon$-transitions and a binary string $w \in \{0, 1\}^*$. Describe and analyze an efficient algorithm to determine whether $M$ accepts $w$. Concretely, the input NFA $M$ is represented as follows:

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

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

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

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

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

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

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

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

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


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

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

**Rubric:** 10 points, standard dynamic programming rubric
4. Oh, no! You've just been appointed as the new organizer of Giggle, Inc.'s annual mandatory holiday party! The employees at Giggle are organized into a strict hierarchy, that is, a tree with the company president at the root. The all-knowing oracles in Human Resources have assigned a real number to each employee measuring how “fun” the employee is. In order to keep things social, there is one restriction on the guest list: An employee cannot attend the party if their immediate supervisor is also present. On the other hand, the president of the company must attend the party, even though she has a negative fun rating; it’s her company, after all.

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

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

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

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

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

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

```python
def ComputeMaxFun(v):
    v.\text{yes} \leftarrow v.\text{fun}
    v.\text{no} \leftarrow 0
    for all children $w$ of $v$
        ComputeMaxFun($w$)
        $v.\text{yes} \leftarrow v.\text{yes} + w.\text{no}$
        $v.\text{no} \leftarrow v.\text{no} + \max\{w.\text{yes}, w.\text{no}\}$

\text{BESTParty}(T):
\quad \text{ComputeMaxFun}(T.\text{root})
\quad \text{return } T.\text{root}.\text{yes}
```

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

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

\[
\text{root}\_\text{fun} + \sum_{\text{grandchildren} \ w \ of \ \text{root}} \text{MaxFun}(w).
\]

The function \( \text{MaxFun} \) obeys the following recurrence:

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

(This recurrence does not require a separate base case, because \( \sum \emptyset = 0 \).) We can memoize this function by adding an additional field \( v\_\text{maxFun} \) 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 \( T \).

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

---

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

---

**Rubric:** 10 points: standard dynamic programming rubric. These are not the only correct solutions.
1. **Racetrack** (also known by several other names, including *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.\(^1\) The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

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

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

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

---

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

\(^2\)However, it is not necessary for the line between the old position and the new position to lie entirely within the track. Sometimes Speed Racer has to push the A button.
2. Jeff likes to go on a long bike ride every Sunday, but because he is lazy, he absolutely refuses to ever ride into the wind.

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

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

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

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

For example, suppose Jeff has the graph \( G \) shown below. On the first Sunday, Jeff can ride from \( s \) to \( w \) along the path shown on the left, including the red edge from \( v \) to \( w \). On the second Sunday, Jeff can ride from \( s \) to \( v \) along the green path on the left in the morning, and then from \( v \) back to \( s \) along the green path on the right in the afternoon; however, he cannot ride to \( w \), because every path from \( w \) to \( s \) requires riding into the wind at least once, and Jeff’s wife is tired of driving out to the middle of nowhere to rescue him.
3. This problem is intended as a practice run for future homeworks, the second midterm, the final exam. Each student must submit individually.

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

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

The assignment will ask you to write/draw something, scan the paper, convert your scan to a PDF file, and upload the PDF to Gradescope. (Gradescope will automatically assign pages of your uploaded PDF to corresponding subproblems.) Alternatively, you can write/draw on a tablet and a note-taking app, export your note as a PDF file, upload the PDF to Gradescope.

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

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

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

**Standard rubric for graph reduction problems.** For problems out of 10 points:
+ 1 for correct vertices, including English explanation for each vertex
+ 1 for correct edges
  − ½ for forgetting “directed” if the graph is directed
+ 1 for stating the correct problem (For the solved problem below: “shortest path in $G$ from $(0, 0, 0)$ to any target vertex”)
  − “Breadth-first search” is not a problem; it’s an algorithm!
+ 1 for correctly applying the correct algorithm. (For the solved problem below, “breadth-first search from $(0, 0, 0)$ and then examine every target vertex”)
  − ½ for using a slower or more specific algorithm than necessary
+ 1 for time analysis in terms of the input parameters.
+ 5 for other details of the reduction
  − If your graph is constructed by naive brute force, you do not need to describe the construction algorithm. In this case, points for vertices, edges, problem, algorithm, and running time are all doubled.
  − Otherwise, apply the appropriate rubric to the construction algorithm. For example, for an algorithm that uses dynamic programming to build the graph quickly, apply the standard dynamic programming rubric.

Solved Problem

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

(a) Fill a jar with water from the lake until the jar is full.
(b) Empty a jar of water by pouring water into the lake.
(c) Pour water from one jar to another, until either the first jar is empty or the second jar is full, whichever happens first.

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

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

Describe and analyze an efficient algorithm that either finds the smallest number of operations that leave exactly $k$ gallons in any jar, or reports correctly that obtaining exactly $k$ gallons of water is impossible. Your input consists of the capacities of the three jars and the positive integer $k$. For example, given the four numbers 6, 10, 15, and 13 as input, your algorithm should return the number 6 (the length of the sequence of operations listed above).
Solution: Let A, B, C denote the capacities of the three jars. We reduce the problem to breadth-first search in a directed graph \( G = (V, E) \) defined as follows:

- \( V = \{(a, b, c) \mid 0 \leq a \leq A \text{ and } 0 \leq b \leq B \text{ and } 0 \leq c \leq C\} \). 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 \((a, b, c)\)\(\rightarrow\)(\(a', b', c'\)) whenever it is possible to move from the first configuration to the second in one step. Specifically, \( G \) contains an edge from \((a, b, c)\) to each of the following vertices (except those already equal to \((a, b, c)\)):

  - \((0, b, c)\) and \((a, 0, c)\) and \((a, b, 0)\) — dumping a jar into the lake
  - \((A, b, c)\) and \((a, B, c)\) and \((a, b, C)\) — filling a jar from the lake
  - \(\{(0, a + b, c) \mid a + b \leq B\}\)
    - pouring from jar 1 into jar 2
  - \(\{(a + b - B, b, c) \mid a + b \geq B\}\)
    - pouring from jar 1 into jar 3
  - \(\{(0, 0, a + c) \mid a + c \leq C\}\)
    - pouring from jar 2 into jar 1
  - \(\{(a + b, 0, c) \mid a + b \leq A\}\)
    - pouring from jar 2 into jar 3
  - \(\{(0, a + b - A, c) \mid a + b \geq A\}\)
    - pouring from jar 2 into jar 3
  - \(\{(a + c, 0) \mid a + c \leq A\}\)
    - pouring from jar 3 into Jar 1
  - \(\{(0, b, a + c) \mid a + c \geq A\}\)
    - pouring from jar 3 into Jar 2
  - \(\{(a + c - C, b, C) \mid a + c \geq A\}\)
    - 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 \textit{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 \textbf{breadth-first search} starting at \((0, 0, 0)\), and then examining every target vertex by brute force. If \textbf{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 \(O(V + E) = O(ABC)\) time.

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

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

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

Describe and analyze an algorithm to solve Morty’s problem. You can assume that it is in fact possible for Morty to succeed. As usual, do not assume that edge weights are integers.

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

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

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

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

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

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 $u \rightarrow v$ **tight** if $dist(u) + w(u \rightarrow v) = dist(v)$. 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:

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

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 $NumPaths(v) = \sum \emptyset = 0$.

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

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

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

*Hint: There exists a reduction to all three problems; you may pick whichever one is most convenient for you.*
2. This problem asks you to develop polynomial-time algorithms for two (apparently) minor variants of 3Sat.

