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

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

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

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

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

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

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

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

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

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

1. Prove that $|w| = |w^R|$ for every string w.

2. Prove that $(w \bullet z)^R = z^R \bullet w^R$ for all strings w and z.

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

[Hint: The proof for problem 3 relies on problem 2, but it may be easier to solve problem 3 first.]

To think about later: Let $\#(a, w)$ denote the number of times symbol a appears in string w. For example, $\#(\texttt{X}, \texttt{WTF374}) = 0$ and $\#(\texttt{0}, \texttt{000010101010010100}) = 12$.

4. Give a formal recursive definition of $\#(a, w)$.

5. Prove that $\#(a, w \bullet z) = \#(a, w) + \#(a, z)$ for all symbols a and all strings w and z.

6. Prove that $\#(a, w^R) = \#(a, w)$ for all symbols a and all strings w.

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

1. All strings containing the substring 000.

2. All strings *not* containing the substring 000.

3. All strings in which every run of 0s has length at least 3.

4. All strings in which every 1 appears before every substring 000.

5. All strings containing at least three 0s.

6. Every string except 000. *[Hint: Don't try to be clever.]*

Work on these later:

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

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

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

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

Describe deterministic finite-state automata that accept each of the following languages over the alphabet $\Sigma = \{0, 1\}$. Give the states of your DFAs mnemonic names, and describe briefly *in English* the meaning or purpose of each state.

Either drawings or formal descriptions are acceptable, as long as the states Q, the start state s, the accept states A, and the transition function δ are all be clear. Try not to use too many states, but *don't* try to use as few states as possible.

Yes, these are exactly the same languages that you saw last Friday.

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

More difficult problems to think about later:

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

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

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

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

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

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

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

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

Harder problems to think about later:

4. All strings in which the subsequence 0101 appears an even number of times.

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

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

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

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

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

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

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

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

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

Work on these later:

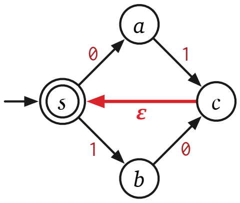
6. Strings of the form $w_1\mathtt{\#}w_2\mathtt{\#}\cdots\mathtt{\#}w_n$ for some $n \geq 2$, where each substring w_i is a string in $\{\mathtt{0},\mathtt{1}\}^*$, and some pair of substrings w_i and w_j are equal.

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

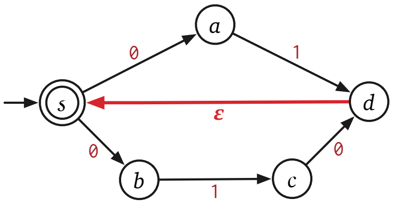
*8. $\{w \in (\mathtt{0}+\mathtt{1})^* \mid w \text{ is the binary representation of a perfect square}\}$

Convert each of the following NFAs into an equivalent DFA, using the incremental subset construction.

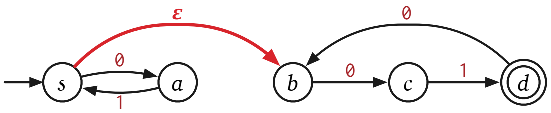
1.



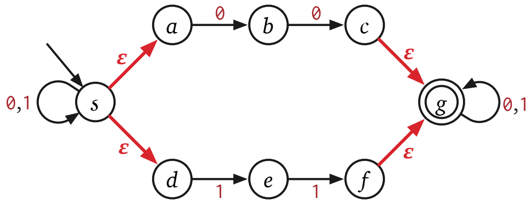
2.



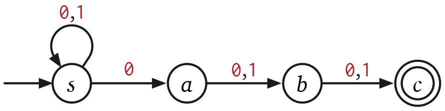
3.



4.



5.



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

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

$$\mathit{flipOdds}(\texttt{\underline{0}0\underline{0}0\underline{1}1\underline{1}1\underline{0}1\underline{0}1\underline{0}1\underline{0}0}) = \texttt{\underline{1}0\underline{1}0\underline{0}1\underline{0}1\underline{1}1\underline{1}1\underline{1}1\underline{1}0}$$

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

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

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

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

■

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

$$\mathit{flipOdd1s}(\mathtt{0000\underline{1}1\underline{1}100\underline{1}010\underline{1}0}) = \mathtt{0000\underline{0}1\underline{0}100\underline{0}010\underline{0}0}$$

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

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

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

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

■

3. FlipOdd1s$(L) := \{\mathit{flipOdd1s}(w) \mid w \in L\}$, where the function *flipOdd1* is defined as in the previous problem.

Solution: Let $M = (Q, s, A, \delta)$ be an arbitrary DFA that accepts L. We construct a new **NFA** $M' = (Q', s', A', \delta')$ that accepts FlipOdd1s(L) as follows.

Intuitively, M' receives some string *flipOdd1s*(w) as input, ***guesses*** which 0 bits to restore to 1s, and simulates M on the restored string w. No string in FlipOdd1s(L) has two 1s in a row, so if M' ever sees 11, it must reject.

Each state (q, flip) of M' indicates that M is in state q, and we need to flip some 0 bit before the next 1 bit if $\mathit{flip} = \text{True}$.

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

■

4. $\textsc{Shuffle}(L) := \{\mathit{shuffle}(w,x) \mid w, x \in L \text{ and } |w| = |x|\}$, where the function *shuffle* is defined recursively as follows:

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

For example, $\mathit{shuffle}(\texttt{0001101}, \texttt{1111001}) = \texttt{01010111100011}$.

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

Intuitively, M' reads the string $\mathit{shuffle}(w,x)$ as input, splits the string into the subsequences w and x, and passes those strings to two independent copies of M. Let M_1 denote the copy that processes the first string w, and let M_2 denote the copy that processes the second string x.

Each state $(q_1, q_2, \mathit{next})$ indicates that machine M_1 is in state q_1, machine M_2 is in state q_2, and *next* indicates whether M_1 or M_2 receives the next input bit.

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

■

You saw the following context-free grammars in class on Thursday; in each example, the grammar itself is on the left; the explanation for each non-terminal is on the right.

- Properly nested strings of parentheses.

  $S \to \varepsilon \mid S\texttt{(}S\texttt{)}$ properly nested parentheses

  Here is a different grammar for the same language:

  $S \to \varepsilon \mid \texttt{(}S\texttt{)} \mid SS$ properly nested parentheses

- $\{\texttt{0}^m\texttt{1}^n \mid m \neq n\}$. This is the set of all binary strings composed of some number of 0s followed by a *different* number of 1s.

$S \to A \mid B$	$\{\texttt{0}^m\texttt{1}^n \mid m \neq n\}$
$A \to \texttt{0}A \mid \texttt{0}C$	$\{\texttt{0}^m\texttt{1}^n \mid m > n\}$
$B \to B\texttt{1} \mid C\texttt{1}$	$\{\texttt{0}^m\texttt{1}^n \mid m < n\}$
$C \to \varepsilon \mid \texttt{0}C\texttt{1}$	$\{\texttt{0}^m\texttt{1}^n \mid m = n\}$

Give context-free grammars for each of the following languages over the alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$. For each grammar, describe the language for each non-terminal, either in English or using mathematical notation, as in the examples above. We probably won't finish all of these during the lab session.

1. All palindromes in Σ^*

2. All palindromes in Σ^* that contain an even number of 1s

3. All palindromes in Σ^* that end with 1

4. All palindromes in Σ^* whose length is divisible by 3

5. All palindromes in Σ^* that do not contain the substring 00

Harder problems to work on later:

6. $\{0^{2n}1^n \mid n \geq 0\}$

7. $\{0^m 1^n \mid m \neq 2n\}$

   [Hint: If $m \neq 2n$, then either $m < 2n$ or $m > 2n$. Extend the previous grammar, but pay attention to parity. This language contains the string 01*.]*

8. $\{0,1\}^* \setminus \{0^{2n}1^n \mid n \geq 0\}$

   [Hint: Extend the previous grammar. What's missing?]

9. $\{w \in \{0,1\}^* \mid \#(0,w) = 2 \cdot \#(1,w)\}$ — Binary strings where the number of 0s is exactly twice the number of 1s.

*10. $\{0,1\}^* \setminus \{ww \mid w \in \{0,1\}^*\}$.

   *[Anti-hint: The language $\{ww \mid w \in 0,1^*\}$ is **not** context-free. Thus, the complement of a context-free language is not necessarily context-free!]*

Consider the following recursively defined function on strings:

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

Intuitively, $stutter(w)$ doubles every symbol in w. For example:

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

Let L be an arbitrary regular language.

1. Prove that the language $\textsc{Unstutter}(L) := \{w \mid stutter(w) \in L\}$ is regular.
2. Prove that the language $\textsc{Stutter}(L) := \{stutter(w) \mid w \in L\}$ is regular.

Work on these later:

3. Let L be an arbitrary regular language.

   (a) Prove that the language $\textsc{Insert}\texttt{1}(L) := \{x\texttt{1}y \mid xy \in L\}$ is regular.
   Intuitively, $\textsc{Insert}\texttt{1}(L)$ is the set of all strings that can be obtained from strings in L by inserting exactly one $\texttt{1}$. For example:

   $$\textsc{Insert}\texttt{1}(\{\varepsilon, \texttt{00}, \texttt{101101}\}) = \{\texttt{1}, \texttt{100}, \texttt{010}, \texttt{001}, \texttt{1101101}, \texttt{1011101}, \texttt{1011011}\}$$

   (b) Prove that the language $\textsc{Delete}\texttt{1}(L) := \{xy \mid x\texttt{1}y \in L\}$ is regular.
   Intuitively, $\textsc{Delete}\texttt{1}(L)$ is the set of all strings that can be obtained from strings in L by deleting exactly one $\texttt{1}$. For example:

   $$\textsc{Delete}\texttt{1}(\{\varepsilon, \texttt{00}, \texttt{101101}\}) = \{\texttt{01101}, \texttt{10101}, \texttt{10110}\}$$

4. Consider the following recursively defined function on strings:

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

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

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

   Once again, let L be an arbitrary regular language.

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

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

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

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

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

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

9	**7**	7	2	**1**	3	7	5	**4**	7	**3**	**3**	4	8	**6**	9
	▲			▲				▲		▲	▲			▲	

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

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

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

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

Harder problem to think about later:

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

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

   your algorithm should return the integer 7.

In lecture on Thursday, we saw a divide-and-conquer algorithm, due to Karatsuba, that multiplies two n-digit integers using $O(n^{\lg 3})$ single-digit additions, subtractions, and multiplications. In this lab, we'll look at one application of Karatsuba's algorithm: converting a number from binary to decimal.

(The standard algorithm that computes one decimal digit of x at a time, by computing $x \bmod 10$ and then recursively converting $\lfloor x/10 \rfloor$, requires $\Theta(n^2)$ time.)

1. Consider the following recurrence, originally used by the Sanksrit prosodist Piṅgala in the second century BCE, to compute the number 2^n:

$$2^n = \begin{cases} 1 & \text{if } n = 0 \\ (2^{n/2})^2 & \text{if } n > 0 \text{ is even} \\ 2 \cdot (2^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is odd} \end{cases}$$

We can use this algorithm to compute the decimal representation of 2^n, by representing all numbers using arrays of decimal digits, and implementing squaring and doubling using decimal arithmetic. Suppose we use Karatsuba's algorithm for decimal multiplication. What is the running time of the resulting algorithm?

2. We can use a similar algorithm to compute the decimal representation of any integer. Suppose we are given an integer x as an array of n bits (binary digits). Write $x = a \cdot 2^{n/2} + b$, where a is represented by the top $n/2$ bits of x, and b is represented by the bottom $n/2$ bits of x. Then we can convert x into decimal as follows:

   (a) Recursively convert a into decimal.
   (b) Recursively convert $2^{n/2}$ into decimal.
   (c) Recursively convert b into decimal.
   (d) Compute $x = a \cdot 2^{n/2} + b$ using decimal multiplication and addition.

Now suppose we use Karatsuba's algorithm for decimal multiplication. What is the running time of the resulting algorithm? (For simplicity, you can assume n is a power of 2.)

3. Now suppose instead of converting $2^{n/2}$ to decimal by recursively calling the algorithm from problem 2, we use the specialized algorithm for powers of 2 from problem 1. Now what is the running time of the resulting algorithm (assuming we use Karatsuba's multiplication algorithm as before)?

Harder problem to think about about later:

4. In fact, it is possible to multiply two n-digit decimal numbers in $O(n \log n)$ time. Describe an algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(n \log^2 n)$ time.

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

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

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

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

   For example, given the array

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

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

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

   For example, given the array

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

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

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

   For example, given the array

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

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

Harder problems to think about later:

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

   For example, given the array

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

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

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

   For example, given the array

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

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

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

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

Describe and analyze ***dynamic programming*** algorithms for the following longest-subsequence problems. Use the recurrences you developed on Wednesday.

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

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

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

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

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

Basic steps in developing a dynamic programming algorithm

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

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

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

   (c) **Don't optimize prematurely.** It may be tempting to ignore "obviously" suboptimal choices, because that will yield an "obviously" faster algorithm, but it's usually a bad idea, for two reasons. First, the optimization may not actually improve the running time of the final dynamic programming algorithm. But more importantly, many "obvious" optimizations are actually incorrect! ***First* make it work; *then* optimize.**

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

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

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

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

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

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

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

3. **Try to improve.** What's the bottleneck in your algorithm? Can you find a faster algorithm by modifying the recurrence? Can you tighten the time analysis? *Now* is the time to think about removing "obviously" redundant or suboptimal choices. (But always make sure that your optimizations are correct!!)

Nancy Gunter, the founding dean of the new Parisa Tabriz School of Computer Science, has commissioned a series of snow ramps on the south slope of the Orchard Downs sledding hill[1] and challenged Erhan Hajek, head of the Department of Electrical and Computer Engineering, to a sledding contest. Erhan and Nancy will both sled down the hill, each trying to maximize their air time. The winner gets to expand their department/school into Siebel Center, the ECE Building, *and* the new Campus Instructional Facility; the loser has to move their entire department/school under the Boneyard bridge behind Everitt Lab.

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

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

   Describe and analyze an algorithm to determine the maximum *total* distance that Erhan or Nancy can travel through the air.

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

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

Harder problem to think about later:

3. When the lawyers realized that imposing their restriction didn't immediately shut down the contest, they added yet another restriction: No ramp may be used more than once! Disgusted by all the legal interference, Erhan and Nancy give up on their bet and decide to cooperate to put on a good show for the spectators.

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

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

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

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

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

Harder problem to think about later:

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

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

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

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

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

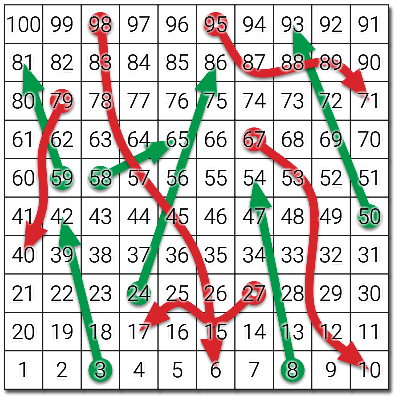
For each of the problems below, transform the input into a graph and apply a standard graph algorithm that you've seen in class. Whenever you use a standard graph algorithm, you ***must*** provide the following information. (I recommend actually using a bulleted list.)

- What are the vertices? What does each vertex represent?
- What are the edges? Are they directed or undirected?
- If the vertices and/or edges have associated values, what are they?
- What problem do you need to solve on this graph?
- What standard algorithm are you using to solve that problem?
- What is the running time of your entire algorithm, *including* the time to build the graph, *as a function of the original input parameters*?

Finally, it is crucial to remember that even when you are explicitly given a graph as part of the input, that may not be the graph you actually want to search!

---

1. ***Snakes and Ladders*** is a classic board game, which emerged in India in many different variants around the 13th century. The board consists of an $n \times n$ grid of squares, numbered consecutively from 1 to n^2, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares, always in different rows, are connected by either "snakes" (leading down) or "ladders" (leading up). *Each square can be an endpoint of at most one snake or ladder.*



100	99	98	97	96	95	94	93	92	91
81	82	83	84	85	86	87	88	89	90
80	79	78	77	76	75	74	73	72	71
61	62	63	64	65	66	67	68	69	70
60	59	58	57	56	55	54	53	52	51
41	42	43	44	45	46	47	48	49	50
40	39	38	37	36	35	34	33	32	31
21	22	23	24	25	26	27	28	29	30
20	19	18	17	16	15	14	13	12	11
1	2	3	4	5	6	7	8	9	10

A typical Snakes and Ladders board.
Upward straight arrows are ladders; downward wavy arrows are snakes.

You start with a token in cell 1, in the bottom left corner. In each move, you advance your token up to k positions, for some fixed constant k (typically 6). Then if the token is at the *top* of a snake, you *must* slide the token down to the bottom of that snake, and if the token is at the *bottom* of a ladder, you *may* move the token up to the top of that ladder.

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

2. Let G be an undirected graph. Suppose we start with two coins on two arbitrarily chosen vertices of G. At every step, each coin ***must*** move to an adjacent vertex. Describe and analyze an efficient algorithm to compute the minimum number of steps to reach a configuration where both coins are on the same vertex, or to report correctly that no such configuration is reachable. The input to your algorithm consists of a graph $G = (V, E)$ and two vertices $u, v \in V$ (which may or may not be distinct).

Harder problem to think about later:

3. Let G be an undirected graph. Suppose we start with 374 coins on 374 arbitrarily chosen vertices of G. At every step, each coin ***must*** move to an adjacent vertex. Describe and analyze an efficient algorithm to compute the minimum number of steps to reach a configuration where all 374 coins are on the same vertex, or to report correctly that no such configuration is reachable. The input to your algorithm consists of a graph $G = (V, E)$ and starting vertices $s_1, s_2, \ldots, s_{374}$ (which may or may not be distinct).

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

- What are the vertices? What does each vertex represent?
- What are the edges? Are they directed or undirected?
- If the vertices and/or edges have associated values, what are they?
- What problem do you need to solve on this graph?
- What standard algorithm are you using to solve that problem?
- What is the running time of your entire algorithm, *including* the time to build the graph, *as a function of the original input parameters*?

Finally, it is crucial to remember that even when you are explicitly given a graph as part of the input, that may not be the graph you actually want to search!

---

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

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

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

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

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

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

Harder problems to think about later:

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

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

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

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

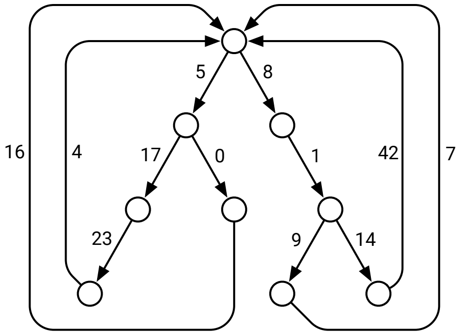
2. You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cities that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving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

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

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

Harder problems to think about later:

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



A looped tree.

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

   (b) Describe and analyze a faster algorithm.

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

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

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

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

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

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

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

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

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

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

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

[1] No, you can't.

1. Suppose you are given an array of numbers, some of which are marked as *icky*, and you want to compute the length of the longest increasing subsequence of A that includes at most k icky numbers. Your input consists of the integer k, the number array $A[1..n]$, and another boolean array $Icky[1..n]$.

   For example, suppose your input consists of the integer $k = 2$ and the following array (with icky numbers are indicated by stars):

$3^\star$	$1^\star$	4	$1^\star$	$5^\star$	9	$2^\star$	6	5	$3^\star$	5	9	7	$9^\star$	3	2	3	$8^\star$	4	$6^\star$	2	$6^\star$

   Then your algorithm should return the integer 5, which is the length of the increasing subsequence $4, 5^\star, 6, 7, 9^\star$.

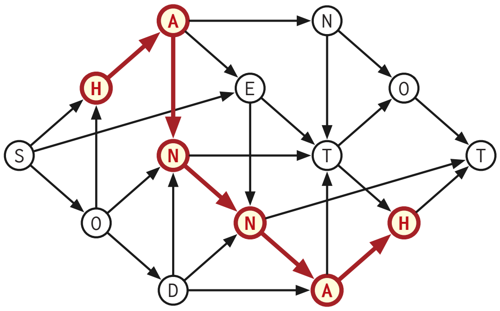
   (a) Describe an algorithm for this problem using dynamic programming.

   (b) Describe an algorithm for this problem by reducing it to a standard graph problem.

Harder problem to think about later:

2. Let G be a directed acyclic graph whose vertices have labels from some fixed alphabet. Any directed path in G has a label, which is a string obtained by concatenating the labels of its vertices. Recall that a *palindrome* is a string that is equal to its reversal.

   Describe and analyze an algorithm to find the length of the longest palindrome that is the label of a path in G. For example, given the dag below, your algorithm should return the integer 6, which is the length of the palindrome HANNAH.



   (a) Describe an algorithm for this problem using dynamic programming.

   (b) Describe an algorithm for this problem by reducing it to a standard graph problem.

1. Let $G = (V, E)$ be a graph. A set of edges $M \subseteq E$ is said to be a matching if no two edges in M intersect at a vertex. A matching M is perfect if every vertex in V is incident to some edge in M; alternatively M is perfect if $|M| = |V|/2$ (which in particular implies $|V|$ is even).

   The PERFECTMATCHING problem is the following: does the given graph G have a perfect matching?

   This can be solved in polynomial time which is a fundamental result in combinatorial optimization with many applications in theory and practice. It turns out that the PERFECTMATCHING problem is easier to solve in bipartite graphs. A graph $G = (V, E)$ is bipartite if its vertex set V can be partitioned into two sets L, R (left and right say) such that all edges are between L and R (in other words L and R are independent sets). Here is an attempted reduction from general graphs to bipartite graphs.

   Given a graph $G = (V, E)$ create a bipartite graph $H = (V \times \{1, 2\}, E_H)$ as follows. Each vertex u is made into two copies $(u, 1)$ and $(u, 2)$ with $V_1 = \{(u, 1) | u \in V\}$ as one side and $V_2 = \{(u, 2) | u \in V\}$ as the other side. Let $E_H = \{((u, 1), (v, 2)) | (u, v) \in E\}$. In other words we add an edge betwen $(u, 1)$ and $(v, 2)$ iff (u, v) is an edge in E. Note that $((u, 1), (u, 2))$ is not an edge in H for any $u \in V$ since there are no self-loops in G. Is the preceding reduction correct? To prove it is correct we need to check that H has a perfect matching if and only if G has one.

   (a) Prove that if G has perfect matching then H has a perfect matching.

   (b) Consider G to be the complete graph on 3 vertices (a triangle). Show that G has no perfect matching but H has a perfect matching.

   (c) Extend the previous example to obtain a graph G with an even number of vertices such that G has no perfect matching but H has one.

   Thus the reduction is incorrect although one of the directions is true.

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

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

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

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

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

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

To think about later:

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

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

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

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

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

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

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

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

---

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

   - Input: A boolean circuit K with n inputs and one output.
   - Output: True if there are input values $x_1, x_2, \dots, x_n \in \{\text{True}, \text{False}\}$ that make K output True, and False otherwise.

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

   - Input: A boolean circuit K with n inputs and one output.
   - Output: Input values $x_1, x_2, \dots, x_n \in \{\text{True}, \text{False}\}$ that make K output True, or None if there are no such inputs.

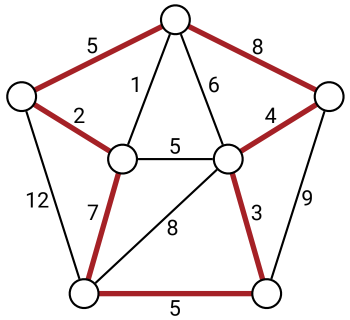
   [Hint: You can use the magic box more than once.]

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

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

To think about later:

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



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

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

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

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

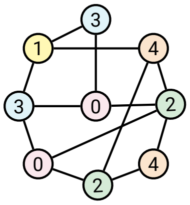
2. Prove that each of the following problems is NP-hard.

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

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

   (c) Given an undirected graph G, does G have a spanning tree with at most 374 leaves?

1. Recall that a 5-coloring of a graph G is a function that assigns each vertex of G a "color" from the set $\{0, 1, 2, 3, 4\}$, such that for any edge uv, vertices u and v are assigned different "colors". A 5-coloring is ***careful*** if the colors assigned to adjacent vertices are not only distinct, but differ by more than 1 (mod 5). Prove that deciding whether a given graph has a careful 5-coloring is NP-hard. *[Hint: Reduce from the standard 5Color problem.]*



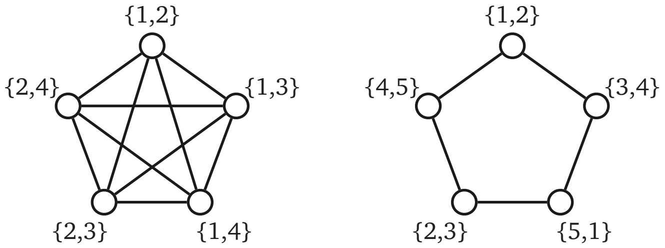
A careful 5-coloring.

2. Prove that the following problem is NP-hard: Given an undirected graph G, find *any* integer $k > 374$ such that G has a proper coloring with k colors but G does not have a proper coloring with $k - 374$ colors.

3. A ***bicoloring*** of an undirected graph assigns each vertex a set of *two* colors. There are two types of bicoloring: In a *weak* bicoloring, the endpoints of each edge must use *different* sets of colors; however, these two sets may share one color. In a *strong* bicoloring, the endpoints of each edge must use *distinct* sets of colors; that is, they must use four colors altogether. Every strong bicoloring is also a weak bicoloring.

   (a) Prove that finding the minimum number of colors in a weak bicoloring of a given graph is NP-hard.

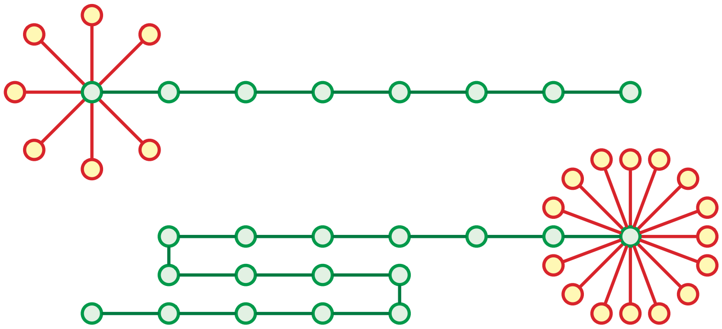
   (b) Prove that finding the minimum number of colors in a strong bicoloring of a given graph is NP-hard.



Left: A weak bicoloring of a 5-clique with four colors.
Right A strong bicoloring of a 5-cycle with five colors.

1. Balanced 3Color: Suppose we are given a graph G with $3n$ vertices, for some integer n. Prove that it is NP-hard to decide whether it is possible to color each vertex of G with three colors, so that no edge connects two vertices of the same color, and there are exactly n vertices of each color.

2. Longest Dandelion: A *dandelion of length ℓ* consists of a path of length ℓ, with exactly ℓ new edges attached to one end. Prove that it is NP-hard to find the longest dandelion subgraph of a given undirected graph.



Two dandelions, one of length 7 and the other of length 15.

3. High-Degree Independent Set: Suppose we are given a graph G and an integer k. Prove that it is NP-hard to decide whether G contains an independent set of k vertices, each of which has degree at least k.

   *[Hint: Reduce from the **decision** version of the IndependentSet problem: Given a graph G and an integer k, does G contain an independent set of size k?]*

4. Half-Clique: Suppose we are given a graph G with $2n$ vertices, for some integer n. Prove that it is NP-hard to decide whether G contains a complete subgraph with n vertices?

   *[Hint: Reduce from the **decision** version of the Clique problem: Given a graph G and an integer k, does G contain a clique of size k?]*

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

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

The language $\textsc{AcceptIn}(\mathcal{L}) := \big\{\langle M\rangle \bigm| \textsc{Accept}(M) \in \mathcal{L}\big\}$ *is undecidable.*

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

1. $\textsc{AcceptRegular} := \big\{\langle M\rangle \bigm| \textsc{Accept}(M) \text{ is regular}\big\}$
2. $\textsc{AcceptIllini} := \big\{\langle M\rangle \bigm| M \text{ accepts the string } \texttt{ILLINI}\big\}$
3. $\textsc{AcceptPalindrome} := \big\{\langle M\rangle \bigm| M \text{ accepts at least one palindrome}\big\}$
4. $\textsc{AcceptThree} := \big\{\langle M\rangle \bigm| M \text{ accepts exactly three strings}\big\}$
5. $\textsc{AcceptUndecidable} := \big\{\langle M\rangle \bigm| \textsc{Accept}(M) \text{ is undecidable } \big\}$

To think about later. Which of the following are undecidable? How would you prove that?

1. $\textsc{Accept}\{\{\varepsilon\}\} := \big\{\langle M\rangle \bigm| M \text{ accepts only the string } \varepsilon\text{; that is, } \textsc{Accept}(M) = \{\varepsilon\}\big\}$
2. $\textsc{Accept}\{\varnothing\} := \big\{\langle M\rangle \bigm| M \text{ does not accept any strings; that is, } \textsc{Accept}(M) = \varnothing\big\}$
3. $\textsc{Accept=Reject} := \big\{\langle M\rangle \bigm| \textsc{Accept}(M) = \textsc{Reject}(M)\ \big\}$
4. $\textsc{Accept}{\neq}\textsc{Reject} := \big\{\langle M\rangle \bigm| \textsc{Accept}(M) \neq \textsc{Reject}(M)\ \big\}$
5. $\textsc{Accept}{\cup}\textsc{Reject} := \big\{\langle M\rangle \bigm| \textsc{Accept}(M) \cup \textsc{Reject}(M) = \Sigma^*\big\}$

CS/ECE 374 A ✧ Fall 2021

Homework 1

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

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

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

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

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

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

See the course web site for more information.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(b) Prove $\mathit{evens}(\mathit{shuffle}(w,z)) = z$ 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*, *evens*, and *odds*. Do not appeal to intuition!

Rubrics

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

Standard induction rubric. For problems worth 10 points:

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

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

Solved Problems

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

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

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

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

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

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

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

- If $w = \varepsilon$, then

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

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

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

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

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

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

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

(b) Prove that $\mathit{swap}(\mathit{swap}(w)) = w$ for every string w.

Solution: Let w be an arbitrary string.

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

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

- If $w = \varepsilon$, then

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

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

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

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

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

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

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

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

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

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

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

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

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

■

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

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

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

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

Solution: Let w be an arbitrary palindrome.

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

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

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

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

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

Rubric: 4 points: standard induction rubric (scaled)

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

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

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

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

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

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

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

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

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

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

  We conclude that w is a palindrome by definition.

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

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

CS/ECE 374 A ✧ Fall 2021

Homework 2

Due Tuesday, September 7, 2021 at 8pm

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

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

---

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

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

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

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

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

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

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

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

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

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

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

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

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

Standard regular expression rubric. For problems worth 10 points:

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

Standard DFA design rubric. For problems worth 10 points:

- 2 points for an unambiguous description of a DFA, including the states set Q, the start state s, the accepting states A, and the transition function δ.
  - **Drawings:**
    - Use an arrow from nowhere to indicate s.
    - Use doubled circles to indicate accepting states A.
    - If $A = \varnothing$, 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`, `◇`, `↲`$\}$ that form a proper comment in the C program language and its descendants, like C++ and Java. Here `↲` represents the newline character, `◇` represents any other whitespace character (like the space and tab characters), and `A` represents any non-whitespace character other than `*` or `/`.[2] There are two types of C comments:

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

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

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

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

   `/*/` `//◇//◇↲◇↲` `/*◇/*◇*/◇*/`

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

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

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

For example, the string `"/*\\\"*/"/*"/*\"/*"*/` is a valid string literal (representing the 5-character string `/*\"*/`, which is itself a valid block comment!) followed immediately by a valid block comment.

For this homework question, pretend that the characters `'`, `"`, and `\` do not exist.

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

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

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

Solution:

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

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

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

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

Solution:

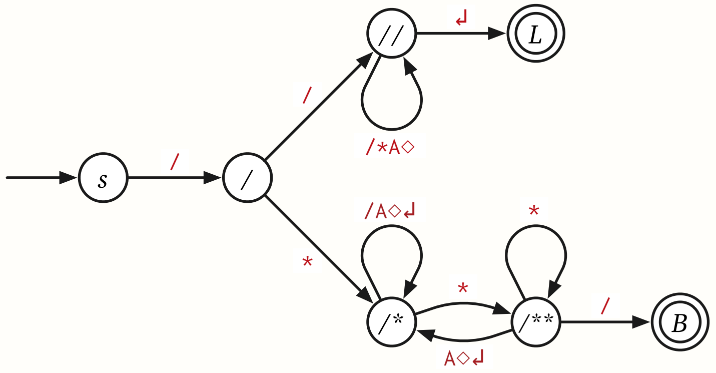
$$\big(\diamond + \text{↲} + \mathtt{//}(\mathtt{/} + \star + \mathtt{A} + \diamond)^*\text{↲} + \mathtt{/}\star(\mathtt{/} + \mathtt{A} + \diamond + \text{↲} + \star\star^*(\mathtt{A} + \diamond + \text{↲}))^*\ \star^*\star\mathtt{/}\big)^*$$

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

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

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

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



The states are labeled mnemonically as follows:

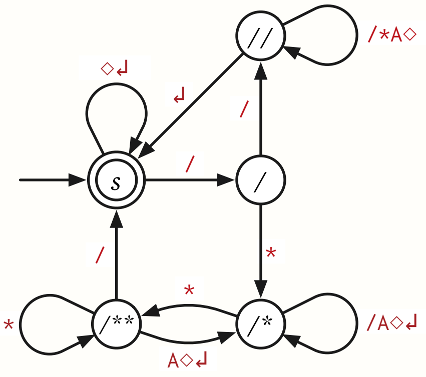
- *s* — We have not read anything.
- / — We just read the initial /.
- // — We are reading a line comment.
- *L* — We have just read a complete line comment.
- /* — We are reading a block comment, and we did not just read a * after the opening /*.
- /** — We are reading a block comment, and we just read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

■

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

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

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



The states are labeled mnemonically as follows:

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

■

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

CS/ECE 374 A ✧ Fall 2021

Homework 3

Due Tuesday, September 14, 2021 at 8pm

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

   (a) $\{0^m1^n \mid m \text{ and } n \text{ are relatively prime}\}$

   (b) $\{w \in (0+1)^* \mid 10^n1^n \text{ for } n > 0 \text{ is a suffix of } w\}$

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

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

   (a) $\{0^a1w10^c \mid w \in \Sigma^*, (a \le |w| + c) \text{ and } (|w| \le a + c \text{ or } c \le a + |w|)\}$

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

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

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

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

Solved problem

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

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

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

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

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

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

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

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

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

   Solution: Not regular. Call this language L_b.

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

   Let x and y be arbitrary distinct strings in F.

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

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

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

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

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

   Because F is infinite, L_b cannot be regular. ■

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

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

Solution: Not regular. Call this language L_c.

I claim that the infinite language $F = (\mathtt{001101})^+$ is a fooling set for L_c.

Let x and y be arbitrary distinct strings in F.

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

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

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

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

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

Because F is infinite, L_c cannot be regular. ■

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

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

Solution: Regular. Call this language L_d.

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

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

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

Solution: Regular. Call this language L_d.

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

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

■

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

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

Standard fooling set rubric. For problems worth 5 points:

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

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

CS/ECE 374 A ✧ Fall 2021

Homework 4

Due Tuesday, September 21, 2021 at 8pm

This is the last homework before Midterm 1.

---

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

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

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

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

   (see next page for Question 2)

2. Let L be any regular language over the alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$. 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, $\text{thirds}(\texttt{011000110}) = \texttt{100}$.
   (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 langage transformation rubric. For problems worth 10 points:

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

Solved problem

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

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

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

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

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

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

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

■

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

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

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

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

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

Let x and y be arbitrary distinct strings in L.

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

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

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

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

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

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

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

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

Let x and y be arbitrary distinct strings in F.

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

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

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

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

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

Rubric: 5 points:

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

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

Standard langage transformation rubric. For problems worth 10 points:

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

CS/ECE 374 A ✧ Fall 2021

Homework 5

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

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

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

```
UNUSUAL($A[1..n]$):
    if $n = 2$
        if $A[1] > A[2]$                      ⟨⟨the only comparison!⟩⟩
            swap $A[1] \longleftrightarrow A[2]$
    else
        for $i \leftarrow 1$ to $n/4$                 ⟨⟨swap 2nd and 3rd quarters⟩⟩
            swap $A[i+n/4] \longleftrightarrow A[i+n/2]$
        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 $A[i].\mathit{birth}$ and $A[i].\mathit{death}$ and a string field $A[i].\mathit{name}$. The desired output is the maximum, over all indices $i \neq j$, of the overlap length

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

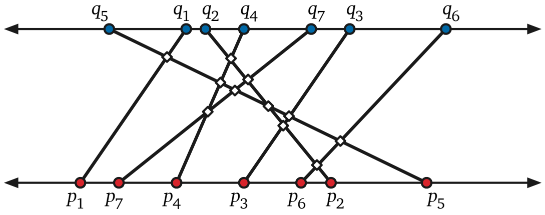
Describe and analyze an efficient algorithm to solve Dakshita's problem.

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



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

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

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

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

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

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

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

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

Rubric: This is enough for full credit.

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

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

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