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

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

(b) The input to MAJORITY3Sat is a boolean formula $\Phi$ in conjunctive normal form, with exactly three literals per clause. MAJORITY3Sat asks whether there is an assignment to the variables of $\Phi$ such that every clause contains at least two True literals.

Describe and analyze a polynomial-time reduction from MAJORITY3Sat to 2Sat. Don’t forget to prove that your reduction is correct.

(c) Combining parts (a) and (b) gives us an algorithm for MAJORITY3Sat. What is the running time of this algorithm?
Solved Problem

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

(1) Every row contains at least one stone.
(2) No column contains stones of both colors.

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

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

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

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

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

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

$\implies$ First, suppose $\Phi$ 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 $\text{FALSE}$ 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 $\Phi$ must contain at least one $\text{TRUE}$ literal, and thus each row still contains at least one stone. We conclude that the puzzle is satisfiable.
On the other hand, suppose the puzzle is solvable; consider an arbitrary solution. For each index $j$, assign a value to $x_j$ depending on the colors of stones left in column $j$:

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

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

This reduction clearly requires only polynomial time. ■

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

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

+ 3 points for the “if” proof of correctness
+ 3 points for the “only if” proof of correctness
+ 1 point for writing “polynomial time”

• An incorrect polynomial-time reduction that still satisfies half of the correctness proof is worth at most 4/10.
• A reduction in the wrong direction is worth 0/10.
1. (a) A **quasi-satisfying assignment** for a 3CNF boolean formula $\Phi$ is an assignment of truth values to the variables such that at most one clause in $\Phi$ does not contain a true literal.

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

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

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

2. A **wye** is an undirected graph that looks like the capital letter Y. More formally, a wye consists of three paths of equal length with one common endpoint, called the hub.

![Diagram of a wye](attachment:image.png)

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

Prove that the following problem is NP-hard: Given an undirected graph $G$, what is the largest wye that is a subgraph of $G$? The three paths of the wye must not share any vertices except the hub, and they must have exactly the same length.
Solved Problem

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

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

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 $v^\#$ and $v^\flat$, along with three edges $vv^\#, vv^\flat$, and $v^\flat v^\#$.

![A vertex in $G$, and the corresponding vertex gadget in $H$.](image)

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

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

$$\cdots \rightarrow v_i \rightarrow v_i^\# \rightarrow v_i^\flat \rightarrow v_i^\flat \rightarrow v_i^\flat \rightarrow v_i \rightarrow \cdots$$

$\impliedby$ 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 $v^\#$ or $v^\flat$ to any other vertex in $H$ must pass through $v$. Thus, one of the two closed walks visits only the vertices $v$, $v^\#$, and $v^\flat$. Thus, if we simply remove the vertices in $H \setminus G$ from $D$, we obtain a closed walk in $G$ that visits every vertex in $G$ once.

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

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

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

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

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

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

Unfortunately, if $H$ has a double-Hamiltonian tour, we cannot conclude that $G$ has a Hamiltonian cycle, because we cannot guarantee that a double-Hamiltonian tour in $H$ uses any self-loops. The graph $G$ shown below is a counterexample; it has a double-Hamiltonian tour (even before adding self-loops!) but no Hamiltonian cycle.
Some useful NP-hard problems. You are welcome to use any of these in your own NP-hardness proofs, except of course for the specific problem you are trying to prove NP-hard.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

IntegerLinearProgramming: Given a matrix $A \in \mathbb{Z}^{n \times d}$ and two vectors $b \in \mathbb{Z}^n$ and $c \in \mathbb{Z}^d$, 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?
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.

$$\text{SELFRevAccept} := \{ \langle M \rangle \mid M \text{ accepts the string } \langle M \rangle^R \}$$

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:

$$\text{SomeSquareSpace} = \{ \langle M \rangle \mid M \text{ accepts at least one string } w \text{ in space } |w|^2 \}$$

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:

$$\text{Picky} = \{ \langle M \rangle \mid M \text{ accepts at least one input string and } M \text{ rejects at least one input string} \}$$

Note that Rice’s theorem does not apply to this language.
For each statement below, check “Yes” if the statement is ALWAYS true and “No” otherwise, and give a brief explanation of your answer.

(a) Every integer in the empty set is prime.

(b) The language \( \{0^m 1^n \mid m + n \leq 374 \} \) is regular.

(c) The language \( \{0^m 1^n \mid m - n \leq 374 \} \) is regular.

(d) For all languages \( L \), the language \( L^* \) is regular.

(e) For all languages \( L \), the language \( L^* \) is infinite.

(f) For all languages \( L \subset \Sigma^* \), if \( L \) can be represented by a regular expression, then \( \Sigma^* \setminus L \) is recognized by a DFA.

(g) For all languages \( L \) and \( L' \), if \( L \cap L' = \emptyset \) and \( L' \) is not regular, then \( L \) is regular.

(h) Every regular language is recognized by a DFA with exactly one accepting state.

(i) Every regular language is recognized by an NFA with exactly one accepting state.

(j) Every language is either regular or context-free.
For each of the following languages over the alphabet $\Sigma = \{0, 1\}$, either prove that the language is regular or prove that the language is not regular. Exactly one of these two languages is regular. Both of these languages contain the string $00110100000110100$.

1. $\{0^n w 0^n \mid w \in \Sigma^+ \text{ and } n > 0\}$

2. $\{w 0^n w \mid w \in \Sigma^+ \text{ and } n > 0\}$
The parity of a bit-string $w$ is 0 if $w$ has an even number of 1s, and 1 if $w$ has an odd number of 1s. For example:

\[
\text{parity}(\epsilon) = 0 \quad \text{parity}(0010100) = 0 \quad \text{parity}(00101110100) = 1
\]

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

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

(c) Let $L$ be an arbitrary regular language. Prove that the language $\text{AddParity}(L) := \{\text{parity}(w) \cdot w \mid w \in L\}$ is also regular.

[Hint: Yes, you have enough room.]
For each of the following languages $L$, give a regular expression that represents $L$ and describe a DFA that recognizes $L$. You do not need to prove that your answers are correct.

(a) All strings in $(0 + 1)^*$ that do not contain the substring $0110$.

(b) All strings in $0^*10^*$ whose length is a multiple of 3.
For any string $w \in \{0, 1\}^*$, let $\text{obliviate}(w)$ denote the string obtained from $w$ by removing every 1. For example:

$$
\begin{align*}
\text{obliviate}(\epsilon) &= \epsilon \\
\text{obliviate}(000000) &= 000000 \\
\text{obliviate}(111111) &= \epsilon \\
\text{obliviate}(01001101) &= 00000
\end{align*}
$$

Let $L$ be an arbitrary regular language.

1. **Prove** that the language $\text{OBLIVIATE}(L) = \{\text{obliviate}(w) \mid w \in L\}$ is regular.

2. **Prove** that the language $\text{UNOBLIVIATE}(L) = \{w \in \{0, 1\}^* \mid \text{obliviate}(w) \in L\}$ is regular.
Directions

• Don’t panic!

• If you brought anything except your writing implements, your hand-written double-sided 8½" × 11" cheat sheet, please put it away for the duration of the exam. In particular, please turn off and put away all medically unnecessary electronic devices.

• The exam has five numbered questions.

• Write your answers on blank white paper. Please start your solution to each numbered question on a new sheet of paper.

• You have 150 minutes to write, scan, and submit your solutions. The exam is designed to take at most 120 minutes to complete. We are providing 30 minutes of slack to scan and submit in case of unforeseen technology issues.

• If you are ready to scan your solutions before 9:15pm, send a private message to the host (“Ready to scan”) and wait for confirmation before leaving the Zoom call.