```
MergeAndCount2($A[1\,..\,n], m$):
  $i \gets 1$;  $j \gets m+1$;  $total \gets 0$
  for $k \gets 1$ to $n$
      if $j > n$
          $B[k] \gets A[i]$;  $i \gets i+1$;  $total \gets total + \boldsymbol{j - m - 1}$
      else if $i > m$
          $B[k] \gets A[j]$;  $j \gets j+1$
      else if $A[i] < A[j]$
          $B[k] \gets A[i]$;  $i \gets i+1$;  $total \gets total + \boldsymbol{j - m - 1}$
      else
          $B[k] \gets A[j]$;  $j \gets j+1$
  for $k \gets 1$ to $n$
      $A[k] \gets B[k]$
  return $total$
```

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

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

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

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

CS/ECE 374 A ✧ Fall 2021

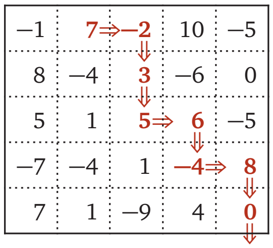
Homework 6

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

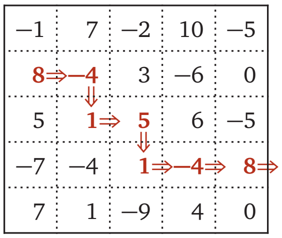
---

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

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



−1	**7**⇒	**−2** ⇓	10	−5
8	−4	**3** ⇓	−6	0
5	1	**5**⇒	**6** ⇓	−5
−7	−4	1	**−4**⇒	**8** ⇓
7	1	−9	4	**0** ⇓



−1	7	−2	10	−5
8⇒	**−4** ⇓	3	−6	0
5	**1**⇒	**5** ⇓	6	−5
−7	−4	**1**⇒	**−4**⇒	**8**⇒
7	1	−9	4	0

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

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

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

   I am the fire demon, moving castles: Calcifer!

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

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

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

   For example, given the input string

   EVENIFYOUAMTHEAREMYFIRELEASTDEMONFAVORITEMOVINGCASTLESVEGETABLECALCIFER

   your algorithm should return the integer 8:

   EVENIFYOUAMTHEAREMYFIRELEASTDEMONFAVORITEMOVINGCASTLESVEGETABLECALCIFER

Standard dynamic programming rubric. For problems worth 10 points:

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

  −1 for naming the function "OPT" or "DP" or any single letter.

- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.

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

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

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

  - **No credit for the rest of the problem if the recursive case(s) are incorrect.**

- 3 points for iterative details

  + 1 for describing an appropriate memoization data structure

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

  + 1 for correct time analysis. (It is not necessary to state a space bound.)

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

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

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

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

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

Solved Problem

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

   `BANANAANANAS` `BANANAANANAS` `BANANAANANAS`

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

   `PRODGYRNAMAMMIINCG` `DYPRONGARMAMMICING`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

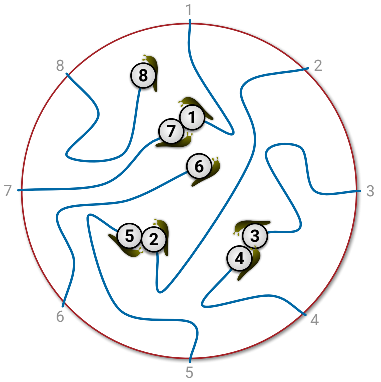
Rubric: 5 points, standard dynamic programming rubric. 3 points for a slower polynomial-time algorithm; scale partial credit accordingly.

CS/ECE 374 A ✧ Fall 2021

Homework 7

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

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



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

For every pair of snails, the Antarctican SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array $M[1\,..\,n, 1\,..\,n]$ posted on the wall behind the Round Table, where $M[i,j] = M[j,i]$ is the reward to be paid if snails i and j 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* ε-transitions and a binary string $w \in \{0,1\}^*$. Describe and analyze an efficient algorithm to determine whether M accepts w. Concretely, the input NFA M is represented as follows:

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

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

Solved Problems

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

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

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

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

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

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

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

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

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

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

■

Rubric: 10 points, standard dynamic programming rubric

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.

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

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

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

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

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

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

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

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

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

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

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

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

The function *MaxFun* obeys the following recurrence:

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

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

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

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

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

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

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

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

CS/ECE 374 A ✧ Fall 2021

Homework 8

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

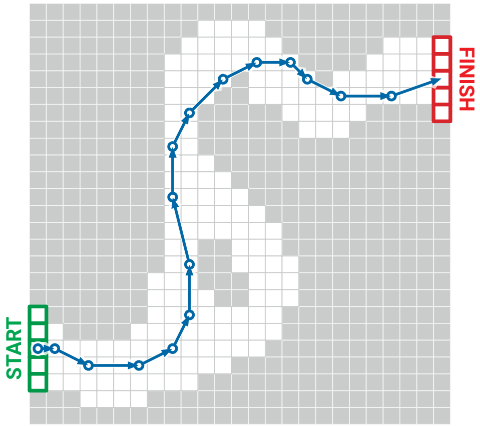
1. ***Racetrack*** (also known by several other names, including *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.[1] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid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

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

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

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

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



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

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

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

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

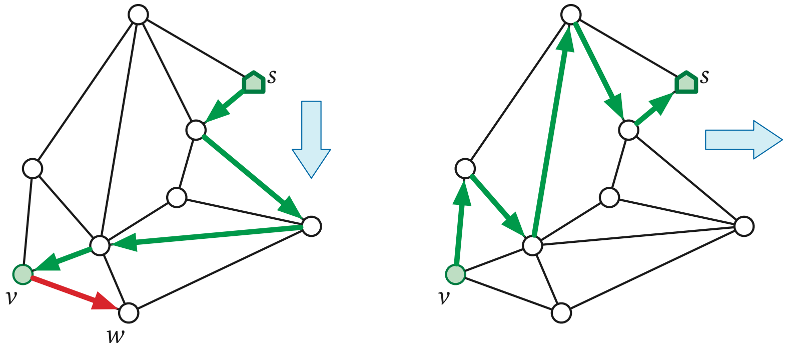
   Jeff has encoded a map of all bike-safe roads in Champaign-Urbana into an undirected graph $G = (V, E)$ whose vertices represent intersections and sharp corners, and whose edges represent straight road segments. Jeff's home is represented in G by a special vertex s. Every edge of G is labeled with its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

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

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

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

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



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

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

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

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

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

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

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

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

Rubrics

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

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

Solved Problem

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

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

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

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

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

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

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

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

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

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

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

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

Rubric: 10 points: standard graph reduction rubric

- Brute force construction is fine.

—1 for calling Dijkstra instead of BFS

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

CS/ECE 374 A Fall 2021

Homework 9

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

This is the last homework before Midterm 2.

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

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

   Describe and analyze an algo(burp)rithm to so(burp)olve Morty's problem. You can assume that it is in fact possible for Morty to succeed. As usual, do not assume that edge weights are integers.

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

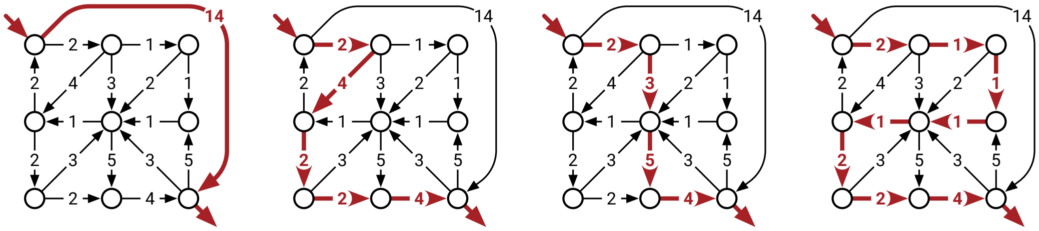
   You have converted the trail map into a directed graph $G = (V, E)$, whose vertices represent trail intersections, and whose edges represent trail segments. A subset $R \subseteq V$ of the vertices indicate the locations of the *R*epellent *R*efilling stations, and a subset $H \subseteq E$ of the edges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

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

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

Solved Problem

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



Four (of many) equal-length shortest paths.

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

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

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

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

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

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

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

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

The overall running time of the algorithm is dominated by Dijkstra’s algorithm in the preprocessing phase, which runs in ***O(E* log *V)* time**. ■

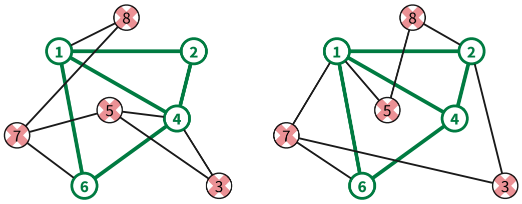
Rubric: 10 points = 5 points for reduction to counting paths in a dag (standard graph reduction rubric) + 5 points for the path-counting algorithm (standard dynamic programming rubric)

CS/ECE 374 A ✧ Fall 2021

Homework 10

Due Tuesday, November 16, 2021 at 8pm

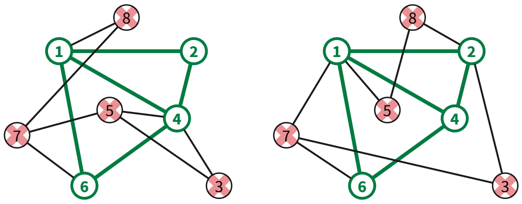
1. Suppose we are given two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ with the same set of vertices $V = \{1, 2, \ldots, n\}$. You are given the following problem: find the smallest subset $S \subseteq V$ of vertices whose deletion leaves identical subgraphs $G_1 \setminus S = G_2 \setminus S$. For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



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

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

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



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

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

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

   (b) The input to **MAJORITY3SAT** is a boolean formula Φ in conjunctive normal form, with exactly three literals per clause. MAJORITY3SAT asks whether there is an assignment to the variables of Φ such that every clause contains *at least two* TRUE literals.

   Describe and analyze a polynomial-time reduction from MAJORITY3SAT to 2SAT. Don't forget to prove that your reduction is correct.

   (c) Combining parts (a) and (b) gives us an algorithm for MAJORITY3SAT. What is the running time of this algorithm?

Solved Problem

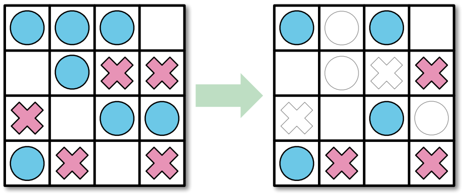
3. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions:

   (1) Every row contains at least one stone.

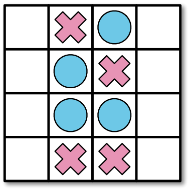
   (2) No column contains stones of both colors.

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

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



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

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

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

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

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

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

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

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

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

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

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

This reduction clearly requires only polynomial time. ■

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

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

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

CS/ECE 374 A ✧ Fall 2021

Homework 11

Due Tuesday, November 30, 2021 at 8pm

---

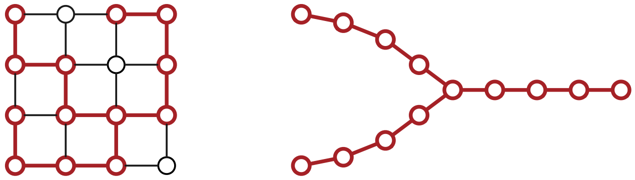
1. (a) A **quasi-satisfying assignment** for a 3CNF boolean formula Φ is an assignment of truth values to the variables such that at most one clause in Φ does not contain a true literal.

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

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

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

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

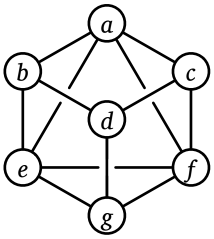


This grid graph contains a wye whose paths have length 4.

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

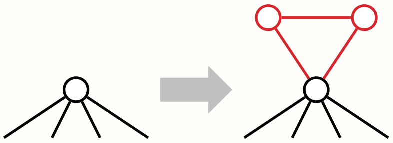
Solved Problem

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



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

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



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

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

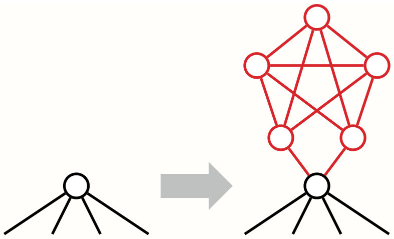
$\Longrightarrow$ Suppose G has a Hamiltonian cycle $v_1 \to v_2 \to \cdots \to v_n \to v_1$. We can construct a double-Hamiltonian tour of H by replacing each vertex v_i with the following walk:

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

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

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

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

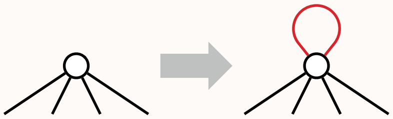


A vertex in G, and the corresponding modified vertex gadget in H.

■

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

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

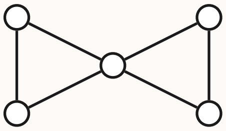


An incorrect vertex gadget.

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

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

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



This graph has a double-Hamiltonian tour.

♣

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $\max\{x \in \mathbb{Z}^d \mid Ax \le b, x \ge 0\}$ is empty.

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

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

CS/ECE 374 A ✧ Fall 2021

“Homework” 12

“Due” Monday, December 6, 2021

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

---

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

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

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

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

Prove that the following language is undecidable:

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

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

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

3. Prove that the following language is undecidable:

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

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

CS/ECE 374 A ✧ Fall 2021

Midterm 1 Study Questions

This is a "core dump" of potential questions for Midterm 1. This should give you a good idea of the *types* of questions that we will ask on the exam—in particular, there *will* be a series of True/False questions—but the actual exam questions may or may not appear in this handout. This list intentionally includes a few questions that are too long or difficult for exam conditions; most of these are indicated with a *star.

Questions from Jeff's past exams are labeled with the semester they were used—⟨⟨***S14***⟩⟩ or ⟨⟨***F19***⟩⟩, for example. Questions from this semester's homework (either written or on PrairieLearn) are labeled ⟨⟨***HW***⟩⟩. Questions from this semester's labs are labeled ⟨⟨***Lab***⟩⟩. Some unflagged questions may have been used in exams by other instructors.

How to Use These Problems

Solving every problem in this handout is ***not*** the best way to study for the exam. Memorizing the solutions to every problem in this handout is the ***absolute worst*** way to study for the exam.

What we recommend instead is to work on a *sample* of the problems. Choose one or two problems at random from each section and try to solve them from scratch under exam conditions—by yourself, in a quiet room, with a 30-minute timer, *without* your notes, *without* the internet, and if possible, even without your cheat sheet. If you're comfortable solving a few problems in a particular section, you're probably ready for that type of problem on the exam. Move on to the next section.

Discussing problems with other people (in your study groups, in the review sessions, in office hours, or on Piazza) and/or looking up old solutions can be *extremely* helpful, but ***only after*** you have (1) made a good-faith effort to solve the problem on your own, and (2) you have either a candidate solution or some idea about where you're getting stuck.

If you find yourself getting stuck on a particular type of problem, try to figure out *why* you're stuck. Do you understand the problem statement? Are you stuck on choosing the right high-level approach, are you stuck on the technical details, or are you struggling to express your ideas clearly?

Similarly, if feedback suggests that your solutions to a particular type of problem are incorrect or incomplete, try to figure out what you missed. For induction proofs: Are you sure you have the right induction hypothesis? Are your cases obviously exhaustive? For regular expressions, DFAs, NFAs, and context-free grammars: Is your solution both exclusive and exhaustive? Did you try a few positive examples *and* a few negative examples? For fooling sets: Are you imposing enough structure? Are x and y really *arbitrary* strings from F? For language transformations: Are you transforming in the right direction? Are you using non-determinism correctly? Do you understand the formal notation for DFAs and NFAs?

Remember that your goal is *not* merely to "understand"—or worse, to *remember*—the solution to any particular problem, but to become more comfortable with solving a certain *type* of problem on your own. **"Understanding" is a seductive trap; aim for mastery.** If you can identify specific steps that you find problematic, read more *about those steps*, focus your practice *on those steps*, and try to find helpful information *about those steps* to write on your cheat sheet. Then work on the next problem!

Induction on Strings

Give complete, formal inductive proofs for the following claims. Your proofs must reply on the formal recursive definitions of the relevant string functions, not on intuition. Recall that the concatenation • and length $|\cdot|$ functions are formally defined as follows:

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

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

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

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

(a) Prove that $(w \bullet x)^R = x^R \bullet w^R$ for all strings w and x. ***⟨⟨Lab⟩⟩***

(b) Prove that $(w^R)^R = w$ for every string w. ***⟨⟨Lab⟩⟩***

(c) Prove that $|w| = |w^R|$ for every string w. ***⟨⟨Lab⟩⟩***

1.2 Let $\#(a, w)$ denote the number of times symbol a appears in string w. For example, #(X, WTF374) = 0 and #(0, 000010101010010100) = 12.

(a) Give a formal recursive definition of $\#(a, w)$.***⟨⟨Lab⟩⟩***

(b) Prove that $\#(a, w \bullet z) = \#(a, w) + \#(a, z)$ for all symbols a and all strings w and z. ***⟨⟨Lab⟩⟩***

(c) Prove that $\#(a, w^R) = \#(a, w)$ for all symbols a and all strings w, where w^r denotes the reversal of w. ***⟨⟨Lab⟩⟩***