• Please scan all paper that you used during the exam — first your solutions, in the correct order, then your cheat sheet (if any), and finally any scratch paper.

• Proofs are required for full credit if and only if we explicitly ask for them, using the word prove in bold italics. In particular, if we ask you to show that a language is regular, you can provide a regular expression, DFA, NFA, or boolean combination without justification. Similarly, if we ask you to give a DFA or NFA, you to not have to name or describe the states.

• Finally, if something goes seriously wrong, send email to jeffe@illinois.edu as soon as possible explaining the situation. If you have already finished the exam but cannot submit to Gradescope for some reason, include a complete scan of your exam in your email. If you are in the middle of the exam, send Jeff email, finish the exam (if you can) within the time limit, and then send a second email with your completed exam.
1. 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 and proving that the set you construct is indeed a fooling set for that language).

(a) \( \{0^p1^q0^r \mid r = p + q\} \)
(b) \( \{0^p1^q0^r \mid r = p + q \mod 2\} \)

[Hint: First think about the language \( \{0^p1^q \mid q = p \mod 2\} \)]

2. Let \( L \) be any regular language over the alphabet \( \Sigma = \{0, 1\} \).

Let \( \text{take2skip2}(w) \) be a function that takes an input string \( w \) and returns the subsequence of symbols at positions \( 1, 2, 5, 6, 9, 10, \ldots 4i + 1, 4i + 2, \ldots \) in \( w \). In other words, \( \text{take2skip2}(w) \) takes the first two symbols of \( w \), skips the next two, takes the next two, skips the next two, and so on. For example:

\[
\begin{align*}
\text{take2skip2}(1) &= 1 \\
\text{take2skip2}(01) &= 01 \\
\text{take2skip2}(01001110011) &= 011001
\end{align*}
\]

Choose exactly one of the following languages, and prove that your chosen language is regular. (In fact, both languages are regular, but we only want a proof for one of them.) Don’t forget to tell us which language you’ve chosen!

(a) \( L_1 = \{w \in \Sigma^* \mid \text{take2skip2}(w) \in L\} \).
(b) \( L_2 = \{\text{take2skip2}(w) \mid w \in L\} \).

3. Prove that the following languages are not regular by building an infinite fooling set for each of them. For each language, prove that the set you constructed is indeed a fooling set.

(a) \( \{0^p1^q0^r \mid r > 0 \quad \text{and} \quad q \mod r = 0 \quad \text{and} \quad p \mod r = 0\} \)
(b) \( \{0^p1^q \mid q > 0 \quad \text{and} \quad p = q^q\} \)

4. Consider the following recursive function:

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

For example, \( \text{MINGLE}(01, 10) = \text{MINGLE}(1, 0100) = \text{MINGLE}(\varepsilon, 101001) = 101001 \).

(a) Prove that \( |\text{MINGLE}(w, z)| = 2|w| + |z| \) for all strings \( w \) and \( z \).
(b) Prove that \( \text{MINGLE}(w, z \cdot z^R) = (\text{MINGLE}(w, z \cdot z^R))^R \) for all strings \( w \) and \( z \).

(There’s one more question on the next page)
5. For each statement below, write “Yes” if the statement is always true and write “No” otherwise, and give a brief (one short sentence) explanation of your answer. Read these statements very carefully—small details matter!

(a) If \( L \) is a regular language over the alphabet \( \{0, 1\} \), then \( \{w1w \mid w \in L\} \) is also regular.

(b) If \( L \) is a regular language over the alphabet \( \{0, 1\} \), then \( \{x1y \mid x, y \in L\} \) is also regular.

(c) The context-free grammar \( S \rightarrow 0S1 \mid 1S0 \mid SS \mid 01 \mid 10 \) generates the language \( (0 + 1)^+ \).

(d) Every regular expression that does not contain a Kleene star (or Kleene plus) represents a finite language.

(e) Let \( L_1 \) be a finite language and \( L_2 \) be an arbitrary language. Then \( L_1 \cap L_2 \) is regular.

(f) Let \( L_1 \) be a finite language and \( L_2 \) be an arbitrary language. Then \( L_1 \cup L_2 \) is regular.

(g) The regular expression \( (00 + 01 + 10 + 11)^* \) represents the language of all strings over \( \{0, 1\} \) of even length.

(h) The \( \epsilon \)-reach of any state in an NFA contains the state itself.

(i) The language \( L = \emptyset^* \) over the alphabet \( \Sigma = \{0, 1\} \) has a fooling set of size 2.

(j) Suppose we define an \( \epsilon \)-DFA to be a DFA that can additionally make \( \epsilon \)-transitions. Any language that can be recognized by an \( \epsilon \)-DFA can also be recognized by a DFA that does not make any \( \epsilon \)-transitions.
Directions

• Don’t panic!

• If you brought anything except your writing implements, your hand-written double-sided 8½" × 11" cheat sheet, please put it away for the duration of the exam. In particular, please turn off and put away all medically unnecessary electronic devices.

• The exam has five numbered questions.

• Write your answers on blank white paper. Please start your solution to each numbered question on a new sheet of paper.

• You have 150 minutes to write, scan, and submit your solutions. The exam is designed to take at most 120 minutes to complete. We are providing 30 minutes of slack to scan and submit in case of unforeseen technology issues.

• If you are ready to scan your solutions before 9:15pm, send a private message to the host (“Ready to scan”) and wait for confirmation before leaving the Zoom call.

• Please scan all paper that you used during the exam — first your solutions, in the correct order, then your cheat sheet (if any), and finally any scratch paper.

• Proofs are required for full credit if and only if we explicitly ask for them, using the word prove in bold italics. In particular, if we ask you to show that a language is regular, you can provide a regular expression, DFA, NFA, or boolean combination without justification. Similarly, if we ask you to give a DFA or NFA, you to not have to name or describe the states.

• Finally, if something goes seriously wrong, send email to jeffe@illinois.edu as soon as possible explaining the situation. If you have already finished the exam but cannot submit to Gradescope for some reason, include a complete scan of your exam in your email. If you are in the middle of the exam, send Jeff email, finish the exam (if you can) within the time limit, and then send a second email with your completed exam.
1. 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 and proving that the set you construct is indeed a fooling set for that language).

(a) $\{0^p 1^q 0^r \mid p = (q + r) \mod 2\}$
(b) $\{0^p 1^q 0^r \mid p = q + r\}$

2. Let $L$ be any regular language over the alphabet $\Sigma = \{0, 1\}$.
Let $\text{compress}(w)$ be a function that takes a string $w$ as input, and returns the string formed by compressing every run of $0$s in $w$ by half. Specifically, every run of $2n$ $0$s is compressed to length $n$, and every run of $2n + 1$ $0$s is compressed to length $n + 1$. For example:

$\text{compress}(0000110001) = 00011001$
$\text{compress}(11000010) = 110010$
$\text{compress}(11111) = 11111$

Choose exactly one of the following languages, and prove that your chosen language is regular. (In fact, both languages are regular, but we only want a proof for one of them.) Don’t forget to tell us which language you’ve chosen!

(a) $\{w \in \Sigma^* \mid \text{compress}(w) \in L\}$
(b) $\{\text{compress}(w) \mid w \in L\}$

3. Recall that the greatest common divisor of two positive integers $p$ and $q$, written $\gcd(p, q)$, is the largest positive integer $r$ that divides both $p$ and $q$. For example, $\gcd(21, 15) = 3$ and $\gcd(3, 74) = 1$.

Prove that the following languages are not regular by building an infinite fooling set for each of them. For each language, prove that the set you constructed is indeed a fooling set.

(a) $\{0^p 1^q 0^r \mid p > 0$ and $q > 0$ and $r = \gcd(p, q)\}.$
(b) $\{0^p 1^pq \mid p > 0$ and $q > 0\}$

4. Consider the following recursive function, $\text{RO}$ (short for remove-ones) that operates on any string $w \in \Sigma^*$, where $\Sigma = \{0, 1\}$:

$$\text{RO}(w) := \begin{cases} 
\varepsilon & \text{if } w = \varepsilon \\
0 \cdot \text{RO}(x) & \text{if } w = 0 \cdot x \text{ for some string } x \\
\text{RO}(x) & \text{if } w = 1 \cdot x \text{ for some string } x 
\end{cases}$$

(a) Prove that $|\text{RO}(w)| \leq |w|$ for all strings $w$.
(b) Prove that $\text{RO}(\text{RO}(w)) = \text{RO}(w)$ for all strings $w$.

(There’s one more question on the next page)
5. For each statement below, write “Yes” if the statement is always true and write “No” otherwise, and give a brief (one short sentence) explanation of your answer. Read these statements very carefully—small details matter!

(a) \{0^n1 | n > 0\} is the only infinite fooling set for the language \{0^n10^n | n > 0\}.

(b) \{0^n10^n | n > 0\} is a context-free language.

(c) The context-free grammar \(S \rightarrow 00S | S11 | 01\) generates the language \(0^n1^n\).

(d) Any language that can be decided by an NFA with \(\epsilon\)-transitions can also be decided by an NFA without \(\epsilon\)-transitions.

(e) For any string \(w \in (\{0, 1\})^*\), let \(w^C\) denote the string obtained by flipping every \(0\) in \(w\) to \(1\), and every \(1\) in \(w\) to \(0\).

If \(L\) is a regular language over the alphabet \(\{0, 1\}\), then \(\{ww^C | w \in L\}\) is also regular.

(f) For any string \(w \in (\{0, 1\})^*\), let \(w^C\) denote the string obtained by flipping every \(0\) in \(w\) to \(1\), and every \(1\) in \(w\) to \(0\).

If \(L\) is a regular language over the alphabet \(\{0, 1\}\), then \(\{xy^C | x, y \in L\}\) is also regular.

(g) The \(\epsilon\)-reach of any state in an NFA contains the state itself.

(h) Let \(L_1, L_2\) be two regular languages. The language \((L_1 + L_2)^*\) is also regular.

(i) The regular expression \((00 + 11)^*\) represents the language of all strings over \(\{0, 1\}\) of even length.

(j) The language \(\{0^{2p} | p \text{ is prime}\}\) is regular.
Clearly indicate the following structures in the directed graph below, or write NONE if the indicated structure does not exist. Don’t be subtle; to indicate a collection of edges, draw a heavy black line along the entire length of each edge.

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

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

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

(d) The shortest directed cycle.

[scratch]
A vertex \( v \) in a (weakly) connected graph \( G \) is called a **cut vertex** if the subgraph \( G - v \) is disconnected. For example, the following graph has three cut vertices, which are shaded in the figure.

Suppose you are given a (weakly) connected **dag** \( G \) with one source and one sink. Describe and analyze an algorithm that returns **TRUE** if \( G \) has a cut vertex and **FALSE** otherwise.
You decide to take your next hiking trip in Jellystone National Park. You have a map of the park’s trails that lists all the scenic views in the park, but also warns that certain trail segments have a high risk of bear encounters. To make the hike worthwhile, you want to see at least three scenic views. You also don’t want to get eaten by a bear, so you are willing to hike along at most one high-bear-risk segment. Because the trails are narrow, each trail segment allows traffic in only one direction.

Your friend has converted the map into a directed graph $G = (V, E)$, where $V$ is the set of intersections and $E$ is the set of trail segments. A subset $S$ of the edges are marked as *Scenic*; another subset $B$ of the edges are marked as *high-Bear-risk*. You may assume that $S \cap B = \emptyset$. Each segment $e \in E$ is also labeled with a positive length $\ell(e)$ in miles. Your campsite appears on the map as a particular vertex $s \in V$, and the visitor center is another vertex $t \in V$.

Describe and analyze an algorithm to compute the shortest hike from your campsite $s$ to the visitor center $t$ that includes *at least* three scenic trail segments and *at most* one high-bear-risk trail segment. You may assume such a hike exists.
During a family reunion over Thanksgiving break, your ultra-competitive thirteen-year-old nephew Elmo challenges you to a card game. At the beginning of the game, Elmo deals a long row of cards. Each card shows a number of points, which could be positive, negative, or zero. After the cards are dealt, you and Elmo alternate taking either the leftmost card or the rightmost card from the row, until all the cards are gone. The player that collects the most points is the winner.

For example, if the initial card values are \([4, 6, 1, 2]\), the game might proceed as follows:

- You take the 4 on the left, leaving \([6, 1, 2]\).
- Elmo takes the 6 on the left, leaving \([1, 2]\).
- You take the 2 on the right, leaving \([1]\).
- Elmo takes the last 1, ending the game.
- You took \(4 + 2 = 6\) points, and Elmo took \(6 + 1 = 7\) points, so Elmo wins!

Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against a perfect opponent. Assume that Elmo generously lets you move first.
For this problem, a \textit{subtree} of a binary tree means any connected subgraph. A binary tree is \textit{complete} if every internal node has two children, and every leaf has exactly the same depth.

Describe and analyze a recursive algorithm to compute the \textit{largest complete subtree} of a given binary tree. Your algorithm should return both the root and the depth of this subtree. For example, given the following tree $T$ as input, your algorithm should return the left child of the root of $T$ and the integer 2.
Directions

• Don’t panic!

• If you brought anything except your writing implements, your hand-written double-sided 8½" × 11" cheat sheet, please put it away for the duration of the exam. In particular, please turn off and put away all medically unnecessary electronic devices.

• We strongly recommend reading the entire exam before trying to solve anything. If you think a question is unclear or ambiguous, please ask for clarification as soon as possible.

• The exam has five numbered questions, each worth 10 points. (Subproblems are not necessarily worth the same number of points.)

• Write your answers on blank white paper using a dark pen. Please start your solution to each numbered question on a new sheet of paper.

• You have 120 minutes to write your solutions, after which you have 30 minutes to scan your solutions, convert your scan to a PDF file, and upload your PDF file to Gradescope.

• If you are ready to scan your solutions before 9:00pm, send a private message to the host of your Zoom call (“Ready to scan”) and wait for confirmation before leaving the Zoom call.

• Gradescope will only accept PDF submissions. Please do not scan your cheat sheets or scratch paper. Please make sure your solution to each numbered problem starts on a new page of your PDF file. Low-quality scans will be penalized.

• Proofs are required for full credit if and only if we explicitly ask for them, using the word prove in bold italics.