1.3 For any string w and any non-negative integer n, let $\boldsymbol{w^n}$ denote the string obtained by concatenating n copies of w; more formally, define

$$w^n := \begin{cases} \varepsilon & \text{if } n = 0 \\ w \bullet w^{n-1} & \text{otherwise} \end{cases}$$

For example, (BLAH)^5 = BLAHBLAHBLAHBLAHBLAH and $\varepsilon^{374} = \varepsilon$.

(a) Prove that $w^m \bullet w^n = w^{m+n}$ for every string w and all non-negative integers n and m.

(b) Prove that $(w^m)^n = w^{mn}$ for every string w and all non-negative integers n and m.

(c) Prove that $|w^n| = n|w|$ for every string w and every integer $n \geq 0$.

(d) Prove that $(w^n)^R = (w^R)^n$ for every string w and every integer $n \geq 0$.

1.4 Consider the following pair of mutually recursive functions:

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

For example, $evens(\texttt{0001101}) = \texttt{010}$ and $odds(\texttt{0001101}) = \texttt{0011}$.

(a) Prove the following identity for all strings w and x: ⟨⟨***HW***⟩⟩

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

(b) State and prove a similar identity for $odds(w \bullet x)$.

(c) Prove the following identity for all strings w:

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

(d) Prove that $|w| = |evens(w)| + |odds(w)|$ for every string w.

1.5 The ***complement*** $\boldsymbol{w^c}$ of a string $w \in \{\texttt{0}, \texttt{1}\}^*$ is obtained from w by replacing every $\texttt{0}$ in w with a $\texttt{1}$ and vice versa. The complement function can be defined recursively as follows:

$$w^c := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \texttt{1} \cdot x^c & \text{if } w = \texttt{0}x \\ \texttt{0} \cdot x^c & \text{if } w = \texttt{1}x \end{cases}$$

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

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

(c) Prove that $\#(\texttt{1}, w) = \#(\texttt{0}, w^c)$ for every string w.

1.6 Consider the following recursively defined function:

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

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

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

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

(c) Prove that $odds(stutter(w)) = w$ for every string w.

(d) Prove that w is a palindrome if and only if $stutter(w)$ is a palindrome, for every string w.

1.7 Consider the following recursive function:

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

For example, $shuffle(\texttt{0011},\texttt{0101}) = \texttt{00011011}$.

(a) Prove that $|shuffle(x,y)| = |x| + |y|$ for all strings x and y.

(b) Prove that $shuffle(w,w) = stutter(w)$ for every string w.

(c) Prove that $shuffle(odds(w), evens(w)) = w$ for every string w. ⟨⟨*HW*⟩⟩

(d) Prove that $evens(shuffle(w,z)) = z$ for all strings w and z such that $|w| = |z|$. ⟨⟨*HW*⟩⟩

Regular expressions

For each of the following languages over the alphabet $\Sigma = \{0, 1\}$, give an equivalent regular expression, and *briefly* argue why your expression is correct. (On exams, we will not ask for justifications, but you should still justify your expressions in your head.)

2.1 Every string of length at most 3. *[Hint: Don't try to be clever.]*

2.2 All strings except 010.

2.3 All strings that end with the suffix 010.

2.4 All strings that do not start with the prefix 010.

2.5 All strings that contain the substring 010.

2.6 All strings that do not contain the substring 010.

2.7 All strings that contain the subsequence 010.

2.8 All strings that do not contain the subsequence 010.

2.9 All strings containing the substring 10 or the substring 01.

2.10 All strings containing either the substring 10 or the substring 01, but not both. ⟨⟨***F16***⟩⟩

2.11 All strings that do not contain either 001 or 110 as a substring. ⟨⟨***F19***⟩⟩

2.12 All strings containing the subsequence 10 or the subsequence 01 (or possibly both).

2.13 All strings containing the subsequence 10 or the subsequence 01, but not both.

2.14 All strings containing at least two 1s and at least one 0. ⟨⟨***Lab***⟩⟩

2.15 All strings containing at least two 1s or at least one 0 (or possibly both).

2.16 All strings containing at least two 1s or at least one 0, but not both.

2.17 All strings in which every run of consecutive 0s has even length. ⟨⟨***S21***⟩⟩

2.18 All strings in which every run of consecutive 0s has even length and every run of consecutive 1s has odd length. ⟨⟨***F14***⟩⟩

2.19 All strings whose length is divisible by 3.

2.20 All strings in which the number of 1s is divisible by 3.

2.21 All strings in 0^*1^* whose length is divisible by 3. ⟨⟨***S14***⟩⟩

2.22 All strings in 0^*10^* whose length is divisible by 3. ⟨⟨***S18***⟩⟩

2.23 All strings in $0^*1^*0^*$ whose length is even. ⟨⟨***S18***⟩⟩

2.24 $\{0^n w 1^n \mid n > 1 \text{ and } q \in \Sigma^*\}$ ⟨⟨***S18***⟩⟩

Direct DFA construction.

Draw or formally describe a DFA that recognizes each of the following languages. Don't forget to describe the states of your DFA in English. Unless otherwise specified, all languages are over the alphabet $\Sigma = \{0, 1\}$.

2.1 The language $\{\texttt{LONG}, \texttt{LUG}, \texttt{LEGO}, \texttt{LEG}, \texttt{LUG}, \texttt{LOG}, \texttt{LINGO}\}$.

2.2 The language $\texttt{MOO}^* + \texttt{MEOO}^*\texttt{W}$

2.3 Every string of length at most 3.

2.4 All strings except 010.

2.5 All strings that end with the suffix 010.

2.6 All strings that do not start with the prefix 010.

2.7 All strings that contain the substring 010.

2.8 All strings that do not contain the substring 010.

2.9 All strings that contain the subsequence 010.

2.10 All strings containing the substring 10 or the substring 01.

2.11 All strings containing either the substring 10 or the substring 01, but not both. ⟨⟨***F16***⟩⟩

2.12 All strings that do not contain either 001 or 110 as a substring. ⟨⟨***F19***⟩⟩

2.13 All strings containing the subsequence 10 or the subsequence 01 (or possibly both).

2.14 All strings containing at least two 1s and at least one 0. ⟨⟨***Lab***⟩⟩

2.15 All strings containing at least two 1s or at least one 0, but not both.

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

2.17 All strings in which every run of consecutive 0s has even length. ⟨⟨***S21***⟩⟩

2.18 All strings in which every run of consecutive 0s has even length and every run of consecutive 1s has odd length. ⟨⟨***F14***⟩⟩

2.19 All strings that end with 01 and that have odd length ⟨⟨***S21***⟩⟩

2.20 All strings in which the number of 1s is divisible by 3.

2.21 All strings that represent an integer divisible by 3 in binary.

2.22 All strings that represent an integer divisible by 5 in base 7.

2.23 All strings in 0^*1^* whose length is divisible by 3. ⟨⟨***S14***⟩⟩

2.24 All strings in 0^*10^* whose length is divisible by 3. ⟨⟨***S18***⟩⟩

2.25 All strings in $0^*1^*0^*$ whose length is even. ⟨⟨***S18***⟩⟩

2.26 $\{0^n w 1^n \mid n > 1 \text{ and } q \in \Sigma^*\}$ ⟨⟨***S18***⟩⟩

Fooling sets

Prove that each of the following languages is *not* regular. Unless specified otherwise, all languages are over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$.

4.1 All strings with more 0s than 1s. ⟨⟨***S14***⟩⟩

4.2 All strings with fewer 0s than 1s.

4.3 All strings with exactly twice as many 0s as 1s. ⟨⟨***Lab***⟩⟩

4.4 All strings with at least twice as many 0s as 1s.

4.5 $\left\{\mathtt{0}^{2^n} \mid n \geq 0\right\}$ ⟨⟨***Lab***⟩⟩

4.6 $\left\{\mathtt{0}^{3^n} \mid n \geq 0\right\}$ ⟨⟨***S21***⟩⟩

4.7 $\left\{\mathtt{0}^{F_n} \mid n \geq 0\right\}$, where F_n is the nth Fibonacci number, defined recursively as follows:

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

[Hint: If $F_i + F_j$ is a Fibonacci number, then either $i = j \pm 1$ or $\min\{i, j\} \leq 2$.]

4.8 $\left\{\mathtt{0}^{n^2} \mid n \geq 0\right\}$ ⟨⟨***Lab***⟩⟩

4.9 $\left\{\mathtt{0}^{n^3} \mid n \geq 0\right\}$

4.10 $\left\{\mathtt{0}^{2n}\mathtt{1}^n \mid n \geq 0\right\}$ ⟨⟨***Lab***⟩⟩

4.11 $\{\mathtt{0}^m\mathtt{1}^n \mid m \neq 2n\}$ ⟨⟨***Lab***⟩⟩

4.12 $\left\{\mathtt{0}^i\mathtt{1}^j\mathtt{0}^k \mid 2i = k \text{ or } i = 2k\right\}$ ⟨⟨***S18***⟩⟩

4.13 $\left\{\mathtt{0}^i\mathtt{1}^j\mathtt{0}^k \mid i + j = 2k\right\}$ ⟨⟨***F19***⟩⟩

4.14 $\left\{x\#y \mid x, y \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and } \#(\mathtt{0}, x) = \#(\mathtt{1}, y)\right\}$

4.15 $\left\{xx^c \mid x \in \{\mathtt{0}, \mathtt{1}\}^*\right\}$, where x^c is the *complement* of x, obtained by replacing every 0 in x with a 1 and vice versa. For example, $\mathtt{0001101}^c = \mathtt{1110010}$.

4.16 Properly balanced strings of parentheses, described by the context-free grammar $S \to \varepsilon \mid SS \mid (S)$. ⟨⟨***Lab***⟩⟩

4.17 Palindromes whose length is divisible by 3.

4.18 Strings in which at least two runs of consecutive 0s have the same length.

4.19 $\left\{(\mathtt{01})^n(\mathtt{10})^n \mid n \geq 0\right\}$

4.20 $\left\{(\mathtt{01})^m(\mathtt{10})^n \mid n \geq m \geq 0\right\}$

4.21 $\left\{w\#x\#y \mid w, x, y \in \Sigma^* \text{ and } w, x, y \text{ are not all equal}\right\}$

Regular or Not?

For each of the following languages, either prove that the language is regular (by describing a DFA, NFA, or regular expression), or prove that the language is not regular (using a fooling set argument). Unless otherwise specified, all languages are over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. Read the language descriptions ***very*** carefully.

5.1 The set of all strings in $\{\mathtt{0}, \mathtt{1}\}^*$ in which the substrings `01` and `10` appear the same number of times. (For example, the substrings `01` and `01` each appear three times in the string `1100001101101`.) ⟨⟨*F14*⟩⟩

5.2 The set of all strings in $\{\mathtt{0}, \mathtt{1}\}^*$ in which the substrings `00` and `11` appear the same number of times. (For example, the substrings `00` and `11` each appear three times in the string `1100001101101`.) ⟨⟨*F14*⟩⟩

5.3 $\left\{www \mid w \in \Sigma^*\right\}$ ⟨⟨*F14*⟩⟩

5.4 $\left\{wxw \mid w, x \in \Sigma^*\right\}$ ⟨⟨*F14*⟩⟩

5.5 All strings such that in every prefix, the number of `0`s is greater than the number of `1`s.

5.6 All strings such that in every *non-empty* prefix, the number of `0`s is greater than the number of `1`s.

5.7 $\left\{\mathtt{0}^m\mathtt{1}^n \mid 0 \le m - n \le 374\right\}$

5.8 $\left\{\mathtt{0}^m\mathtt{1}^n \mid 0 \le m + n \le 374\right\}$

5.9 The language generated by the following context-free grammar:

$$S \to \mathtt{0}A\mathtt{1} \mid \varepsilon$$
$$A \to \mathtt{1}S\mathtt{0} \mid \varepsilon$$

5.10 The language generated by the following free grammar $S \to \mathtt{0}S\mathtt{1} \mid \mathtt{1}S\mathtt{0} \mid \varepsilon$

5.11 $\left\{w\#x \mid w, x \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and no substring of } w \text{ is also a substring of } x\right\}$

5.12 $\left\{w\#x \mid w, x \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and no } \textit{non-empty} \text{ substring of } w \text{ is also a substring of } x\right\}$

5.13 $\left\{w\#x \mid w, x \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and } \textit{every} \text{ non-empty substring of } w \text{ is also a substring of } x\right\}$

5.14 $\left\{w\#x \mid w, x \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and } w \text{ is a substring of } x\right\}$

5.15 $\left\{w\#x \mid w, x \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and } w \text{ is a proper substring of } x\right\}$

5.16 $\{xy \mid x \text{ is a palindrome and } y \text{ is a palindrome}\}$ ⟨⟨*F19*⟩⟩

5.17 $\{xy \mid x \text{ is not a palindrome}\}$ ⟨⟨*F19*⟩⟩

5.18 $\{xy \mid x \text{ is a palindrome and } |x| > 1\}$ ⟨⟨*F19*⟩⟩

5.19 $\left\{xy \mid \#(\mathtt{0}, x) = \#(\mathtt{1}, y) \text{ and } \#(\mathtt{1}, x) = \#(\mathtt{0}, y)\right\}$

5.20 $\left\{xy \mid \#(\mathtt{0}, x) = \#(\mathtt{1}, y) \text{ or } \#(\mathtt{1}, x) = \#(\mathtt{0}, y)\right\}$

Product/Subset Constructions

For each of the following languages L over the alphabet $\{0,1\}$, formally describe a DFA $M = (Q, s, A, \delta)$ that recognizes L. ***Do not attempt to draw the DFA. Do not use the phrase "product construction".*** Instead, give a complete, precise, and self-contained description of the state set Q, the start state s, the accepting state A, and the transition function δ.

6.1 ⟨⟨*S14*⟩⟩ All strings that satisfy *all* of the following conditions:

(a) the number of 0s is even

(b) the number of 1s is divisible by 3

(c) the total length is divisible by 5

6.2 All strings that satisfy *at least one* of the following conditions: ...

6.3 All strings that satisfy *exactly one* of the following conditions: ...

6.4 All strings that satisfy *exactly two* of the following conditions: ...

6.5 All strings that satisfy *an odd number of* of the following conditions: ...

- Other possible conditions:

(a) The number of 0s in w is odd.

(b) The number of 1s in w is not divisible by 5.

(c) The length $|w|$ is divisible by 7.

(d) The binary value of w is divisible by 7.

(e) w represents a number divisible by 5 in base 7.

(f) w contains the substring 00

(g) w does not contain the substring 11

(h) ww does not contain the substring 101

Regular Language Transformations

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

7.1 All strings in L whose length is divisible by 3.

7.2 $\textsc{OneInFront}(L) := \{\mathtt{1}x \mid x \in L\}$

7.3 $\textsc{OnlyOnes}(L) := \left\{\mathtt{1}^{\#(\mathtt{1},w)} \,\middle|\, w \in L\right\}$

7.4 $\textsc{OnlyOnes}^{-1}(L) := \left\{w \,\middle|\, \mathtt{1}^{\#(\mathtt{1},w)} \in L\right\}$

7.5 $\textsc{MissingFirstOne}(L) := \{w \in \Sigma^* \mid \mathtt{1}w \in L\}$

7.6 $\textsc{MissingOneOne}(L) := \{xy \mid x\mathtt{1}y \in L\}$

7.7 $\textsc{Prefixes}(L) := \{x \mid xy \in L \text{ for some } y \in \Sigma^*\}$

7.8 $\textsc{Suffixes}(L) := \{y \mid xy \in L \text{ for some } x \in \Sigma^*\}$ ⟨⟨***F16***⟩⟩

7.9 $\textsc{Evens}(L) := \{\mathit{evens}(w) \mid w \in L\}$, where the functions *evens* and *odds* are recursively defined as follows:

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

For example, $\mathit{evens}(\mathtt{0001101}) = \mathtt{010}$ and $\mathit{odds}(\mathtt{0001101}) = \mathtt{0011}$. ⟨⟨***F14***⟩⟩

7.10 $\textsc{Evens}^{-1}(L) := \{w \mid \mathit{evens}(w) \in L\}$, where the functions *evens* and *odds* are recursively defined as above. ⟨⟨***F14***⟩⟩

7.11 $\textsc{AddParity}(L) = \{\mathit{addparity}(w) \mid w \in L\}$, where ⟨⟨***S18***⟩⟩

$$\mathit{addparity}(w) = \begin{cases} \mathtt{0}w & \text{if } \#(\mathtt{1}, w) \text{ is even} \\ \mathtt{1}w & \text{if } \#(\mathtt{1}, w) \text{ is odd} \end{cases}$$

7.12 $\textsc{StripFinal}\mathtt{0}\textsc{s}(L) = \{w \mid w\mathtt{0}^n \in L \text{ for some } n \ge 0\}$. Less formally, $\textsc{StripFinal}\mathtt{0}\textsc{s}(L)$ is the set of all strings obtained by removing any number of final $\mathtt{0}$s from strings in L. ⟨⟨***S18***⟩⟩

7.13 $\textsc{Obliviate}(L) := \{\mathit{obliviate}(w) \mid w \in L\}$, where $\mathit{obliviate}(w)$ is the string obtained from w by deleting every $\mathtt{1}$. ⟨⟨***F19***⟩⟩

7.14 $\textsc{UnObliviate}(L) := \{w \in \Sigma^* \mid \mathit{obliviate}(w) \in L\}$, where $\mathit{obliviate}(w)$ is the string obtained from w by deleting every $\mathtt{1}$. ⟨⟨***F19***⟩⟩

7.15 $\textsc{SameSlash}(w) = \{\mathit{sameslash}(w) \mid w \in L\}$, where $\mathit{sameslash}(w)$ is the string in $\{\mathtt{0}, \mathtt{1}, \mathtt{/}\}$ obtained from w by inserting a new symbol $\mathtt{/}$ between any two consecutive appearances of the same symbol. ⟨⟨***F19***⟩⟩

7.16 $\textsc{DiffSlash}(w) = \{\mathit{diffslash}(w) \mid w \in L\}$, where $\mathit{diffslash}(w)$ is the string in $\{\mathtt{0}, \mathtt{1}, \mathtt{/}\}$ obtained from w by inserting a new symbol $\mathtt{/}$ between any two consecutive symbols that are ***not*** equal. ⟨⟨***F19***⟩⟩

Context-Free Grammars

Construct context-free grammars for each of the following languages, and give a *brief* explanation of how your grammar works, including the language of each non-terminal. Unless specified otherwise, all languages are over the alphabet $\{\mathtt{0},\mathtt{1}\}$. We explicitly do ***not*** want a formal proof of correctness.

8.1 All strings in $\{\mathtt{0},\mathtt{1}\}^*$ whose length is divisible by 5.

8.2 All strings in which the substrings `01` and `01` appear the same number of times.

8.3 $\{\mathtt{0}^n\mathtt{1}^{2n} \mid n \ge 0\}$

8.4 $\{\mathtt{0}^m\mathtt{1}^n \mid n \ne 2m\}$

8.5 $\{\mathtt{0}^i\mathtt{1}^j\mathtt{0}^{i+j} \mid i, j \ge 0\}$

8.6 $\{\mathtt{0}^{i+j}\mathtt{\#0}^j\mathtt{\#0}^i \mid i, j \ge 0\}$

8.7 $\{\mathtt{0}^i\mathtt{1}^j\mathtt{2}^k \mid j \ne i + k\}$

8.8 $\{\mathtt{0}^i\mathtt{1}^j\mathtt{2}^k \mid i = 2k \text{ or } 2i = k\}$ ***⟨⟨S18⟩⟩***

8.9 $\{\mathtt{0}^i\mathtt{1}^j\mathtt{2}^k \mid i + j = 2k\}$ ***⟨⟨F19⟩⟩***

8.10 $\left\{w\mathtt{\#0}^{\#(\mathtt{0},w)} \;\middle|\; w \in \{\mathtt{0},\mathtt{1}\}^*\right\}$

8.11 $\{\mathtt{0}^i\mathtt{1}^j\mathtt{2}^k \mid i = j \text{ or } j = k \text{ or } i = k\}$

8.12 $\{\mathtt{0}^i\mathtt{1}^j\mathtt{2}^k \mid i \ne j \text{ or } j \ne k\}$

8.13 $\{\mathtt{0}^{2i}\mathtt{1}^{i+j}\mathtt{2}^{2j} \mid i, j \ge 0\}$

8.14 $\left\{x\mathtt{\#}y^R \;\middle|\; x, y \in \{\mathtt{0},\mathtt{1}\}^* \text{ and } x \ne y\right\}$

8.15 All strings in $\{\mathtt{0},\mathtt{1}\}^*$ that are *not* palindromes.

8.16 $\left\{\mathtt{0}^n\mathtt{1}^{an+b} \;\middle|\; n \ge 0\right\}$, where a and b are arbitrary fixed natural numbers.

8.17 $\left\{\mathtt{0}^n\mathtt{1}^{an-b} \;\middle|\; n \ge b/a\right\}$, where a and b are arbitrary fixed natural numbers.

True or False (sanity check)

For each statement below, check "Yes" if the statement is ***ALWAYS*** true and "No" otherwise, and give a ***brief*** explanation of your answer. For example:

☒ Yes □ No If $2 + 2 = 5$ then Jeff is the Queen of England.
The hypothesis is false, so the implication is true.

□ Yes ☒ No $x + y$ is even.
Suppose $x = 1$ and $y = 0$.

☒ Yes □ No The set of all binary strings with an even number of 1s is regular.
`0*(10*10*)*`
—or—
Accepted by 2-state DFA, where current state $= \#1s \bmod 2$.

Read each statement *very* carefully. Some of these are deliberately subtle. On the other hand, you should not spend more than two minutes on any single statement.

Definitions

A.1 Every language is regular.

A.2 Every finite language is regular.

A.3 Every infinite language is regular.

A.4 For every language L, if L is regular then L can be represented by a regular expression.

A.5 For every language L, if L is not regular then L cannot be represented by a regular expression.

A.6 For every language L, if L can be represented by a regular expression, then L is regular.

A.7 For every language L, if L cannot be represented by a regular expression, then L is not regular.

A.8 For every language L, if there is a DFA that accepts every string in L, then L is regular.

A.9 For every language L, if there is a DFA that accepts every string not in L, then L is not regular.

A.10 For every language L, if there is a DFA that rejects every string not in L, then L is regular.

A.11 For every language L, if for every string $w \in L$ there is a DFA that accepts w, then L is regular.
⟨⟨*S14*⟩⟩

A.12 For every language L, if for every string $w \notin L$ there is a DFA that rejects w, then L is regular.

A.13 For every language L, if some DFA recognizes L, then some NFA also recognizes L.

A.14 For every language L, if some NFA recognizes L, then some DFA also recognizes L.

A.15 For every language L, if some NFA with ε-transitions recognizes L, then some NFA without ε-transitions also recognizes L.

A.16 For every language L, and for every string $w \in L$, there is a DFA that accepts w. ⟨⟨*F19*⟩⟩

A.17 Every regular language is recognized by a DFA with exactly 374 accepting states. ⟨⟨*F19*⟩⟩

A.18 Every regular language is recognized by an NFA with exactly 374 accepting states. ⟨⟨*F19*⟩⟩

Closure Properties of Regular Languages

B.1 For all regular languages L and L', the language $L \cap L'$ is regular.

B.2 For all regular languages L and L', the language $L \cup L'$ is regular.

B.3 For all regular languages L, the language L^* is regular.

B.4 For all regular languages A, B, and C, the language $(A \cup B) \setminus C$ is regular.

B.5 For all languages $L \subseteq \Sigma^*$, if L is regular, then $\Sigma^* \setminus L$ is regular.

B.6 For all languages $L \subseteq \Sigma^*$, if L is regular, then $\Sigma^* \setminus L$ is not regular.

B.7 For all languages $L \subseteq \Sigma^*$, if L is not regular, then $\Sigma^* \setminus L$ is regular.

B.8 For all languages $L \subseteq \Sigma^*$, if L is not regular, then $\Sigma^* \setminus L$ is not regular.

B.9 ⟨⟨*S14*⟩⟩ For all languages L and L', the language $L \cap L'$ is regular.

B.10 ⟨⟨*F14*⟩⟩ For all languages L and L', the language $L \cup L'$ is regular.