• Finally, if something goes seriously wrong, send email to jeffe@illinois.edu as soon as possible explaining the situation. If you have already finished the exam but cannot submit to Gradescope for some reason, include a complete scan of your exam as a PDF file in your email. If you are in the middle of the exam, send Jeff email, continue working until the time limit, and then send a second email with your completed exam as a PDF file. Please do not email raw photos.
1. Short answers:
   (a) Solve the recurrence \( T(n) = 2T(n/3) + O(pn) \).
   (b) Solve the recurrence \( T(n) = 2T(n/7) + O(\sqrt{n}) \).
   (c) Solve the recurrence \( T(n) = 2T(n/4) + O(\sqrt{n}) \).
   (d) Draw a connected undirected graph \( G \) with at most ten vertices, such that every vertex has degree at least 2, and no spanning tree of \( G \) is a path.
   (e) Draw a directed acyclic graph with at most ten vertices, exactly one source, exactly one sink, and more than one topological order.
   (f) Describe an appropriate memoization structure and evaluation order for the following (meaningless) recurrence, and give the running time of the resulting iterative algorithm to compute \( Pibby(1, n) \). (Assume all array accesses are legal.)

\[
Pibby(i, k) = \begin{cases} 
0 & \text{if } i > k \\
A[i] & \text{if } i = k \\
\max \left\{ Pibby(i + 2, k), Pibby(i + 1, k - 1), Pibby(i, k - 2) \right\} & \text{otherwise}
\end{cases}
\]

2. Your company has two offices, one in San Francisco and the other in New York. Each week you decide whether you want to work in the San Francisco office or in the New York office. Depending on the week, your company makes more money by having you work at one office or the other. You are given a schedule of the profits you can earn at each office for the next \( n \) weeks. You’d obviously prefer to work each week in the location with higher profit, but there’s a catch: Flying from one city to the other costs $1000. Your task is to design a travel schedule for the next \( n \) weeks that yields the maximum total profit, assuming you start in San Francisco.

For example: suppose you are given the following schedule:

<table>
<thead>
<tr>
<th></th>
<th>SF</th>
<th>NY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>$800</td>
<td>$300</td>
</tr>
<tr>
<td>Week 2</td>
<td>$200</td>
<td>$900</td>
</tr>
<tr>
<td>Week 3</td>
<td>$500</td>
<td>$700</td>
</tr>
<tr>
<td>Week 4</td>
<td>$400</td>
<td>$2000</td>
</tr>
<tr>
<td>Week 5</td>
<td>$1200</td>
<td>$200</td>
</tr>
</tbody>
</table>

If you spend the first week in San Francisco, the next three weeks in New York, and the last week in San Francisco, your total profit for those five weeks is $800 – $1000 + $900 + $700 + $2000 – $1000 + $1200 = $3600.

(a) Prove that the obvious greedy strategy (each week, fly to the city with more profit) does not always yield the maximum total profit.

(b) Describe and analyze an algorithm to compute the maximum total profit you can earn, assuming you start in San Francisco. The input to your algorithm is a pair of arrays \( NY[1..n] \) and \( SF[1..n] \), containing the profits in each city for each week.
3. Suppose you are given a directed graph $G = (V, E)$, whose vertices are either red, green, or blue. Edges in $G$ do not have weights, and $G$ is not necessarily a dag. The remoteness of a vertex $v$ is the maximum of three shortest-path lengths:

- The length of a shortest path to $v$ from the closest red vertex
- The length of a shortest path to $v$ from the closest blue vertex
- The length of a shortest path to $v$ from the closest green vertex

In particular, if $v$ is not reachable from vertices of all three colors, then $v$ is infinitely remote.

Describe and analyze an algorithm to find a vertex of $G$ whose remoteness is smallest.

4. Suppose you are given an array $A[1..n]$ of integers such that $A[i] + A[i + 1]$ is even for exactly one index $i$. In other words, the elements of $A$ alternate between even and odd, except for exactly one adjacent pair that are either both even or both odd.

Describe and analyze an efficient algorithm to find the unique index $i$ such that $A[i] + A[i + 1]$ is even. For example, given the following array as input, your algorithm should return the integer 6, because $A[6] + A[7] = 88 + 62$ is even. (Cells containing even integers are shaded blue.)

```
17  40  23  72  39  88  62  13  40  53  92  21  10  73  68
```

5. A zigzag walk in a directed graph $G$ is a sequence of vertices connected by edges in $G$, but the edges alternately point forward and backward along the sequence. Specifically, the first edge points forward, the second edge points backward, and so on. The length of a zigzag walk is the sum of the weights of its edges, both forward and backward.

For example, the following graph contains the zigzag walk $a \rightarrow b \leftarrow d \rightarrow f \rightarrow c \rightarrow e$. Assuming every edge in the graph has weight 1, this zigzag walk has length 5.

Suppose you are given a directed graph $G$ with non-negatively weighted edges, along with two vertices $s$ and $t$. Describe and analyze an algorithm to find the shortest zigzag walk from $s$ to $t$ in $G$. 
CS/ECE 374 A  Fall 2021
Conflict Midterm 2  November 9, 2021

Directions

• Don’t panic!

• If you brought anything except your writing implements, your hand-written double-sided 8½” × 11” cheat sheet, please put it away for the duration of the exam. In particular, please turn off and put away all medically unnecessary electronic devices.

• We strongly recommend reading the entire exam before trying to solve anything. If you think a question is unclear or ambiguous, please ask for clarification as soon as possible.

• The exam has five numbered questions, each worth 10 points. (Subproblems are not necessarily worth the same number of points.)

• Write your answers on blank white paper using a dark pen. Please start your solution to each numbered question on a new sheet of paper.

• You have 120 minutes to write your solutions, after which you have 30 minutes to scan your solutions, convert your scan to a PDF file, and upload your PDF file to Gradescope.

• If you are ready to scan your solutions before 9:00pm, send a private message to the host of your Zoom call (“Ready to scan”) and wait for confirmation before leaving the Zoom call.

• Gradescope will only accept PDF submissions. Please do not scan your cheat sheets or scratch paper. Please make sure your solution to each numbered problem starts on a new page of your PDF file. Low-quality scans will be penalized.

• Proofs are required for full credit if and only if we explicitly ask for them, using the word prove in bold italics.

• Finally, if something goes seriously wrong, send email to jeffe@illinois.edu as soon as possible explaining the situation. If you have already finished the exam but cannot submit to Gradescope for some reason, include a complete scan of your exam as a PDF file in your email. If you are in the middle of the exam, send Jeff email, continue working until the time limit, and then send a second email with your completed exam as a PDF file. Please do not email raw photos.
1. Short answers:

(a) Solve the recurrence \( T(n) = 3T(n/2) + O(n^2) \).

(b) Solve the recurrence \( T(n) = 7T(n/2) + O(n^2) \).

(c) Solve the recurrence \( T(n) = 4T(n/2) + O(n^2) \).

(d) Draw a directed acyclic graph with at most ten vertices, exactly one source, exactly one sink, and more than one topological order.

(e) Draw a directed graph with at most ten vertices, with distinct edge weights, that has more than one shortest path from some vertex \( s \) to some other vertex \( t \).

(f) Describe an appropriate memoization structure and evaluation order for the following (meaningless) recurrence, and give the running time of the resulting iterative algorithm to compute \( Huh(1, n) \).

\[
Huh(i, k) = \begin{cases} 
0 & \text{if } i > n \text{ or } k < 0 \\
\min \left\{ \frac{Huh(i + 1, k - 2)}{Huh(i + 2, k - 1)} \right\} + A[i, k] & \text{if } A[i, k] \text{ is even} \\
\max \left\{ \frac{Huh(i + 1, k - 2)}{Huh(i + 2, k - 1)} \right\} - A[i, k] & \text{if } A[i, k] \text{ is odd}
\end{cases}
\]

2. **Quadhopper** is a solitaire game played on a row of \( n \) squares. Each square contains four positive integers. The player begins by placing a token on the leftmost square. On each move, the player chooses one of the numbers on the token's current square, and then moves the token that number of squares to the right. The game ends when the token moves past the rightmost square. The object of the game is to make as many moves as possible before the game ends.

(a) Prove that the obvious greedy strategy (always choose the smallest number) does not give the largest possible number of moves for every quadhopper puzzle.

(b) Describe and analyze an efficient algorithm to find the largest possible number of legal moves for a given quadhopper puzzle.
3. Suppose you are given a directed graph $G = (V, E)$, each of whose vertices is either red, green, or blue. Edges in $G$ do not have weights, and $G$ is not necessarily a dag.

Describe and analyze an algorithm to find a shortest path in $G$ that contains at least one vertex of each color. (In particular, your algorithm must choose the best start and end vertices for the path.)

4. Your grandmother dies and leaves you her treasured collection of $n$ radioactive Beanie Babies. Her will reveals that one of the Beanie Babies is a rare specimen worth 374 million dollars, but all the others are worthless. All of the Beanie Babies are equally radioactive, except for the valuable Beanie Baby, which is either slightly more or slightly less radioactive, but you don’t know which. Otherwise, as far as you can tell, the Beanie Babies are all identical.

You have access to a state-of-the-art Radiation Comparator at your job. The Comparator has two chambers. You can place any two disjoint sets of Beanie Babies in Comparator’s two chambers; the Comparator will then indicate which subset emits more radiation, or that the two subsets are equally radioactive. (Two subsets are equally radioactive if and only if they contain the same number of Beanie Babies, and they are all worthless.) The Comparator is slow and consumes a lot of power, and you really aren’t supposed to use it for personal projects, so you really want to use it as few times as possible.

Describe an efficient algorithm to identify the valuable Beanie Baby. How many times does your algorithm use the Comparator in the worst case, as a function of $n$?

5. Ronnie and Hyde are a professional robber duo who plan to rob a house in the Leverwood neighborhood of Sham-Poobanana. They have managed to obtain a map of the neighborhood in the form of a directed graph $G$, whose vertices represent houses, whose edges represent one-way streets.

- One vertex $s$ represents the house that Ronnie and Hyde plan to rob.
- A set $X$ of special vertices designate eXits from the neighborhood.
- Each directed edge $u \rightarrow v$ has a non-negative weight $w(u \rightarrow v)$, indicating the time required to drive directly from house $u$ to house $v$.
- Thanks to Leverwood’s extensive network of traffic cameras, speeding or driving backwards along any one-way street would mean certain capture.

Describe and analyze an algorithm to compute the shortest time needed to exit the neighborhood, starting at house $s$. The input to your algorithm is the directed graph $G = (V, E)$, with clearly marked subset of exit vertices $X \subseteq V$, and non-negative weights $w(u \rightarrow v)$ for every edge $u \rightarrow v$. 
• Don’t panic!

• If you brought anything except your writing implements, your two hand-written double-sided 8½" × 11" cheat sheets, please put it away for the duration of the exam. In particular, please turn off and put away all medically unnecessary electronic devices.

• We strongly recommend reading the entire exam before trying to solve anything. If you think a question is unclear or ambiguous, please ask for clarification as soon as possible.

• The exam has six numbered questions, each worth 10 points. (Subproblems are not necessarily worth the same number of points.)

• You have 150 minutes to write your solutions, after which you have 30 minutes to scan your solutions, convert your scan to a PDF file, and upload your PDF file to Gradescope. (Both of these times are extended if you have time accommodations through DRES.)

• Proofs are required for full credit if and only if we explicitly ask for them, using the word prove in bold italics.

• Write your answers on blank white paper using a dark pen. Please start your solution to each numbered question on a new sheet of paper.

• If you are ready to scan your solutions and there are more than 15 minutes of writing time remaining, send a private message to the host of your Zoom call (“Ready to scan”) and wait for confirmation before leaving the Zoom call.

• Gradescope will only accept PDF submissions. Please do not scan your cheat sheets or scratch paper. Please make sure your solution to each numbered problem starts on a new page of your PDF file.

• Finally, if something goes seriously wrong, send email to jeffe@illinois.edu as soon as possible explaining the situation. If you have already finished the exam but cannot submit to Gradescope for some reason, include a complete scan of your exam as a PDF file in your email. If you are in the middle of the exam, send Jeff email, continue working until the time limit, and then send a second email with your completed exam as a PDF file. Please do not email raw photos.
Some useful NP-hard problems. You are welcome to use any of these in your own NP-hardness proofs, except of course for the specific problem you are trying to prove NP-hard.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**IntegerLinearProgramming:** Given a matrix $A \in \mathbb{Z}^{n \times d}$ and two vectors $b \in \mathbb{Z}^n$ and $c \in \mathbb{Z}^d$, 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?

**SteamedHams:** Aurora borealis? At this time of year, at this time of day, in this part of the country, localized entirely within your kitchen? May I see it?
1. For each statement below, write “YES” if the statement is always true and “NO” otherwise, and give a brief (at most one short sentence) explanation of your answer. Assume $P \neq NP$. If there is any other ambiguity or uncertainty about an answer, write “NO”. For example:

- $x + y = 5$
  **NO** — Suppose $x = 3$ and $y = 4$.

- 3SAT can be solved in polynomial time.
  **NO** — 3SAT is NP-hard.

- If $P = NP$ then Jeff is the Queen of England.
  **YES** — The hypothesis is false, so the implication is true.

Read each statement very carefully; some of these are deliberately subtle!

Which of the following statements are true?

(a) The solution to the recurrence $T(n) = 4T(n/2) + O(n^2)$ is $T(n) = O(n^2)$.
(b) The solution to the recurrence $T(n) = 2T(n/4) + O(n^2)$ is $T(n) = O(n^2)$.
(c) Every directed acyclic graph contains at least one sink.
(d) Given any undirected graph $G$, we can compute a spanning tree of $G$ in $O(V + E)$ time using whatever-first search.
(e) Suppose we want to iteratively evaluate the following recurrence:

$$
\text{What}(i, j) = \begin{cases} 
0 & \text{if } i > n \text{ or } j < 0 \\
\max \left\{ \begin{array}{l}
\text{What}(i, j - 1) \\
\text{What}(i + 1, j) \\
A[i] \cdot A[j] + \text{What}(i + 1, j - 1)
\end{array} \right. & \text{otherwise}
\end{cases}
$$

We can fill the array $\text{What}[0..n, 0..n]$ in $O(n^2)$ time, by decreasing $i$ in the outer loop and decreasing $j$ in the inner loop.

Which of the following statements are true for at least one language $L \subseteq \{0, 1\}^*$?

(f) $L^* = (L^*)^*$
(g) $L$ is decidable, but $L^*$ is undecidable.
(h) $L$ is neither regular nor NP-hard.
(i) $L$ is in P, and $L$ has an infinite fooling set.
(j) The language $\{\langle M \rangle \mid M \text{ accepts } L\}$ is undecidable.
2. For each statement below, write “YES” if the statement is always true and “NO” otherwise, and give a brief (at most one short sentence) explanation of your answer. Assume $P \neq NP$. If there is any other ambiguity or uncertainty about an answer, write “NO”.

Read each statement very carefully; some of these are deliberately tricky!

(Please remember to start your answers to this problem on a new page. Yes, this is really just a continuation of problem 1; we split it into two problems to make grading easier.)

Consider the following pair of languages:

- $\text{ACYCLIC} := \{\text{undirected graph } G \mid G \text{ contains no cycles}\}$
- $\text{HALFIND} := \{\text{undirected graph } G = (V, E) \mid G \text{ has an independent set of size } |V|/2\}$

(For concreteness, assume that in both of these languages, graphs are represented by their adjacency matrices.) The language $\text{HALFIND}$ is actually NP-hard; you do not need to prove that fact.