B.11 For every language L, the language L^* is regular. ⟨⟨*F14, F16*⟩⟩

B.12 For every language L, if L^* is regular, then L is regular.

B.13 For all languages A, B, and C, the language $(A \cup B) \setminus C$ is regular.

B.14 For every language L, if L is finite, then L is regular.

B.15 For all languages L and L', if L and L' are finite, then $L \cup L'$ is regular.

B.16 For all languages L and L', if L and L' are finite, then $L \cap L'$ is regular.

B.17 For all languages $L \subseteq \Sigma^*$, if L contains infinitely many strings in Σ^*, then L is not regular.

B.18 For all languages $L \subseteq \Sigma^*$, if L contains all but a finite number of strings of Σ^*, then L is regular. ⟨⟨*S14*⟩⟩

B.19 For all languages $L \subseteq \{\mathtt{0}, \mathtt{1}\}^*$, if L contains a finite number of strings in $\mathtt{0}^*$, then L is regular.

B.20 For all languages $L \subseteq \{\mathtt{0}, \mathtt{1}\}^*$, if L contains all but a finite number of strings in $\mathtt{0}^*$, then L is regular.

B.21 If L and L' are not regular, then $L \cap L'$ is not regular.

B.22 If L and L' are not regular, then $L \cup L'$ is not regular.

B.23 If L is regular and $L \cup L'$ is regular, then L' is regular. ⟨⟨*S14*⟩⟩

B.24 If L is regular and $L \cup L'$ is not regular, then L' is not regular. ⟨⟨*S14*⟩⟩

B.25 If L is not regular and $L \cup L'$ is regular, then L' is regular.

B.26 If L is regular and $L \cap L'$ is regular, then L' is regular.

B.27 If L is regular and $L \cap L'$ is not regular, then L' is not regular.

B.28 If L is regular and L' is finite, then $L \cup L'$ is regular. ⟨⟨*S14*⟩⟩

B.29 If L is regular and L' is finite, then $L \cap L'$ is regular.

B.30 If L is regular and $L \cap L'$ is finite, then L' is regular.

B.31 If L is regular and $L \cap L' = \varnothing$, then L' is not regular.

B.32 If L is not regular and $L \cap L' = \varnothing$, then L' is regular. ⟨⟨*F16*⟩⟩

B.33 If L is regular and L' is not regular, then $L \cap L' = \varnothing$.

B.34 If $L \subseteq L'$ and L is regular, then L' is regular.

B.35 If $L \subseteq L'$ and L' is regular, then L is regular. ⟨⟨*F14*⟩⟩

B.36 If $L \subseteq L'$ and L is not regular, then L' is not regular.

B.37 If $L \subseteq L'$ and L' is not regular, then L is not regular. ⟨⟨*F14*⟩⟩

B.38 Two languages L and L' are regular if and only if $L \cap L'$ is regular. ⟨⟨*F19*⟩⟩

B.39 For all languages $L \subseteq \Sigma^*$, if L cannot be described by a regular expression, then some DFA accepts $\Sigma^* \setminus L$.

B.40 For all languages $L \subseteq \Sigma^*$, if no DFA accepts L, then the complement $\Sigma^* \setminus L$ can be described by a regular expression.

B.41 For all languages $L \subseteq \Sigma^*$, if no DFA accepts L, then the complement $\Sigma^* \setminus L$ cannot be described by a regular expression.

B.42 For all languages $L \subseteq \Sigma^*$, if L is recognized by a DFA, then $\Sigma^* \setminus L$ can be described by a regular expression. ⟨⟨*F16*⟩⟩

Properties of Context-free Languages

C.1 For all languages $L \subseteq \Sigma^*$, if L cannot be recognized by a DFA, then L is context-free.

C.2 For all languages $L \subseteq \Sigma^*$, if L cannot be recognized by a DFA, then L is not context-free.

C.3 For all languages $L \subseteq \Sigma^*$, if L can be recognized by a DFA, then L is context-free.

C.4 For all languages $L \subseteq \Sigma^*$, if L can be recognized by a DFA, then L is not context-free.

C.5 For all languages $L \subseteq \Sigma^*$, if L is not context-free, then L is regular.

C.6 For all languages $L \subseteq \Sigma^*$, if L is not context-free, then $\Sigma^* \setminus L$ is regular.

C.7 For all languages $L \subseteq \Sigma^*$, if L is not context-free, then L is not regular.

C.8 For all languages $L \subseteq \Sigma^*$, if L is not context-free, then $\Sigma^* \setminus L$ is not regular.

C.9 The empty language is context-free. ⟨⟨*F19*⟩⟩

C.10 Every finite language is context-free.

C.11 Every context-free language is regular. ⟨⟨*F14*⟩⟩

C.12 Every regular language is context-free.

C.13 Every non-context-free language is non-regular. ⟨⟨*F16*⟩⟩

C.14 Every language is either regular or context-free. ⟨⟨*F19*⟩⟩

C.15 For all context-free languages L and L', the language $L \bullet L'$ is also context-free. ⟨⟨*F16*⟩⟩

C.16 For every context-free language L, the language L^* is also context-free.

C.17 For all context-free languages A, B, and C, the language $(A \cup B)^* \bullet C$ is also context-free.

C.18 For every language L, the language L^* is context-free.

C.19 For every language L, if L^* is context-free then L is context-free.

Equivalence Classes. Recall that for any language $L \subset \Sigma^*$, two strings $x, y \in \Sigma^*$ are equivalent with respect to L if and only if, for every string $z \in \Sigma^*$, either both xz and yz are in L, or neither xz nor yz is in L—or more concisely, if x and y have no distinguishing suffix with respect to L. We denote this equivalence by $x \equiv_L y$.

D.1 For every language L, if L is regular, then $\equiv_L$ has finitely many equivalence classes.

D.2 For every language L, if L is not regular, then $\equiv_L$ has infinitely many equivalence classes. ⟨⟨*S14*⟩⟩

D.3 For every language L, if $\equiv_L$ has finitely many equivalence classes, then L is regular.

D.4 For every language L, if $\equiv_L$ has infinitely many equivalence classes, then L is not regular.

D.5 For all regular languages L, each equivalence class of $\equiv_L$ is a regular language.

D.6 For every language L, each equivalence class of $\equiv_L$ is a regular language.

Fooling Sets

E.1 If a language L has an infinite fooling set, then L is not regular.

E.2 If a language L has an finite fooling set, then L is regular.

E.3 If a language L does not have an infinite fooling set, then L is regular.

E.4 If a language L is not regular, then L has an infinite fooling set.

E.5 If a language L is regular, then L has no infinite fooling set.

E.6 If a language L is not regular, then L has no finite fooling set. ⟨⟨*F14, F16*⟩⟩

E.7 If a language L has a fooling set of size 374, then L is not regular. ⟨⟨*F19*⟩⟩

E.8 If a language L does not have a fooling set of size 374, then L is regular. ⟨⟨*F19*⟩⟩

Specific Languages (Gut Check). Do *not* construct complete DFAs, NFAs, regular expressions, or fooling-set arguments for these languages. You don't have time.

F.1 $\{0^i1^j2^k \mid i+j-k=374\}$ is regular. ⟨⟨*S14*⟩⟩

F.2 $\{0^i1^j2^k \mid i+j-k\leq 374\}$ is regular.

F.3 $\{0^i1^j2^k \mid i+j+k=374\}$ is regular.

F.4 $\{0^i1^j2^k \mid i+j+k>374\}$ is regular.

F.5 $\{0^i1^j \mid i<374<j\}$ is regular. ⟨⟨*S14*⟩⟩

F.6 $\left\{0^m1^n \;\middle|\; 0\leq m+n\leq 374\right\}$ is regular. ⟨⟨*F14*⟩⟩

F.7 $\left\{0^m1^n \;\middle|\; 0\leq m-n\leq 374\right\}$ is regular. ⟨⟨*F14*⟩⟩

F.8 $\{0^i1^j \mid i,j\geq 0\}$ is not regular. ⟨⟨*F16*⟩⟩

F.9 $\{0^i1^j \mid (i-j)$ is divisible by 374$\}$ is regular. ⟨⟨*S14*⟩⟩

F.10 $\{0^i1^j \mid (i+j)$ is divisible by 374$\}$ is regular.

F.11 $\left\{0^{n^2} \;\middle|\; n\geq 0\right\}$ is regular.

F.12 $\left\{0^{37n+4} \;\middle|\; n\geq 0\right\}$ is regular.

F.13 $\left\{0^n10^n \;\middle|\; n\geq 0\right\}$ is regular.

F.14 $\left\{0^m10^n \;\middle|\; m\geq 0 \text{ and } n\geq 0\right\}$ is regular.

F.15 $\left\{0^{374n} \;\middle|\; n\geq 0\right\}$ is regular. ⟨⟨*F19*⟩⟩

F.16 $\left\{0^{37n}1^{4n} \;\middle|\; n\geq 374\right\}$ is regular. ⟨⟨*F19*⟩⟩

F.17 $\left\{0^{37n}1^{4n} \;\middle|\; n\leq 374\right\}$ is regular. ⟨⟨*F19*⟩⟩

F.18 $\{w \in \{0,1\}^* \mid |w| \text{ is divisible by } 374\}$ is regular.

F.19 $\{w \in \{0,1\}^* \mid w \text{ represents a integer divisible by } 374 \text{ in binary}\}$ is regular.

F.20 $\{w \in \{0,1\}^* \mid w \text{ represents a integer divisible by } 374 \text{ in base } 473\}$ is regular.

F.21 $\left\{w \in \{0,1\}^* \;\middle|\; |\#(0,w) - \#(1,w)| < 374\right\}$ is regular.

F.22 $\left\{w \in \{0,1\}^* \;\middle|\; |\#(0,x) - \#(1,x)| < 374 \text{ for every prefix } x \text{ of } w\right\}$ is regular.

F.23 $\left\{w \in \{0,1\}^* \;\middle|\; |\#(0,x) - \#(1,x)| < 374 \text{ for every substring } x \text{ of } w\right\}$ is regular.

F.24 $\left\{w0^{\#(0,w)} \;\middle|\; w \in \{0,1\}^*\right\}$ is regular.

F.25 $\left\{w0^{\#(0,w) \bmod 374} \;\middle|\; w \in \{0,1\}^*\right\}$ is regular.

Playing with Automata

G.1 Let $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q, s, Q \setminus A, \delta)$ be arbitrary **DFAs** with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then $L(M) \cap L(M') = \varnothing$. ⟨⟨*F16*⟩⟩

G.2 Let $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q, s, Q \setminus A, \delta)$ be arbitrary **NFAs** with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then $L(M) \cap L(M') = \varnothing$. ⟨⟨*F16*⟩⟩

G.3 Let M be a **DFA** over the alphabet Σ. Let M' be identical to M, except that accepting states in M are non-accepting in M' and vice versa. Each string in Σ^* is accepted by exactly one of M and M'.

G.4 Let M be an **NFA** over the alphabet Σ. Let M' be identical to M, except that accepting states in M are non-accepting in M' and vice versa. Each string in Σ^* is accepted by exactly one of M and M'.

G.5 If a language L is recognized by a DFA with n states, then the complementary language $\Sigma^* \setminus L$ is recognized by a DFA with at most $n+1$ states.

G.6 If a language L is recognized by an NFA with n states, then the complementary language $\Sigma^* \setminus L$ is recognized by a NFA with at most $n+1$ states.

G.7 If a language L is recognized by a DFA with n states, then L^* is recognized by a DFA with at most $n+1$ states.

G.8 If a language L is recognized by an NFA with n states, then L^* is recognized by a NFA with at most $n+1$ states.

Language Transformations

H.1 For every regular language L, the language $\left\{w^R \;\middle|\; w \in L\right\}$ is also regular.

H.2 For every language L, if the language $\left\{w^R \;\middle|\; w \in L\right\}$ is regular, then L is also regular. ⟨⟨*F14*⟩⟩

H.3 For every language L, if the language $\left\{w^R \;\middle|\; w \in L\right\}$ is not regular, then L is also not regular. ⟨⟨*F14*⟩⟩

H.4 For every regular language L, the language $\left\{w \;\middle|\; ww^R \in L\right\}$ is also regular.

H.5 For every regular language L, the language $\left\{ww^R \;\middle|\; w \in L\right\}$ is also regular.

H.6 For every language L, if the language $\left\{w \;\middle|\; ww^R \in L\right\}$ is regular, then L is also regular. *[Hint: Consider the language $L = \{0^n 1^n \mid n \geq 0\}$.]*

H.7 For every regular language L, the language $\left\{0^{|w|} \;\middle|\; w \in L\right\}$ is also regular.

H.8 For every language L, if the language $\left\{0^{|w|} \;\middle|\; w \in L\right\}$ is regular, then L is also regular.

H.9 For every context-free language L, the language $\left\{w^R \;\middle|\; w \in L\right\}$ is also context-free.

CS/ECE 374 A ✧ Fall 2021 Fake Midterm 1 Problem 1	Name:

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.

□ Yes □ No

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

□ Yes □ No

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

□ Yes □ No

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

□ Yes □ No

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

□ Yes □ No

(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.

□ Yes □ No

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

□ Yes □ No

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

□ Yes □ No

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

□ Yes □ No

(j) Every language is either regular or context-free.

□ Yes □ No

CS/ECE 374 A ✧ Fall 2021 **Fake Midterm 1 Problem 2**	Name:

For each of the following languages over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, 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. $\left\{\mathtt{0}^n w \mathtt{0}^n \;\middle|\; w \in \Sigma^+ \text{ and } n > 0\right\}$

2. $\left\{w \mathtt{0}^n w \;\middle|\; w \in \Sigma^+ \text{ and } n > 0\right\}$

CS/ECE 374 A ✧ Fall 2021 Fake Midterm 1 Problem 3	Name:

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:

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

(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 $OddParity(L) := \{w \in L \mid parity(w) = \texttt{1}\}$ is also regular.

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

[Hint: Yes, you have enough room.]

CS/ECE 374 A ✧ Fall 2021 Fake Midterm 1 Problem 4	Name:

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.

CS/ECE 374 A ✧ Fall 2021 Fake Midterm 1 Problem 5	Name:

For any string $w \in \{0, 1\}^*$, let *obliviate*(w) denote the string obtained from w by removing every 1. For example:

$$\begin{aligned} \mathit{obliviate}(\varepsilon) &= \varepsilon \\ \mathit{obliviate}(\texttt{000000}) &= \texttt{000000} \\ \mathit{obliviate}(\texttt{111111}) &= \varepsilon \\ \mathit{obliviate}(\texttt{010001101}) &= \texttt{00000} \end{aligned}$$

Let L be an arbitrary regular language.

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

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

CS/ECE 374 A ✧ Fall 2021

Midterm 1

September 27, 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.
- 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 boldt 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) $\{0^p 1^q 0^r \mid r = p + q\}$

   (b) $\{0^p 1^q 0^r \mid r = p + q \bmod 2\}$
   [Hint: First think about the language $\{0^p 1^q \mid q = p \bmod 2\}$*]*

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

   Let $\text{take2skip2}(w)$ be a function 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, skip the next two, takes the next two, skips the next two, and so on. For example:

   $$\text{take2skip2}(\underline{1}) = 1$$
   $$\text{take2skip2}(\underline{01}0) = 01$$
   $$\text{take2skip2}(\underline{01}00\underline{11}11\underline{00}01\underline{1}) = 0111001$$

   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) $\{0^p 1^q 0^r \mid r > 0 \quad \text{and} \quad q \bmod r = 0 \quad \text{and} \quad p \bmod r = 0\}$

   (b) $\{0^p 1^q \mid q > 0 \quad \text{and} \quad p = q^q\}$

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 $\text{MINGLE}(w, z \bullet z^R) = (\text{MINGLE}(w, z \bullet z^R))^R$ 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 ε-reach of any state in an NFA contains the state itself.

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

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

CS/ECE 374 A ✧ Fall 2021

Conflict Midterm 1

September 28, 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.
- 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 $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, either prove that the language is regular (by constructing an appropriate DFA, NFA, or regular expression) or prove that the language is not regular (by constructing an infinite fooling set and proving that the set you construct is indeed a fooling set for that language).

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

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

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

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

   $$\mathsf{compress}(\underline{\mathtt{00000}}\mathtt{11}\underline{\mathtt{000}}\mathtt{1}) = \underline{\mathtt{000}}\mathtt{11}\underline{\mathtt{00}}\mathtt{1}$$
   $$\mathsf{compress}(\mathtt{11}\underline{\mathtt{0000}}\mathtt{1}\underline{\mathtt{0}}) = \underline{\mathtt{1100}}\mathtt{1}\underline{\mathtt{0}}$$
   $$\mathsf{compress}(\mathtt{11111}) = \mathtt{11111}$$

   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) $\left\{w \in \Sigma^* \;\middle|\; \mathsf{compress}(w) \in L\right\}$

   (b) $\left\{\mathsf{compress}(w) \;\middle|\; w \in L\right\}$

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) $\{\mathtt{0}^p\mathtt{1}^q\mathtt{0}^r \mid p > 0 \text{ and } q > 0 \text{ and } r = \gcd(p, q)\}$.

   (b) $\{\mathtt{0}^p\mathtt{1}^{pq} \mid p > 0 \text{ and } q > 0\}$

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

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

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

   (b) Prove that $\mathsf{RO}(\mathsf{RO}(w)) = \mathsf{R0}(w)$ 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) $\{0^n1 \mid n > 0\}$ is the only infinite fooling set for the language $\{0^n10^n \mid n > 0\}$.

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

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

   (d) Any language that can be decided by an NFA with ε-transitions can also be decided by an NFA without ε-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 $\{ww^C \mid w \in L\}$ 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 $\{xy^C \mid x, y \in L\}$ is also regular.

   (g) The ε-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 $\{0^{2p} \mid p \text{ is prime}\}$ is regular.

CS/ECE 374 A ✧ Fall 2021

Midterm 2 Study Questions

This is a “core dump” of potential questions for Midterm 2. This should give you a good idea of the *types* of questions that we will ask on the exam, but the actual exam questions may or may not appear in this list. This list intentionally includes a few questions that are too long or difficult for exam conditions; *most* of these are indicated with a *star.

Questions from Jeff’s past exams are labeled with the semester they were used, for example, ***⟨⟨S18⟩⟩*** or ***⟨⟨F19⟩⟩***. Questions from this semester’s homework are labeled ***⟨⟨HW⟩⟩***. Questions from this semester’s labs are labeled ***⟨⟨Lab⟩⟩***. Some unflagged questions may have been used in exams by other instructors.

How to Use These Problems

Solving every problem in this handout is ***not*** the best way to study for the exam. Memorizing the solutions to every problem in this handout is the ***absolute worst*** way to study for the exam.

What we recommend instead is to work on a *sample* of the problems. Choose one or two problems at random from each section and try to solve them from scratch under exam conditions—by yourself, in a quiet room, with a 30-minute timer, *without* your notes, *without* the internet, and if possible, even without your cheat sheet. If you’re comfortable solving a few problems in a particular section, you’re probably ready for that type of problem on the exam. Move on to the next section.

Discussing problems with other people (in your study groups, in the review sessions, in office hours, or on Piazza) and/or looking up old solutions can be *extremely* helpful, but ***only after*** you have (1) made a good-faith effort to solve the problem on your own, and (2) you have either a candidate solution or some idea about where you’re getting stuck.

If you find yourself getting stuck on a particular type of problem, try to figure out *why* you’re stuck. Do you understand the problem statement? Have you tried several example inputs to see what the correct output *should* be? Are you stuck on choosing the right high-level approach, are you stuck on the details, or are you struggling to express your ideas clearly?

Similarly, if feedback suggests that your solutions to a particular type of problem are incorrect or incomplete, try to figure out what you missed. For recursion/dynamic programming: Are you solving the right recursive generalization of the stated problem? Are you having trouble writing a specification of the function, as opposed to a description of the algorithm? Are you struggling to find a good evaluation order? Are you trying to use a greedy algorithm? *[Hint: Don’t.]* For graph algorithms: Are you aiming for the right problem? Are you having trouble figuring out the interesting states of the problem (otherwise known as vertices) and the transitions between them (otherwise known as edges)? Do you keep trying to modify the algorithm instead of modifying the graph?

Remember that your goal is *not* merely to “understand” the solution to any particular problem, but to become more comfortable with solving a certain *type* of problem on your own. **"Understanding" is a trap; aim for mastery.** If you can identify specific steps that you find problematic, read more *about those steps*, focus your practice *on those steps*, and try to find helpful information *about those steps* to write on your cheat sheet. Then work on the next problem!

Recursion and Dynamic Programming

Elementary Recursion/Divide and Conquer

1. ⟨⟨*Lab*⟩⟩

   (a) Suppose $A[1..n]$ is an array of n distinct integers, sorted so that $A[1] < A[2] < \cdots < A[n]$. Each integer $A[i]$ could be positive, negative, or zero. Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists..

   (b) Now suppose $A[1..n]$ is a sorted array of n distinct ***positive*** integers. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. *[Hint: This is* **really** *easy.]*