Which of the following statements are true, assuming $P \neq NP$?

(a) $\text{ACYCLIC}$ is NP-hard.

(b) $\text{HALFIND} \setminus \text{ACYCLIC} \in P$
   
   (Recall that $X \setminus Y$ is the subset of elements of $X$ that are not in $Y$.)

(c) $\text{HALFIND}$ is decidable.

(d) A polynomial-time reduction from $\text{HALFIND}$ to $\text{ACYCLIC}$ would imply $P=NP$.

(e) A polynomial-time reduction from $\text{ACYCLIC}$ to $\text{HALFIND}$ would imply $P=NP$.

Suppose there is a polynomial-time reduction from some language $A$ over the alphabet $\{0, 1\}$ to some other language $B$ over the alphabet $\{0, 1\}$. Which of the following statements are true, assuming $P \neq NP$?

(f) $A$ is a subset of $B$.

(g) If $B \in P$, then $A \in P$.

(h) If $B$ is NP-hard, then $A$ is NP-hard.

(i) If $B$ is decidable, then $A$ is decidable.

(j) If $B$ is regular, then $A$ is decidable.
3. Suppose you are asked to tile a \(2 \times n\) grid of squares with dominos (1 \(\times\) 2 rectangles). Each domino must cover exactly two grid squares, either horizontally or vertically, and each grid square must be covered by exactly one domino.

Each grid square is worth some number of points, which could be positive, negative, or zero. The value of a domino tiling is the sum of the points in squares covered by vertical dominos, minus the sum of the points in squares covered by horizontal dominos.

Describe and analyze an efficient algorithm to compute the largest possible value of a domino tiling of a given \(2 \times n\) grid. Your input is an array \(\text{Points}[1..2, 1..n]\) of point values.

As an example, here are three domino tilings of the same \(2 \times 6\) grid, along with their values. The third tiling is optimal; no other tiling of this grid has larger value. Thus, given this \(2 \times 6\) grid as input, your algorithm should return the integer 16.

\[
\begin{array}{cccccc}
5 & 2 & -3 & 2 & -7 & 3 \\
1 & -6 & 0 & -1 & 4 & -2 \\
\end{array}
\]

\[
\begin{array}{cccccc}
5 & 2 & -3 & 2 & -7 & 3 \\
1 & -6 & 0 & -1 & 4 & -2 \\
\end{array}; \text{ value = -6}
\]

\[
\begin{array}{cccccc}
5 & 2 & -3 & 2 & -7 & 3 \\
1 & -6 & 0 & -1 & 4 & -2 \\
\end{array}; \text{ value = 2}
\]

\[
\begin{array}{cccccc}
5 & 2 & -3 & 2 & -7 & 3 \\
1 & -6 & 0 & -1 & 4 & -2 \\
\end{array}; \text{ value = 16}
\]

4. Submit a solution to exactly one of the following problems. Don’t forget to tell us which problem you’ve chosen!

(a) Let \(\Phi\) be a boolean formula in conjunctive normal form, with exactly three literals per clause (or in other words, an instance of 3Sat). Prove that it is NP-hard to decide whether \(\Phi\) has a satisfying assignment in which exactly half of the variables are TRUE.

(b) Let \(G = (V,E)\) be an arbitrary undirected graph. Recall that a proper 3-coloring of \(G\) assigns each vertex of \(G\) one of three colors—red, blue, or green—so that every edge in \(G\) has endpoints with different colors. Prove that it is NP-hard to decide whether \(G\) has a proper 3-coloring in which exactly half of the vertices are red.

(In fact, both of these problems are NP-hard, but we only want a proof for one of them.)

5. Suppose you are given a height map of a mountain, in the form of an \(n \times n\) grid of evenly spaced points, each labeled with an elevation value. You can safely hike directly from any point to any neighbor immediately north, south, east, or west, but only if the elevations of those two points differ by at most \(\Delta\). (The value of \(\Delta\) depends on your hiking experience and your physical condition.)

Describe and analyze an algorithm to determine the longest hike from some point \(s\) to some other point \(t\), where the hike consists of an uphill climb (where elevations must increase at each step) followed by a downhill climb (where elevations must decrease at each step). Your input consists of an array \(\text{Elevation}[1..n, 1..n]\) of elevation values, the starting point \(s\), the target point \(t\), and the parameter \(\Delta\).
6. Recall that a run in a string $w \in \{0, 1\}^*$ is a maximal substring of $w$ whose characters are all equal. For example, the string $00011111110000$ is the concatenation of three runs:

$$00011111110000 = 000 \cdot 1111111 \cdot 000$$

(a) Let $L_a$ denote the set of all strings in $\{0, 1\}^*$ where every 0 is followed immediately by at least one 1.

For example, $L_a$ contains the strings $010111$ and $1111$ and the empty string $\epsilon$, but does not contain either $001100$ or $111110$.

- Describe a DFA or NFA that accepts $L_a$ and
- Give a regular expression that describes $L_a$.

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

(b) Let $L_b$ denote the set of all strings in $\{0, 1\}^*$ whose run lengths are increasing; that is, every run except the last is followed immediately by a longer run.

For example, $L_b$ contains the strings $01001111$ and $110000$ and $000$ and the empty string $\epsilon$, but does not contain either $000111$ or $100011$.

Prove that $L_b$ is not a regular language.
Directions

• Don’t panic!

• If you brought anything except your writing implements, your two hand-written double-sided 8½" × 11" cheat sheets, please put it away for the duration of the exam. In particular, please turn off and put away all medically unnecessary electronic devices.

• We strongly recommend reading the entire exam before trying to solve anything. If you think a question is unclear or ambiguous, please ask for clarification as soon as possible.

• The exam has six numbered questions, each worth 10 points. (Subproblems are not necessarily worth the same number of points.)

• You have 150 minutes to write your solutions, after which you have 30 minutes to scan your solutions, convert your scan to a PDF file, and upload your PDF file to Gradescope. (Both of these times are extended if you have time accommodations through DRES.)

• Proofs are required for full credit if and only if we explicitly ask for them, using the word prove in bold italics.

• Write your answers on blank white paper using a dark pen. Please start your solution to each numbered question on a new sheet of paper.

• If you are ready to scan your solutions and there are more than 15 minutes of writing time, send a private message to the host of your Zoom call (“Ready to scan”) and wait for confirmation before leaving the Zoom call.

• Gradescope will only accept PDF submissions. Please do not scan your cheat sheets or scratch paper. Please make sure your solution to each numbered problem starts on a new page of your PDF file.

• Finally, if something goes seriously wrong, send email to jeffe@illinois.edu as soon as possible explaining the situation. If you have already finished the exam but cannot submit to Gradescope for some reason, include a complete scan of your exam as a PDF file in your email. If you are in the middle of the exam, send Jeff email, continue working until the time limit, and then send a second email with your completed exam as a PDF file. Please do not email raw photos.
Some useful NP-hard problems. You are welcome to use any of these in your own NP-hardness proofs, except of course for the specific problem you are trying to prove NP-hard.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**IntegerLinearProgramming:** Given a matrix $A \in \mathbb{Z}^{n \times d}$ and two vectors $b \in \mathbb{Z}^n$ and $c \in \mathbb{Z}^d$, 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?

**SteamedHams:** Aurora borealis? At this time of year, at this time of day, in this part of the country, localized entirely within your kitchen? May I see it?
1. For each statement below, write “YES” if the statement is always true and “NO” otherwise, and give a brief (at most one short sentence) explanation of your answer. Assume $P \neq NP$. If there is any other ambiguity or uncertainty about an answer, write “NO”. For example:

- $x + y = 5$
  NO — Suppose $x = 3$ and $y = 4$.
- 3SAT can be solved in polynomial time.
  NO — 3SAT is NP-hard.
- If $P = NP$ then Jeff is the Queen of England.
  YES — The hypothesis is false, so the implication is true.

Read each statement very carefully; some of these are deliberately subtle!

Which of the following statements are true?

(a) The solution to the recurrence $T(n) = 2T(n/4) + O(n^2)$ is $T(n) = O(n^2)$.
(b) The solution to the recurrence $T(n) = 4T(n/2) + O(n^2)$ is $T(n) = O(n^2)$.
(c) For every directed graph $G$, if $G$ has at least one source, then $G$ has at least one sink.
(d) Given any undirected graph $G$, we can compute a spanning tree of $G$ in $O(V + E)$ time using whatever-first search.
(e) Suppose we want to iteratively evaluate the following recurrence:

\[
What(i, j) = \begin{cases} 
  0 & \text{if } i < 0 \text{ or } j < 0 \\
  \max \left\{ What(i, j - 1), What(i - 1, j) \right\} & \text{otherwise} \\
  A[i] \cdot A[j] + What(i - 1, j - 1) & \text{otherwise}
\end{cases}
\]

We can fill the array $What[0..n, 0..n]$ in $O(n^2)$ time, by decreasing $i$ in the outer loop and decreasing $j$ in the inner loop.

Which of the following statements are true for all languages $L \subseteq \{0, 1\}^*$?

(f) $L^* = (L^*)^*$
(g) If $L$ is decidable, then $L^*$ is decidable.
(h) $L$ is either regular or NP-hard.
(i) If $L$ is undecidable, then $L$ has an infinite fooling set.
(j) The language $\{\langle M \rangle \mid M \text{ decides } L \}$ is undecidable.
2. For each statement below, write “YES” if the statement is always true and “NO” otherwise, and give a brief (at most one short sentence) explanation of your answer. Assume $P \neq NP$. If there is any other ambiguity or uncertainty about an answer, write “NO”.

Read each statement very carefully; some of these are deliberately tricky!

(Please remember to start your answers to this problem on a new page. Yes, this is really just a continuation of problem 1; we split it into two problems to make grading easier.)

Consider the following pair of languages:

- $\text{DirHamPath} := \{ G \mid G$ is a directed graph with a Hamiltonian path $\}$
- $\text{Acyclic} := \{ G \mid G$ is a directed acyclic graph $\}$

(For concreteness, assume that in both of these languages, graphs are represented by their adjacency matrices.) Which of the following statements are true, assuming $P \neq NP$?

(a) $\text{Acyclic} \in NP$
(b) $\text{Acyclic} \cap \text{DirHamPath} \in P$
(c) $\text{DirHamPath}$ is decidable.
(d) A polynomial-time reduction from $\text{DirHamPath}$ to $\text{Acyclic}$ would imply $P=NP$.
(e) A polynomial-time reduction from $\text{Acyclic}$ to $\text{DirHamPath}$ would imply $P=NP$.

Suppose there is a polynomial-time reduction from some language $A \subseteq \{0, 1\}$ reduces to some other language $B \subseteq \{0, 1\}$. Which of the following statements are true, assuming $P \neq NP$?

(f) $A \subseteq B$.
(g) There is an algorithm to transform any Python program that solves $B$ in polynomial time into a Python program that solves $A$ in polynomial time.
(h) If $A$ is NP-hard then $B$ is NP-hard.
(i) If $A$ is decidable then $B$ is decidable.
(j) If a Turing machine $M$ accepts $B$, the same Turing machine $M$ also accepts $A$. 


3. Aladdin and Badroulbadour are playing a cooperative game. Each player has an array of positive integers, arranged in a row of squares from left to right. Each player has a token, which starts at the leftmost square of their row; their goal is to move both tokens to the rightmost squares.

On each turn, both players move their tokens in the same direction, either left or right. The distance each token travels is equal to the number under that token at the beginning of the turn. For example, if a token starts on a square labeled 5, then it moves either five squares to the right or five squares to the left. If either token moves past either end of its row, then both players immediately lose.

For example, if Aladdin and Badroulbadour are given the arrays

\[
\begin{align*}
A : & \quad 7 \ 5 \ 4 \ 1 \ 2 \ 3 \ 3 \ 2 \ 3 \ 1 \ 4 \ 2 \\
B : & \quad 5 \ 1 \ 2 \ 4 \ 7 \ 3 \ 5 \ 2 \ 4 \ 6 \ 3 \ 1
\end{align*}
\]

they can win the game by moving right, left, left, right, right, left, right. On the other hand, if they are given the arrays

\[
\begin{align*}
A : & \quad 2 \ 3 \ 5 \ 1 \ 3 \\
B : & \quad 3 \ 4 \ 1 \ 2 \ 1
\end{align*}
\]

they cannot win the game. (The first move must be to the right; then Aladdin’s token moves out of bounds on the second turn.)

Describe and analyze an algorithm to determine whether Aladdin and Badroulbadour can solve their puzzle, given the input arrays A[1..n] and B[1..n].

4. Submit a solution to exactly one of the following problems. Don’t forget to tell us which problem you’ve chosen!

(a) Let \( G = (V, E) \) be an arbitrary undirected graph. A subset \( S \subseteq V \) of vertices is mostly independent if less than half the vertices of \( S \) have a neighbor that is also in \( S \). Prove that finding the largest mostly independent set in \( G \) is NP-hard.

(b) Let \( G = (V, E) \) be an arbitrary directed graph with colored edges. A rainbow Hamiltonian cycle in \( G \) is a cycle that visits every vertex of \( G \) exactly one, in which no pair of consecutive edges have the same color. Prove that it is NP-hard to decide whether \( G \) has a rainbow Hamiltonian cycle.

(In fact, both of these problems are NP-hard, but we only want a proof for one of them.)
5. Suppose we are given an \( n \)-digit integer \( X \). Repeatedly remove one digit from either end of \( X \) (your choice) until no digits are left. The square-depth of \( X \) is the maximum number of perfect squares that you can see during this process. For example, the number 32492 has square-depth 3, by the following sequence of removals:

\[
32492 \rightarrow 3249 \rightarrow 324 \rightarrow 24 \rightarrow 4 \rightarrow \varepsilon.
\]

Describe and analyze an algorithm to compute the square-depth of a given integer \( X \), represented as an array \( X[1..n] \) of \( n \) decimal digits. Assume you have access to a subroutine \textsc{IsSquare} that determines whether a given \( k \)-digit number (represented by an array of digits) is a perfect square in \( O(k^2) \) time.

6. Recall that a run in a string \( w \in \{0,1\}^* \) is a maximal substring of \( w \) whose characters are all equal. For example, the string 0001111110000 is the concatenation of three runs:

\[
0001111110000 = 000 \circ 111111 \circ 0000
\]

(a) Let \( L_a \) denote the set of all strings in \( \{0,1\}^* \) in which every run of 1s has even length and every run of 0s has odd length.

- Describe a DFA or NFA that accepts \( L_a \) and
- Give a regular expression that describes \( L_a \).

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

(b) Let \( L_b \) denote the set of all strings in \( \{0,1\}^* \) in which every run of 0s is immediately followed by a longer run of 1s. Prove that \( L_b \) is not a regular language.

Both of these languages contain the strings 0111100011 and 110001111 and 111111 and the empty string \( \varepsilon \), but neither language contains 000111 or 100011 or 0000.