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

9	7	7	2	1	3	7	5	4	7	3	3	4	8	6	9
	▲			▲				▲		▲	▲			▲	

   Describe and analyze a fast algorithm that returns the index of one local minimum. For example, given the array above, your algorithm could return the integer 5, because $A[5]$ is a local minimum. *[Hint: With the given boundary conditions, any array* ***must*** *contain at least one local minimum. Why?]*

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

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

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

4. ⟨⟨*F14, S14*⟩⟩ An array $A[0..n-1]$ of n distinct numbers is ***bitonic*** if there are unique indices i and j such that $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ and $A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,

4	6	9	8	7	5	1	2	3	is bitonic, but
3	6	9	8	7	5	1	2	4	is *not* bitonic.

   Describe and analyze an algorithm to find the index of the *smallest* element in a given bitonic array $A[0..n-1]$ in $O(\log n)$ time. You may assume that the numbers in the input array are distinct. For example, given the first array above, your algorithm should return 6, because $A[6] = 1$ is the smallest element in that array.

5. ⟨⟨*F16*⟩⟩ Suppose you are given a sorted array $A[1..n]$ of distinct numbers that has been *rotated* k steps, for some ***unknown*** integer k between 1 and $n-1$. That is, the prefix $A[1..k]$ is sorted in increasing order, the suffix $A[k+1..n]$ is sorted in increasing order, and $A[n] < A[1]$. For example, you might be given the following 16-element array (where $k = 10$):

9	13	16	18	19	23	28	31	37	42	−4	0	2	5	7	8

Describe and analyze an efficient algorithm to determine if the given array contains a given number x. The input to your algorithm is the array $A[1..n]$ and the number x; your algorithm is ***not*** given the integer k.

6. ⟨⟨*F16*⟩⟩ Suppose you are given two unsorted arrays $A[1..n]$ and $B[1..n]$ containing $2n$ distinct integers, such that $A[1] < B[1]$ and $A[n] > B[n]$. Describe and analyze an efficient algorithm to compute an index i such that $A[i] < B[i]$ and $A[i+1] > B[i+1]$. *[Hint: Why does such an index i always exist?]*

7. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a *flip*—insert a spatula under the top k pancakes, for some integer k between 1 and n, and flip them all over.

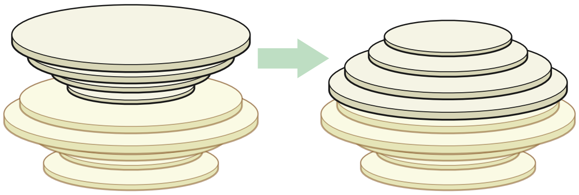


Figure 1. Flipping the top four pancakes.

   (a) Describe an algorithm to sort an arbitrary stack of n pancakes using as few flips as possible. *Exactly* how many flips does your algorithm perform in the worst case?

   (b) Now suppose one side of each pancake is burned. Describe an algorithm to sort an arbitrary stack of n pancakes, so that the burned side of every pancake is facing down, using as few flips as possible. *Exactly* how many flips does your algorithm perform in the worst case?

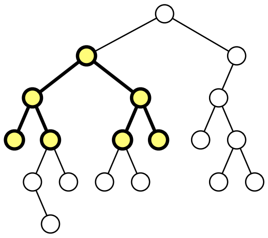
   *[Hint: This problem has **nothing** to do with the Tower of Hanoi!]*

8. (a) Describe an algorithm to determine in $O(n)$ time whether an arbitrary array $A[1..n]$ contains more than $n/4$ copies of any value.

   (b) Describe and analyze an algorithm to determine, given an arbitrary array $A[1..n]$ and an integer k, whether A contains more than k copies of any value. Express the running time of your algorithm as a function of both n and k.

   Do not use hashing, or radix sort, or any other method that depends on the precise input values, as opposed to their order.

9. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *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 *largest complete subtree* of a given binary tree. Your algorithm should return both the root and the depth of this subtree.



The largest complete subtree of this binary tree has depth 2.

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

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

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

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

Dynamic Programming

1. ⟨⟨*Lab*⟩⟩ Describe and analyze efficient algorithms for the following problems.

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

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

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

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

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

2. ⟨⟨*F19*⟩⟩

   (a) Recall that a *palindrome* is any string that is equal to its reversal, like `REDIVIDER` or `POOP`. Describe an algorithm to find the length of the longest subsequence of a given string that is a palindrome.

   (b) A *double palindrome* is the concatenation of two *non-empty* palindromes, like `AREDIVIDER` or `POOPPOOP`. Describe an algorithm to find the length of the longest subsequence of a given string that is a *double* palindrome. *[Hint: Use your algorithm from part (a).]*

   For both algorithms, the input is an array $A[1..n]$, and the output is an integer. For example, given the string `MAYBEDYNAMICPROGRAMMING` as input, your algorithm for part (a) should return 7 (for the palindrome subsequence `NMRORMN`), and your algorithm for part (b) should return 12 (for the double palindrome subsequence `MAYBYAMIRORI`).

3. ⟨⟨*S14*⟩⟩ Recall that a *palindrome* is any string that is the same as its reversal. For example, `I`, `DAD`, `HANNAH`, `AIBOHPHOBIA` (fear of palindromes), and the empty string are all palindromes.

   (a) Describe and analyze an algorithm to find the length of the longest substring (not *subsequence*!) of a given input string that is a palindrome. For example, **`BASEESAB`** is the longest palindrome substring of `BUB`**`BASEESAB`**`ANANA` ("Bubba sees a banana."). Thus, given the input string `BUBBASEESABANANA`, your algorithm should return the integer 8.

   (b) Any string can be decomposed into a sequence of palindrome substrings. For example, the string `BUBBASEESABANANA` can be broken into palindromes in the following ways (and many others):

$$\mathsf{BUB} + \mathsf{BASEESAB} + \mathsf{ANANA}$$
$$\mathsf{B} + \mathsf{U} + \mathsf{BB} + \mathsf{A} + \mathsf{SEES} + \mathsf{ABA} + \mathsf{NAN} + \mathsf{A}$$
$$\mathsf{B} + \mathsf{U} + \mathsf{BB} + \mathsf{A} + \mathsf{SEES} + \mathsf{A} + \mathsf{B} + \mathsf{ANANA}$$
$$\mathsf{B} + \mathsf{U} + \mathsf{B} + \mathsf{B} + \mathsf{A} + \mathsf{S} + \mathsf{E} + \mathsf{E} + \mathsf{S} + \mathsf{A} + \mathsf{B} + \mathsf{A} + \mathsf{N} + \mathsf{A} + \mathsf{N} + \mathsf{A}$$

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

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

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

$$\mathsf{BUB} \bullet \mathsf{B} \bullet \mathsf{ALA} \bullet \mathsf{SEES} \bullet \mathsf{ABA} \bullet \mathsf{N} \bullet \mathsf{ANA}$$

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

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

4. *⟨⟨F16⟩⟩* It's almost time to show off your flippin' sweet dancing skills! Tomorrow is the big dance contest you've been training for your entire life, except for that summer you spent with your uncle in Alaska hunting wolverines. You've obtained an advance copy of the the list of n songs that the judges will play during the contest, in chronological order.

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

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

5. *⟨⟨S16⟩⟩* After the Revolutionary War, Alexander Hamilton's biggest rival as a lawyer was Aaron Burr. (Sir!) In fact, the two worked next door to each other. Unlike Hamilton, Burr cannot work non-stop; every case he tries exhausts him. The bigger the case, the longer he must rest before he is well enough to take the next case. (Of course, he is willing to wait for it.) If a case arrives while Burr is resting, Hamilton snatches it up instead.

Burr has been asked to consider a sequence of n upcoming cases. He quickly computes two arrays $\mathit{profit}[1..n]$ and $\mathit{skip}[1..n]$, where for each index i,

- *profit*$[i]$ is the amount of money Burr would make by taking the ith case, and
- *skip*$[i]$ is the number of consecutive cases Burr must skip if he accepts the ith case. That is, if Burr accepts the ith case, he cannot accept cases $i+1$ through $i + skip[i]$.

Design and analyze an algorithm that determines the maximum total profit Burr can secure from these n cases, using his two arrays as input.

6. ⟨⟨*F16*⟩⟩ A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways.

$$\mathtt{BANANA}{}_{\mathtt{ANANAS}} \qquad \mathtt{BAN}{}_{\mathtt{ANA}}{}^{\mathtt{ANA}}{}_{\mathtt{NAS}} \qquad \mathtt{B}{}_{\mathtt{AN}}{}^{\mathtt{AN}}{}_{\mathtt{A}}{}^{\mathtt{A}}{}_{\mathtt{NA}}{}^{\mathtt{NA}}{}_{\mathtt{S}}$$

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

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

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

7. 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:

$$32492 \to \overset{57^2}{3249} \to \overset{18^2}{324} \to 24 \to \overset{2^2}{4} \to \varepsilon.$$

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 IsSquare that determines whether a given k-digit number (represented by an array of digits) is a perfect square ***in $O(k^2)$ time***.

8. Suppose you are given a sequence of non-negative integers separated by + and × signs; for example:

$$2 \times 3 + 0 \times 6 \times 1 + 4 \times 2$$

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

$$2 \times (3 + (0 \times (6 \times (1 + (4 \times 2))))) = 6$$
$$(((((2 \times 3) + 0) \times 6) \times 1) + 4) \times 2 = 80$$
$$((2 \times 3) + (0 \times 6)) \times (1 + (4 \times 2)) = 108$$
$$(((2 \times 3) + 0) \times 6) \times ((1 + 4) \times 2) = 360$$

Describe and analyze an algorithm to compute, given a list of integers separated by + and × signs, the smallest possible value we can obtain by inserting parentheses.

Your input is an array $A[0..2n]$ where each $A[i]$ is an integer if i is even and $+$ or $\times$ if i is odd. Assume any arithmetic operation in your algorithm takes $O(1)$ time.

9. Suppose you are given three strings $A[1\,..\,n]$, $B[1\,..\,n]$, and $C[1\,..\,n]$.

   (a) Describe and analyze an algorithm to find the length of the longest common subsequence of all three strings. For example, given the input strings

   $$A = \texttt{AxxBxxCDxEF}, \qquad B = \texttt{yyABCDyEyFy}, \qquad C = \texttt{zAzzBCDzEFz},$$

   your algorithm should output the number 6, which is the length of the longest common subsequence `ABCDEF`.

   (b) Describe and analyze an algorithm to find the length of the shortest common supersequence of all three strings. For example, given the input strings

   $$A = \texttt{AxxBxxCDxEF}, \qquad B = \texttt{yyABCDyEyFy}, \qquad C = \texttt{zAzzBCDzEFz},$$

   your algorithm should output the number 21, which is the length of the shortest common supersequence `yzyAxzzxBxxCDxyzEyFzy`.

10. (a) Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

    (b) Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which ***every*** pair of segments intersects.

11. ⟨⟨***S18***⟩⟩ Suppose we want to split an array $A[1\,..\,n]$ of integers into k contiguous intervals that partition the sum of the values as evenly as possible. Specifically, define the *cost* of such a partition as the maximum, over all k intervals, of the sum of the values in that interval; our goal is to minimize this cost. Describe and analyze an algorithm to compute the minimum cost of a partition of A into k intervals, given the array A and the integer k as input.

    For example, given the array $A = [1, 6, -1, 8, 0, 3, 3, 9, 8, 8, 7, 4, 9, 8, 9, 4, 8, 4, 8, 2]$ and the integer $k = 3$ as input, your algorithm should return the integer 37, which is the cost of the following partition:

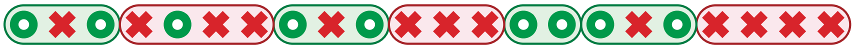
    $$\Big[\, \overbrace{1, 6, -1, 8, 0, 3, 3, 9, 8}^{37} \;\Big|\; \overbrace{8, 7, 4, 9, 8}^{36} \;\Big|\; \overbrace{9, 4, 8, 4, 8, 2}^{35} \,\Big]$$

    The numbers above each interval show the sum of the values in that interval.

12. ⟨⟨***S18***⟩⟩ The City Council of Sham-Poobanana needs to partition Purple Street into voting districts. A total of n people live on Purple Street, at consecutive addresses $1, 2, \ldots, n$. Each voting district must be a contiguous interval of addresses $i, i+1, \ldots, j$ for some $1 \le i < j \le n$. By law, each Purple Street address must lie in exactly one district, and the number of addresses in each district must be between k and $2k$, where k is some positive integer parameter.

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

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



Describe an algorithm to find the largest possible number of *good* districts in a legal partition. Your input consists of the integer k and a boolean array $\text{GoodVote}[1\,..\,n]$ indicating which residents previously voted for Oceania (True) or Eurasia (False). You can assume that a legal partition exists. Analyze the running time of your algorithm in terms of the parameters n and k.

13. Suppose you are given an $m \times n$ bitmap, represented by an array $M[1\,..\,n, 1\,..\,n]$ of 0s and 1s. A *solid square block* in M is a subarray of the form $M[i\,..\,i+w, j\,..\,j+w]$ containing only 1-bits. Describe and analyze an algorithm to find the largest solid square block in M.

14. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

    Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it's his turn, Elmo *always* takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

    (a) Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do *not* follow the same greedy strategy as Elmo.

    (b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

    (c) Five years later, thirteen-year-old Elmo has become a *much* stronger player. 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.

15. ⟨⟨*S16*⟩⟩ Your nephew Elmo is visiting you for Christmas, and he's brought a different card game. Like your previous game with Elmo, this game is played with a row of n cards, each labeled with an integer (which could be positive, negative, or zero). Both players can see all n card values. Otherwise, the game is almost completely different.

On each turn, the current player must take the leftmost card. The player can either keep the card or give it to their opponent. If they keep the card, their turn ends and their opponent takes the next card; however, if they give the card to their opponent, the current player's turn continues with the next card. In short, the player that does *not* get the ith card decides who gets the $(i+1)$th card. The game ends when all cards have been played. Each player adds up their card values, and whoever has the higher total wins.

For example, suppose the initial cards are $[3, -1, 4, 1, 5, 9]$, and Elmo plays first. Then the game might proceed as follows:

- Elmo keeps the 3, ending his turn.
- You give Elmo the -1.
- You keep the 4, ending your turn.
- Elmo gives you the 1.
- Elmo gives you the 5.
- Elmo keeps the 9, ending his turn. All cards are gone, so the game is over.
- Your score is $1+4+5=10$ and Elmo's score is $3-1+9=11$, so Elmo wins.

Describe an algorithm to compute the highest possible score you can earn from a given row of cards, assuming Elmo plays first and plays perfectly. Your input is the array $C[1..n]$ of card values. For example, if the input is $[3, -1, 4, 1, 5, 9]$, your algorithm should return the integer 10.

16. ⟨⟨*F14*⟩⟩ The new mobile game *Candy Swap Saga XIII* involves n cute animals numbered 1 through n. Each animal holds one of three types of candy: circus peanuts, Heath bars, and Cioccolateria Gardini chocolate truffles. You also have a candy in your hand; at the start of the game, you have a circus peanut.

To earn points, you visit each of the animals in order from 1 to n. For each animal, you can either keep the candy in your hand or exchange it with the candy the animal is holding.

- If you swap your candy for another candy of the *same* type, you earn one point.
- If you swap your candy for a candy of a *different* type, you lose one point. (Yes, your score can be negative.)
- If you visit an animal and decide not to swap candy, your score does not change.

You *must* visit the animals in order, and once you visit an animal, you can never visit it again.

Describe and analyze an efficient algorithm to compute your maximum possible score. Your input is an array $C[1..n]$, where $C[i]$ is the type of candy that the ith animal is holding.

17. ⟨⟨*F14*⟩⟩ Farmers Boggis, Bunce, and Bean have set up an obstacle course for Mr. Fox. The course consists of a row of n booths, each with an integer painted on the front with bright red paint, which could be positive, negative, or zero. Let $A[i]$ denote the number painted on the front of the ith booth. Everyone has agreed to the following rules:

- At each booth, Mr. Fox ***must*** say either “Ring!” or “Ding!”.
- If Mr. Fox says “Ring!” at the ith booth, he earns a reward of $A[i]$ chickens. (If $A[i] < 0$, Mr. Fox pays a penalty of $-A[i]$ chickens.)
- If Mr. Fox says “Ding!” at the ith booth, he pays a penalty of $A[i]$ chickens. (If $A[i] < 0$, Mr. Fox earns a reward of $-A[i]$ chickens.)
- Mr. Fox is forbidden to say the same word more than three times in a row. For example, if he says “Ring!” at booths 6, 7, and 8, then he *must* say “Ding!” at booth 9.
- All accounts will be settled at the end; Mr. Fox does not actually have to carry chickens through the obstacle course.
- If Mr. Fox violates any of the rules, or if he ends the obstacle course owing the farmers chickens, the farmers will shoot him.

Describe and analyze an algorithm to compute the largest number of chickens that Mr. Fox can earn by running the obstacle course, given the array $A[1..n]$ of booth numbers as input.

18. ***⟨⟨F19⟩⟩*** Satya is in charge of establishing a new testing center for the Standardized Awesomeness Test (SAT), and he found an old conference hall that is perfect. The conference hall has n rooms of various sizes along a single long hallway, numbered in order from 1 through n. Satya knows exactly how many students fit into each room, and he wants to use a subset of the rooms to host as many students as possible for testing.

Unfortunately, there have been several incidents of students cheating at other testing centers by tapping secret codes through walls. To prevent this type of cheating, Satya can use two adjacent rooms only if he demolishes the wall between them. For example, if Satya wants to use rooms 1, 3, 4, 5, 7, 8, and 10, he must demolish three walls: between rooms 3 and 4, between rooms 4 and 5, and between rooms 7 and 8.

(a) The city’s chief architect has determined that demolishing the walls on both sides of the same room would threaten the building’s structural integrity. For this reason, Satya can never host students in three consecutive rooms.

Describe an efficient algorithm that computes the largest number of students that Satya can host for testing without using three consecutive rooms.

The input to your algorithm is an array $S[1..n]$, where each $S[i]$ is the (non-negative integer) number of students that can fit in room i.

(b) The city’s chief architect has determined that demolishing more than k walls would threaten the structural integrity of the building.

Describe an efficient algorithm that computes the largest number of students that Satya can host for testing without demolishing more than k walls.

The input to your algorithm is the integer k and an array $S[1..n]$, where each $S[i]$ is the (non-negative integer) number of students that can fit in room i. Analyze your algorithm as a function of both n and k.

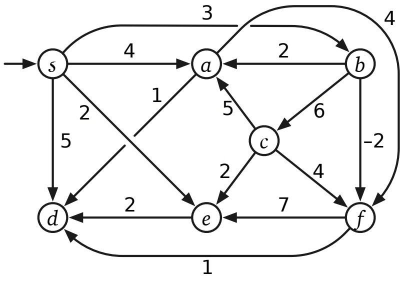
Parts (a) and (b) appeared as complete problems in different versions of the same exam.

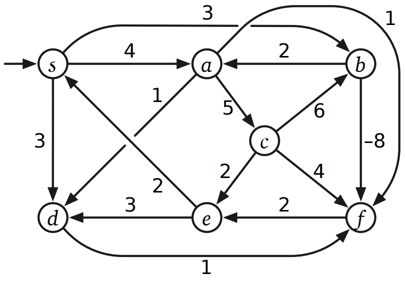
Graph Algorithms

Sanity Check

1. *⟨⟨S14, F14, F16, F19⟩⟩* Indicate the following structures in the example graphs below.

   - To indicate a subgraph (such as a path, a spanning tree, or a cycle), draw over every edge in the subgraph with a **heavy black line**. Your subgraph should be visible from across the room.
   - To indicate a subset of vertices, either draw a heavy black line around the entire subset, completely blacken the vertices in the subset, or list the vertex labels.
   - If the requested structure does not exist, just write the word NONE.





(a) A depth-first spanning tree rooted at node s.

(b) A breadth-first spanning tree rooted at node s.

(c) A shortest-path tree rooted at node s.

(d) The set of all vertices reachable from node c.

(e) The set of all vertices that can reach node c.

(f) The strong components. (Circle each strong component.)

(g) A simple cycle containing vertex s.

(h) A directed cycle with the minimum number of edges.

(i) A directed cycle with the smallest total weight.

(j) A walk from s to d with the maximum number of edges.

(k) A walk from s to d with the largest total weight.

(l) A depth-first pre-ordering of the vertices. (List the vertices in order.)

(m) A depth-first post-ordering of the vertices. (List the vertices in order.)

(n) A topological ordering of the vertices. (List the vertices in order.)

(o) A breadth-first ordering of the vertices. (List the vertices in order.)

(p) Draw the strong-component graph.

[On an actual exam, we would only ask about one graph, and we would ask for only a few of these structures. If the exam is given on paper, we would give you several copies of the graph on which to mark your answers.]

Reachability/Connectivity/Traversal

1. Describe and analyze algorithms for the following problems; in each problem, you are given a graph $G = (V, E)$ with unweighted edges, which may be directed or undirected. You may or may not need different algorithms for directed and undirected graphs.

   (a) Find two vertices that are (strongly) connected.

   (b) Find two vertices that are not (strongly) connected.

   (c) Find two vertices, such that neither vertex can reach the other.

   (d) Find all vertices reachable from a given vertex s.

   (e) Find all vertices that can reach a given vertex s.

   (f) Find all vertices that are strongly connected to a given vertex s.

   (g) Find a simple cycle, or correctly report that the graph has no cycles. (A simple cycle is a closed walk that visits each vertex at most once.)

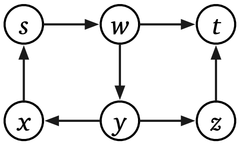
   (h) Find the *shortest* simple cycle, or correctly report that the graph has no cycles.

   (i) Determine whether deleting a given vertex v would disconnect the graph.

   [On an actual exam, we would ask for at most a few of these structures, and we would specify whether the input graph is directed or undirected.]

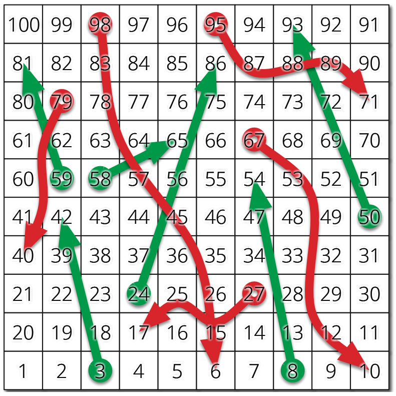
2. ***⟨⟨F14⟩⟩*** Suppose you are given a directed graph $G = (V, E)$ and two vertices s and t. Describe and analyze an algorithm to determine if there is a walk in G from s to t (possibly repeating vertices and/or edges) whose length is divisible by 3.

   For example, given the graph below, with the indicated vertices s and t, your algorithm should return TRUE, because the walk $s \to w \to y \to x \to s \to w \to t$ has length 6.



   [Hint: Build a (different) graph.]

3. ***⟨⟨Lab⟩⟩*** ***Snakes and Ladders*** is a classic board game, originating in India no later than the 16th century. The board consists of an $n \times n$ grid of squares, numbered consecutively from 1 to n^2, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares, always in different rows, are connected by either "snakes" (leading down) or "ladders" (leading up). Each square can be an endpoint of at most one snake or ladder.



A typical Snakes and Ladders board.
Upward straight arrows are ladders; downward wavy arrows are snakes.

You start with a token in cell 1, in the bottom left corner. In each move, you advance your token up to k positions, for some fixed constant k (typically 6). If the token ends the move at the *top* end of a snake, you ***must*** slide the token down to the bottom of that snake. If the token ends the move at the *bottom* end of a ladder, you ***may*** move the token up to the top of that ladder.

Describe and analyze an algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.

4. Let G be a connected undirected graph. Suppose we start with two coins on two arbitrarily chosen vertices of G, and we want to move the coins so that they lie on the same vertex using as few moves as possible. At every step, each coin *must* move to an adjacent vertex.

   (a) ⟨⟨*Lab*⟩⟩ Describe and analyze an algorithm to compute the minimum number of steps to reach a configuration where both coins are on the same vertex, or to report correctly that no such configuration is reachable. The input to your algorithm consists of the graph $G = (V, E)$ and two vertices $u, v \in V$ (which may or may not be distinct).

   (b) ⟨⟨*Lab*⟩⟩ Now suppose there are *forty-two* coins. Describe and analyze an algorithm to determine whether it is possible to move all 42 coins to the same vertex. Again, *every* coin must move at *every* step. The input to your algorithm consists of the graph $G = (V, E)$ and an array of 42 vertices (which may or may not be distinct). For full credit, your algorithm should run in $O(V + E)$ time.

5. A graph (V, E) is bipartite if the vertices V can be partitioned into two subsets L and R, such that every edge has one vertex in L and the other in R.

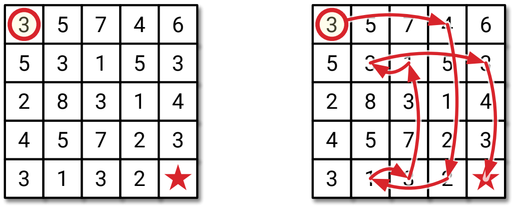
   (a) Prove that every tree is a bipartite graph.

   (b) Describe and analyze an efficient algorithm that determines whether a given undirected graph is bipartite.

6. ⟨⟨*F14, S18*⟩⟩ A ***number maze*** is an $n \times n$ grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner. On each turn,

you are allowed to move the token up, down, left, or right; the distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three steps down, three steps left, or three steps right. However, you are never allowed to move the token off the edge of the board.

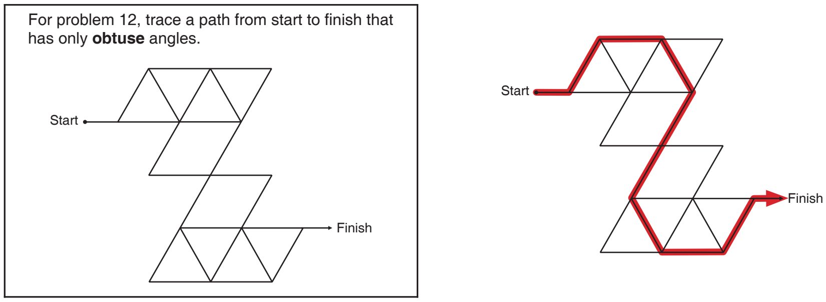
Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution. For example, given the maze shown below, your algorithm would return the number 8.



3	5	7	4	6
5	3	1	5	3
2	8	3	1	4
4	5	7	2	3
3	1	3	2	★

A 5 × 5 number maze that can be solved in eight moves.

7. ⟨⟨*F16*⟩⟩ The following puzzle appeared in my daughter's math workbook several years ago.[1] (I've put the solution on the right so you don't waste time solving it during the exam.)



For problem 12, trace a path from start to finish that has only **obtuse** angles.

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

You are given a connected undirected graph G, whose vertices are points in the plane and whose edges are line segments. Edges do not intersect, except at their endpoints. For example, a drawing of the letter X would have five vertices and four edges; the maze above has 17 vertices and 26 edges. You are also given two vertices Start and Finish.

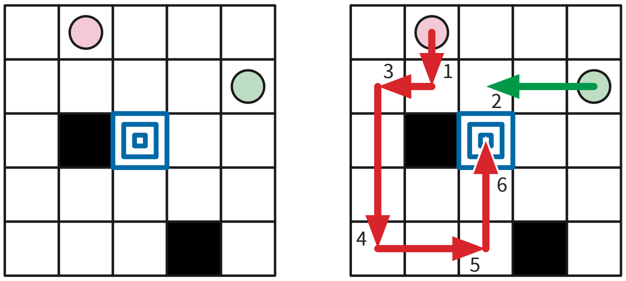
Your algorithm should return TRUE if G contains a walk from Start to Finish that has only obtuse angles, and FALSE otherwise. Formally, a walk through G is valid if $\pi/2 < \angle uvw \leq \pi$ for every pair of consecutive edges $u \to v \to w$ in the walk. Assume you have a subroutine that can determine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

8. The famous puzzle-maker Kaniel the Dane invented a solitaire game played with two tokens on an $n \times n$ square grid. Some squares of the grid are marked as *obstacles*, and one

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

grid square is marked as the *target*. In each turn, the player must move one of the tokens from is current position *as far as possible* upward, downward, right, or left, stopping just before the token hits (1) the edge of the board, (2) an obstacle square, or (3) the other token. The goal is to move either of the tokens onto the target square.

For example, in the instance below, we move the red token down until it hits the obstacle, then move the green token left until it hits the red token, and then move the red token left, down, right, and up. In the last move, the red token stops at the target *because* the green token is on the next square above.



An instance of the Kaniel Dane puzzle that can be solved in six moves.
Circles indicate the initial token positions; black squares are obstacles; the center square is the target.

Describe and analyze an algorithm to determine whether an instance of this puzzle is solvable. Your input consist of the integer n, a list of obstacle locations, the target location, and the initial locations of the tokens. The output of your algorithm is a single boolean: TRUE if the given puzzle is solvable and FALSE otherwise. The running time of your algorithm should be a small polynomial in n. *[Hint: Don't forget about the time required to construct the graph!]*

9. ⟨⟨*F16*⟩⟩ Suppose you have a collection of n lockboxes and m gold keys. Each key unlocks *at most* one box; however, each box might be unlocked by one key, by multiple keys, or by no keys at all. There are only two ways to open each box once it is locked: Unlock it properly (which requires having a matching key in your hand), or smash it to bits with a hammer.

   Your baby brother, who loves playing with shiny objects, has somehow managed to lock all your keys inside the boxes! Luckily, your home security system recorded everything, so you know exactly which keys (if any) are inside each box. You need to get all the keys back out of the boxes, because they are made of gold. Clearly you have to smash at least one box.

   (a) Your baby brother has found the hammer and is eagerly eyeing one of the boxes. Describe and analyze an algorithm to determine if smashing the box your brother has chosen would allow you to retrieve all m keys.

   (b) Describe and analyze an algorithm to compute the minimum number of boxes that must be smashed to retrieve all the keys. *[Hint: This subproblem should really be in the next section.]*

Depth-First Search, Dags, Strong Connectivity

1. ⟨⟨*Lab*⟩⟩ Inspired by an earlier question, you decided to organize a Snakes and Ladders competition with n participants. In this competition, each game of Snakes and Ladders involves three players. After the game is finished, they are ranked first, second and third. Each player may be involved in any (non-negative) number of games, and the number needs not be equal among players.

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

   You are given the list of players involved and the ranking in each of the m games. Describe and analyze an algorithm to produce an overall ranking of the n players that satisfies the condition, or correctly reports that it is impossible.

2. Let G be a directed acyclic graph with a unique source s and a unique sink t.

   (a) A *Hamiltonian path* in G is a directed path in G that contains every vertex in G. Describe an algorithm to determine whether G has a Hamiltonian path.

   (b) Suppose the vertices of G have weights. Describe an efficient algorithm to find the path from s to t with maximum total weight.

   (c) Suppose we are also given an integer ℓ. Describe an efficient algorithm to find the maximum-weight path from s to t, such that the path contains at most ℓ edges. (Assume there is at least one such path.)

   (d) Suppose several vertices in G are marked *essential*, and we are given an integer k. Design an efficient algorithm to determine whether there is a path from s to t that passes through at least k essential vertices.

   (e) Suppose the vertices of G have integer labels, where $label(s) = -\infty$ and $label(t) = \infty$. Describe an algorithm to find the path from s to t with the maximum number of edges, such that the vertex labels define an increasing sequence.

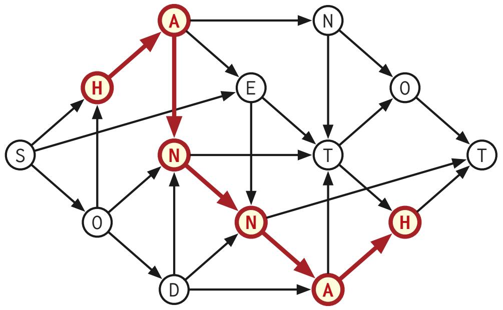
   (f) Describe an algorithm to compute the number of distinct paths from s to t in G. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

3. Suppose you are given a directed acyclic graph G whose nodes represent jobs and whose edges represent *precedence constraints*: Each edge $u \to v$ indicates that job u must be completed before job v begins. Each node v stores a non-negative number $v.duration$ indicating the time required to execute job v. All jobs are executed in parallel; any job can start or end while any number of other jobs are executing, provided all the precedence constraints are satisfied. You'd like to get all these jobs done as quickly as possible.

   Describe an algorithm to determine, for every vertex v in G, the earliest time that job v can ***begin***, assuming the first job starts at time 0 and no precedence constraints are violated. Your algorithm should record the answer for each vertex v in a new field $v.earliest$.

4. Let G be a directed acyclic graph whose vertices have labels from some fixed alphabet. Any directed path in G has a label, which is a string obtained by concatenating the labels of its vertices. Recall that a *palindrome* is a string that is equal to its reversal.

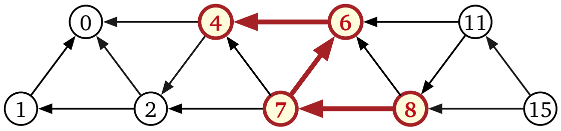
   Describe and analyze an algorithm to find the length of the longest palindrome that is the label of a path in G. For example, given the dag below, your algorithm should return the integer 6, which is the length of the palindrome HANNAH.



5. ⟨⟨*S18*⟩⟩ Let G be a ***directed*** graph, where every vertex v has an associated height $h(v)$, and for every edge $u \to v$ we have the inequality $h(u) > h(v)$. Assume all heights are distinct. The *span* of a path from u to v is the height difference $h(u) - h(v)$.

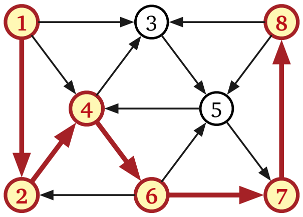
   Describe and analyze an algorithm to find the ***minimum span*** of a path in G with ***at least*** k edges. Your input consists of the graph G, the vertex heights $h(\cdot)$, and the integer k. Report the running time of your algorithm as a function of V, E, and k.

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



6. ⟨⟨*S18*⟩⟩ Let G be an arbitrary (*not* necessarily acyclic) directed graph in which every vertex v has an integer label $\ell(v)$. Describe an algorithm to find the longest directed path in G whose vertex labels define an increasing sequence. Assume all labels are distinct.

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



Shortest Paths

1. Suppose you are given a directed graph G with weighted edges and a vertex s of G.

   (a) ⟨⟨*F14*⟩⟩ Suppose every vertex $v \neq s$ stores a pointer $pred(v)$ to another vertex in G. Describe and analyze an algorithm to determine whether these predecessor pointers correctly define a single-source shortest path tree rooted at s.

   (b) Suppose every vertex v stores a finite real value $dist(v)$. (In particular, $dist(v)$ is never equal to ∞ or $-\infty$.) Describe and analyze an algorithm to determine whether these real values are correct shortest-path distances from s.

   Do ***not*** assume that G has no negative cycles.

2. ⟨⟨*F14*⟩⟩ Suppose we are given an undirected graph G in which every *vertex* has a positive weight.

   (a) Describe and analyze an algorithm to find a *spanning tree* of G with minimum total weight. (The total weight of a spanning tree is the sum of the weights of its vertices.)

   (b) Describe and analyze an algorithm to find a *path* in G from one given vertex s to another given vertex t with minimum total weight. (The total weight of a path is the sum of the weights of its vertices.)

3. ⟨⟨*S14, S18, Lab*⟩⟩ You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cites that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving toward each other. But where *exactly* should you meet?

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

   Describe and analyze an algorithm to find the target vertex t that allows you and your friend to meet as quickly as possible.

4. ⟨⟨*F16*⟩⟩ There are n galaxies connected by m intergalactic teleport-ways. Each teleport-way joins two galaxies and can be traversed in both directions. Also, each teleport-way uv has an associated cost of $c(uv)$ galactic credits, for some positive integer $c(uv)$. The same teleport-way can be used multiple times in either direction, but the same toll must be paid every time it is used.

   Judy wants to travel from galaxy s to galaxy t, but teleportation is rather unpleasant, so she wants to minimize the number of times she has to teleport. However, she also wants the total cost to be a multiple of 10 galactic credits, because carrying small change is annoying.

   Describe and analyze an algorithm to compute the minimum number of times Judy must teleport to travel from galaxy s to galaxy t so that the total cost of all teleports is an integer multiple of 10 galactic credits. Your input is a graph $G = (V, E)$ whose vertices are galaxies and whose edges are teleport-ways; every edge uv in G stores the corresponding cost $c(uv)$.

*[Hint: This is **not** the same Intergalactic Judy problem that you saw in lab.]*

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

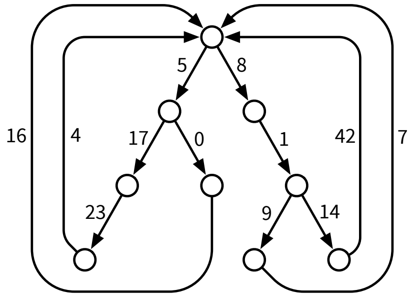


Figure 2. A looped tree.

(a) How much time would Dijkstra's algorithm require to compute the shortest path from one vertex s to another vertex t in a looped tree with n nodes?

(b) Describe and analyze a faster algorithm.

6. ⟨⟨*F17, Lab*⟩⟩ Suppose you are given a directed graph G with weighted edges, where *exactly one* edge has negative weight and all other edge weights are positive, along with two vertices s and t. Describe and analyze an algorithm that either computes a shortest path in G from s to t, or reports correctly that the G contains a negative cycle. (As always, faster algorithms are worth more points.)

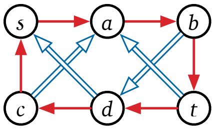
7. When there is more than one shortest path from one node s to another node t, it is often convenient to choose a shortest path with the fewest edges; call this the ***best*** path from s to t. Suppose we are given a directed graph G with positive edge weights and a source vertex s in G. Describe and analyze an algorithm to compute best paths in G from s to every other vertex.

8. After graduating you accept a job with Aerophobes-Я-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of all the flights on the planet.

Suppose one of your customers wants to fly from city X to city Y. Describe an algorithm to find a sequence of flights that minimizes the *total time in transit*—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights.

9. ⟨⟨*S18*⟩⟩ Suppose you are given a directed graph G where some edges are red and the remaining edges are blue. Describe an algorithm to find the shortest walk in G from one vertex s to another vertex t in which no three consecutive edges have the same color. That is, if the walk contains two red edges in a row, the next edge must be blue, and if the walk contains two blue edges in a row, the next edge must be red.

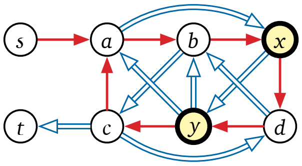
   For example, if you are given the graph below (where single arrows are red and double arrows are blue), your algorithm should return the integer 7, because the shortest legal walk from s to t is $s \to a \to b \Rightarrow d \to c \Rightarrow a \to b \to c$.



10. ⟨⟨*S18*⟩⟩ Suppose you are given a directed graph G in which every edge is either red or blue, and a subset of the vertices are marked as *special*. A walk in G is *legal* if color changes happen only at special vertices. That is, for any two consecutive edges $u \to v \to w$ in a legal walk, if the edges $u \to v$ and $v \to w$ have different colors, the intermediate vertex v must be special.

    Describe and analyze an algorithm that either returns the length of the shortest legal walk from vertex s to vertex t, or correctly reports that no such walk exists.[2]

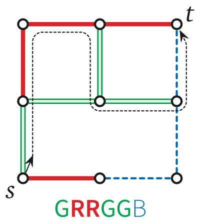
    For example, if you are given the following graph below as input (where single arrows are red, double arrows are blue), with special vertices x and y, your algorithm should return the integer 8, which is the length of the shortest legal walk $s \to x \to a \to b \to x \Rightarrow y \Rightarrow b \Rightarrow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

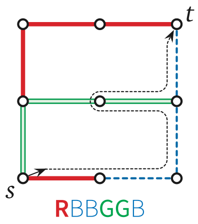


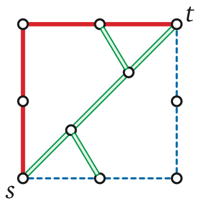
11. Suppose you are given an undirected graph G in which every edge is either red, green, or blue, along with two vertices s and t. Call a walk from s to t *awesome* if the walk does not contain three consecutive edges with the same color.

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

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







12. ⟨⟨*S18*⟩⟩ Let G be a directed graph with weighted edges, in which every vertex is colored either red, green, or blue. Describe and analyze an algorithm to compute the length of the shortest walk in G that starts at a red vertex, then visits any number of vertices of any color, then visits a green vertex, then visits any number of vertices of any color, then visits a blue vertex, then visits any number of vertices of any color, and finally ends at a red vertex. Assume all edge weights are positive.

13. ⟨⟨*F19*⟩⟩ During her walk to work every morning, Rachel likes to buy a cappuccino at a local coffee shop, and a croissant at a local bakery. Her home town has *lots* of coffee shops and lots of bakeries, but strangely never in the same building. Punctuality is not Rachel's strongest trait, so to avoid losing her job, she wants to follow the shortest possible route.

    Rachel has a map of her home town in the form of an undirected graph G, whose vertices represent intersections and whose edges represent roads between them. A subset of the vertices are marked as bakeries; another disjoint subset of vertices are marked as coffee shops. The graph has two special nodes s and t, which represent Rachel's home and work, respectively.

    Describe an algorithm that computes the shortest path in G from s to t that visits both a bakery and a coffee shop, or correctly reports that no such path exists.

14. ⟨⟨*F19*⟩⟩ As the days get shorter in winter, Eggsy Hutmacher is increasingly worried about his walk home from work. The city has recently been invaded by the notorious Antimilliner gang, whose members hang out on dark street corners and steal hats from unwary passers-by, and a gentleman is simply *not* seen out in public without a hat. The city council is slowly installing street lamps at intersections to deter the Antimilliners, whose uncovered faces can be easily identified in the light. Eggsy keeps k extra hats in his briefcase in case of theft or other millinery emergencies.

    Eggsy has a map of the city in the form of an undirected graph G, whose vertices represent intersections and whose edges represent streets between them. A subset of the vertices are marked to indicate that the corresponding intersections are lit. Every edge e has a non-negative length $\ell(e)$. The graph has two special nodes s and t, which represent Eggsy's work and home, respectively.

    Describe an algorithm that computes the shortest path in G from s to t that visits at most k unlit vertices, or correctly reports that no such path exists. Analyze your algorithm as a function of the parameters V, E, and k.

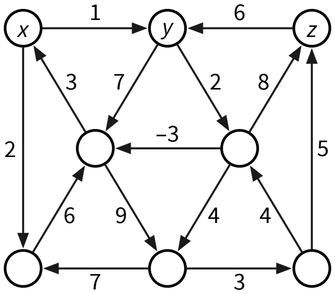
15. ⟨⟨*F19*⟩⟩ You and your friends are planning a hiking trip in Jellystone National Park over winter break. You have a map of the park's trails that lists all the scenic views in the park but also warns that certain trail segments have a high risk of bear encounters. To make the hike worthwhile, you 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 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 $S \cap B = \varnothing$. 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 $t \in V$.

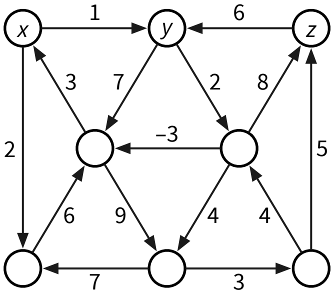
    Describe and analyze an algorithm to compute the shortest hike from your campsite s to the visitor center t that includes *at least* three scenic trail segments and *at most* one high-bear-risk trail segment. You may assume such a hike exists.

CS/ECE 374 A ✧ Fall 2021 Fake Midterm 2 Problem 1	Name:

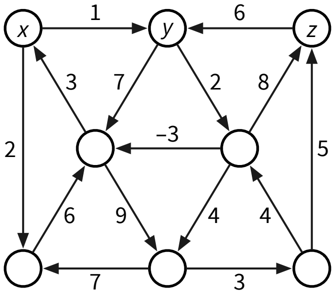
Clearly indicate the following structures in the directed graph below, or write NONE if the indicated structure does not exist. Don't be subtle; to indicate a collection of edges, draw a heavy black line along the entire length of each edge.



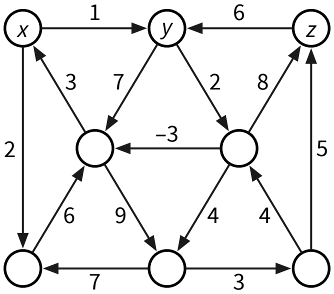
(a) A depth-first tree rooted at x.



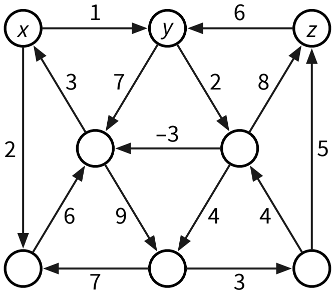
(b) A breadth-first tree rooted at y.



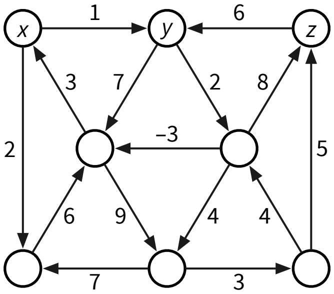
(c) A shortest-path tree rooted at z.



(d) The shortest directed cycle.



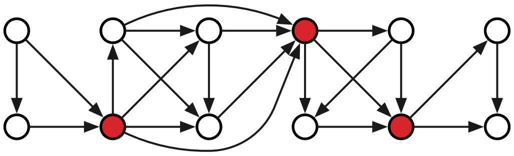
[scratch]



[scratch]

CS/ECE 374 A ✧ Fall 2021 **Fake Midterm 2 Problem 2**	Name:

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.

CS/ECE 374 A ✧ Fall 2021 **Fake Midterm 2 Problem 3**	Name:

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 $S \cap B = \varnothing$. 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.

CS/ECE 374 A ✧ Fall 2021 **Fake Midterm 2 Problem 4**	Name:

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, is the initial card values are $[4, 6, 1, 2]$, the game might proceed as follows:

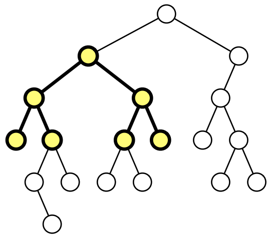
- You take the 4 on the left, leaving $[6, 1, 2]$.
- Elmo takes the 6 on the left, leaving $[1, 2]$.
- You take the 2 on the right, leaving $[1]$.
- Elmo takes the last 1, ending the game.
- You took $4 + 2 = 6$ points, and Elmo took $6 + 1 = 7$ points, so Elmo wins!

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.

CS/ECE 374 A ✧ Fall 2021 **Fake Midterm 2 Problem 5**	Name:

For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *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 ***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.



CS/ECE 374 A ✧ Fall 2021

Midterm 2

November 8, 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 $T(n) = 2T(n/3) + O(\sqrt{n})$.

   (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.)

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

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:

SF	\$800	\$200	\$500	\$400	\$1200
NY	\$300	\$900	\$700	\$2000	\$200

   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*.

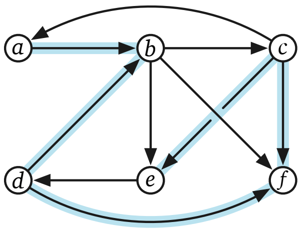
4. Suppose you are given an array $A[1\,..\,n]$ of integers such that $A[i] + A[i+1]$ is even for *exactly one* index i. In other words, the elements of A alternate between even and odd, except for exactly one adjacent pair that are either both even or both odd.

   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 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 $a{\to}b{\leftarrow}d{\to}f{\leftarrow}c{\to}e$. 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 $T(n) = 4T(n/2) + O(n^2)$.

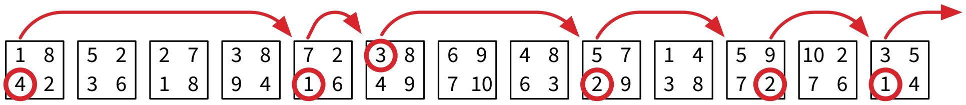
   (d) Draw a directed acyclic graph with at most ten vertices, exactly one source, exactly one sink, and more than one topological order.

   (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 $Huh(1, n)$.

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

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 quadhopper puzzle that allows six moves. (This is ***not*** the longest legal sequence of moves.)

   (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 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 $u \to v$ has a non-negative weight $w(u \to v)$, 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 $w(u \to v)$ for every edge $u \to v$.

CS/ECE 374 A ✧ Fall 2021

Final Exam Study Questions

This is a “core dump” of potential questions for the final exam. This should give you a good idea of the *types* of questions that we will ask on the exam. In particular, there will be a series of True/False or short-answer questions—but the actual exam questions may or may not appear in this handout. This list intentionally includes a few questions that are too long or difficult for exam conditions; these are indicated with a *star.

Don’t forget to review the study problems for Midterms 1 and 2; the final exam is cumulative!

How to Use These Problems

Solving every problem in this handout is ***not*** the best way to study for the exam. Memorizing the solutions to every problem in this handout is the ***absolute worst*** way to study for the exam.

What we recommend instead is to work on a *sample* of the problems. Choose one or two problems at random from each section and try to solve them from scratch under exam conditions—by yourself, in a quiet room, with a 30-minute timer, *without* your notes, *without* the internet, and if possible, even without your cheat sheet. If you’re comfortable solving a few problems in a particular section, you’re probably ready for that type of problem on the exam. Move on to the next section.

Discussing problems with other people (in your study groups, in the review sessions, in office hours, or on Piazza) and/or looking up old solutions can be *extremely* helpful, but ***only after*** you have (1) made a good-faith effort to solve the problem on your own, and (2) you have either a candidate solution or some idea about where you’re getting stuck.

If you find yourself getting stuck on a particular type of problem, try to figure out *why* you’re stuck. Do you understand the problem statement? Are you stuck on choosing the right high-level approach? Are you stuck on the technical details? Or are you struggling to express your ideas clearly? (We *strongly* recommend writing solutions that follow the homework grading rubrics bullet-by-bullet.)

Similarly, if feedback from other people suggests that your solutions to a particular type of problem are incorrect or incomplete, try to figure out what you missed. For NP-hardness proofs: Are you choosing a good problem to reduce from? Are you reducing in the correct direction? Are you designing your reduction with both good instances and bad instances in mind? You’re not trying *solve* the problem, are you? For undecidability proofs: Does the problem have the right structure to apply Rice’s theorem? If you are arguing by reduction, are you reducing in the correct direction? You’re not using *pronouns*, are you?

Remember that your goal is *not* merely to “understand” the solution to any particular problem, but to become more comfortable with solving a certain *type* of problem on your own. **“Understanding” is a trap; aim for mastery.** If you can identify specific steps that you find problematic, read more *about those steps*, focus your practice *on those steps*, and try to find helpful information *about those steps* to write on your cheat sheet. Then work on the next problem!

True or False? (All from previous final exams)

For each statement below, write "YES" or "True" if the statement is ***always*** true and "NO" or "False" otherwise, and give a brief (at most one short sentence) explanation of your answer. **Assume *P* ≠ *NP*.** If there is any other ambiguity or uncertainty about an answer, write "NO" or "False". 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.

---

1. Which of the following are clear English specifications of a recursive function that could possibly be used to compute the edit distance between two strings $A[1..n]$ and $B[1..n]$?

   (a) $\mathit{Edit}(i, j)$ is the answer for i and j.

   (b) $\mathit{Edit}(i, j)$ is the edit distance between $A[i]$ and $B[j]$.

   (c) $$\mathit{Edit}[i,j] = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \mathit{Edit}[i-1, j-1] & \text{if } A[i] = B[j] \\ \max\left\{\begin{array}{c} 1 + \mathit{Edit}[i, j-1] \\ 1 + \mathit{Edit}[i-1, j] \\ 1 + \mathit{Edit}[i-1, j-1] \end{array}\right\} & \text{otherwise} \end{cases}$$

   (d) $\mathit{Edit}[1..n, 1..n]$ stores the edit distances for all prefixes.

   (e) $\mathit{Edit}(i, j)$ is the edit distance between $A[i..n]$ and $B[j..n]$.

   (f) $\mathit{Edit}[i, j]$ is the value stored at row i and column j of the table.

   (g) $\mathit{Edit}(i, j)$ is the edit distance between the last i characters of A and the last j characters of B.

   (h) $\mathit{Edit}(i, j)$ is the edit distance when i and j are the current characters in A and B.

   (i) Iterate through both strings and update $\mathit{Edit}[\cdot, \cdot]$ at each character.

   (j) $\mathit{Edit}(i, j, k, l)$ is the edit distance between substrings $A[i..j]$ and $B[k..l]$.

   (k) *[I don't need an English description; my pseudocode is clear enough!]*

---

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

   (a) $E \neq \varnothing$.

   (b) Given the graph G as input, Floyd-Warshall runs in $O(E^3)$ time.

(c) If G has at least one source and at least one sink, then G is a dag.

(d) We can compute a spanning tree of G using whatever-first search.

(e) If the edges of G are weighted, we can compute the shortest path from any node s to any node t in $O(E \log V)$ time using Dijkstra's algorithm.

---

3. Which of the following statements are true for ***every*** language $L \subseteq \{0, 1\}^*$?

(a) L is non-empty.

(b) L is infinite.

(c) L contains the empty string ε.

(d) L^* is infinite.

(e) L^* is regular.

(f) L is accepted by some DFA if and only if L is accepted by some NFA.

(g) L is described by some regular expression if and only if L is rejected by some NFA.

(h) L is accepted by some DFA with 42 states if and only if L is accepted by some NFA with 42 states.

(i) If L is decidable, then L is infinite.

(j) If L is not decidable, then L is infinite.

(k) If L is not regular, then L is undecidable.

(l) If L has an infinite fooling set, then L is undecidable.

(m) If L has a finite fooling set, then L is decidable.

(n) If L is the union of two regular languages, then its complement $\overline{L}$ is regular.

(o) If L is the union of two regular languages, then its complement $\overline{L}$ is context-free.

(p) If L is the union of two decidable languages, then L is decidable.

(q) If L is the union of two undecidable languages, then L is undecidable.

(r) If $L \notin \mathrm{P}$, then L is not regular.

(s) L is decidable if and only if its complement $\overline{L}$ is undecidable.

(t) Both L and its complement $\overline{L}$ are decidable.

---

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

(a) L is non-empty.

(b) L is infinite.

(c) L contains the empty string ε.

(d) L^* is finite.

(e) L^* is not regular.

(f) L is not regular but L^* is regular.

(g) L is finite and L is undecidable.

(h) L is decidable but L^* is not decidable.

(i) L is not decidable but L^* is decidable.

(j) L is the union of two decidable languages, but L is not decidable.

(k) L is the union of two undecidable languages, but L is decidable.

(l) L is accepted by an NFA with 374 states, but L is not accepted by a DFA with 374 states.

(m) L is accepted by an DFA with 374 states, but L is not accepted by a NFA with 374 states.

(n) L is regular and $L \notin \mathrm{P}$.

(o) There is a Turing machine that accepts L.

(p) There is an algorithm to decide whether an arbitrary given Turing machine accepts L.

---

5. Which of the following languages over the alphabet $\{\mathtt{0}, \mathtt{1}\}$ are ***regular***?

(a) $\{\mathtt{0}^m\mathtt{1}^n \mid m \geq 0 \text{ and } n \geq 0\}$

(b) All strings with the same number of 0s and 1s

(c) Binary representations of all positive integers divisible by 17

(d) Binary representations of all prime numbers less than 10^{100}

(e) $\{ww \mid w \text{ is a palindrome}\}$

(f) $\{wxw \mid w \text{ is a palindrome and } x \in \{\mathtt{0}, \mathtt{1}\}^*\}$

(g) $\{\langle M\rangle \mid M \text{ accepts a regular language}\}$

(h) $\{\langle M\rangle \mid M \text{ accepts a finite number of non-palindromes}\}$

---

6. Which of the following languages/decision problems are ***decidable***?

(a) $\varnothing$

(b) $\{\mathtt{0}^n\mathtt{1}^{2n}\mathtt{0}^n\mathtt{1}^{2n} \mid n \geq 0\}$

(c) $\{ww \mid w \text{ is a palindrome}\}$

(d) $\{\langle M\rangle \mid M \text{ accepts } \langle M\rangle \bullet \langle M\rangle\}$

(e) $\{\langle M\rangle \mid M \text{ accepts a finite number of non-palindromes}\}$

(f) $\{\langle M\rangle \bullet w \mid M \text{ accepts } ww\}$

(g) $\{\langle M\rangle \bullet w \mid M \text{ accepts } ww \text{ after at most } |w|^2 \text{ steps}\}$

(h) Given an NFA N, is the language $L(N)$ infinite?

(i) CircuitSat

(j) Given an undirected graph G, does G contain a Hamiltonian cycle?

(k) Given encodings of two Turing machines M and M', is there a string w that is accepted by both M and M'?

---

7. Which of the following languages can be proved undecidable ***using Rice's Theorem***?

   (a) $\varnothing$

   (b) $\{\mathtt{0}^n\mathtt{1}^{2n}\mathtt{0}^n\mathtt{1}^{2n} \mid n \ge 0\}$

   (c) $\{ww \mid w \text{ is a palindrome}\}$

   (d) $\{\langle M\rangle \mid M \text{ accepts an infinite number of strings}\}$

   (e) $\{\langle M\rangle \mid M \text{ accepts a finite number of strings}\}$

   (f) $\{\langle M\rangle \mid M \text{ accepts either } \langle M\rangle \text{ or } \langle M\rangle^R\}$

   (g) $\{\langle M\rangle \mid M \text{ accepts both } \langle M\rangle \text{ and } \langle M\rangle^R\}$

   (h) $\{\langle M\rangle \mid M \text{ does not accept exactly 374 palindromes}\}$

   (i) $\{\langle M\rangle \mid M \text{ accepts some string } w \text{ after at most } |w|^2 \text{ steps}\}$

   (j) $\{\langle M\rangle \bullet w \mid M \text{ rejects } w \text{ after at most } |w|^2 \text{ steps}\}$

   (k) Given the encodings of two Turing machines M and M', is there a string w that is accepted by both M and M'?

---

8. Recall the halting language $\text{HALT} = \{\langle M\rangle \bullet w \mid M \text{ halts on input } w\}$. Which of the following statements about its complement $\overline{\text{HALT}} = \Sigma^* \setminus \text{HALT}$ are true?

   (a) $\overline{\text{HALT}}$ is empty.

   (b) $\overline{\text{HALT}}$ is regular.

   (c) $\overline{\text{HALT}}$ is infinite.

   (d) $\overline{\text{HALT}}$ is in NP.

   (e) $\overline{\text{HALT}}$ is decidable.

---

9. Suppose some language $A \in \{\mathtt{0},\mathtt{1}\}^*$ reduces to another language $B \in \{\mathtt{0},\mathtt{1}\}^*$. Which of the following statements ***must*** be true?

   (a) A Turing machine that recognizes A can be used to construct a Turing machine that recognizes B.

   (b) A is decidable.

   (c) If B is decidable then A is decidable.

   (d) If A is decidable then B is decidable.

   (e) If B is NP-hard then A is NP-hard.

   (f) If A has no polynomial-time algorithm then neither does B.

---

10. Suppose there is a ***polynomial-time*** reduction from problem A to problem B. Which of the following statements ***must*** be true?

    (a) Problem B is NP-hard.

    (b) A polynomial-time algorithm for B can be used to solve A in polynomial time.

    (c) If B has no polynomial-time algorithm then neither does A.

    (d) If A is NP-hard and B has a polynomial-time algorithm then $\mathrm{P} = \mathrm{NP}$.

    (e) If B is NP-hard then A is NP-hard.

    (f) If B is undecidable then A is undecidable.

---

11. Consider the following pair of languages:

    - $\textsc{HamPath} := \left\{G \mid G \text{ is an undirected graph with a Hamiltonian path}\right\}$
    - $\textsc{Connected} := \left\{G \mid G \text{ is a connected undirected graph}\right\}$

    (For concreteness, assume that in both of these languages, graphs are represented by their adjacency matrices.) Which of the following ***must*** be true, assuming P≠NP?

    (a) $\textsc{Connected} \in \mathrm{NP}$

    (b) $\textsc{HamPath} \in \mathrm{NP}$

    (c) $\textsc{HamPath}$ is decidable.

    (d) There is no polynomial-time reduction from $\textsc{HamPath}$ to $\textsc{Connected}$.

    (e) There is no polynomial-time reduction from $\textsc{Connected}$ to $\textsc{HamPath}$.

---

12. Consider the following pair of languages:

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

    (For concreteness, assume that in both of these languages, graphs are represented by their adjacency matrices.) Which of the following ***must*** be true, assuming P≠NP?

    (a) $\textsc{Acyclic} \in \mathrm{NP}$

    (b) $\textsc{Acyclic} \cap \textsc{DirHamPath} \in \mathrm{P}$

    (c) $\textsc{DirHamPath}$ is decidable.

    (d) There is a polynomial-time reduction from $\textsc{DirHamPath}$ to $\textsc{Acyclic}$.

    (e) There is a polynomial-time reduction from $\textsc{Acyclic}$ to $\textsc{DirHamPath}$.

---

13. Consider the following pair of languages:

    - $\textsc{3Color} := \left\{G \mid G \text{ is a 3-colorable undirected graph}\right\}$
    - $\textsc{Tree} := \left\{G \mid G \text{ is a connected acyclic undirected graph}\right\}$

(For concreteness, assume that in both of these languages, graphs are represented by adjacency matrices.) Which of the following ***must*** be true, assuming P≠NP?

(a) Tree is NP-hard.

(b) Tree ∩ 3Color ∈ P

(c) 3Color is undecidable.

(d) There is a polynomial-time reduction from 3Color to Tree.

(e) There is a polynomial-time reduction from Tree to 3Color.

NP-hardness

1. A boolean formula is in *disjunctive normal form* (or *DNF*) if it consists of a *disjunction* (Or) or several *terms*, each of which is the conjunction (And) of one or more literals. For example, the formula
$$(\overline{x} \wedge y \wedge \overline{z}) \vee (y \wedge z) \vee (x \wedge \overline{y} \wedge \overline{z})$$
is in disjunctive normal form. DNF-Sat asks, given a boolean formula in disjunctive normal form, whether that formula is satisfiable.

   (a) Describe a polynomial-time algorithm to solve DNF-Sat.

   (b) What is the error in the following argument that P=NP?

   > *Suppose we are given a boolean formula in conjunctive normal form with at most three literals per clause, and we want to know if it is satisfiable. We can use the distributive law to construct an equivalent formula in disjunctive normal form. For example,*
   > $$(x \vee y \vee \overline{z}) \wedge (\overline{x} \vee \overline{y}) \iff (x \wedge \overline{y}) \vee (y \wedge \overline{x}) \vee (\overline{z} \wedge \overline{x}) \vee (\overline{z} \wedge \overline{y})$$
   > *Now we can use the algorithm from part (a) to determine, in polynomial time, whether the resulting DNF formula is satisfiable. We have just solved* 3Sat *in polynomial time. Since* 3Sat *is NP-hard, we must conclude that P=NP!*

2. A ***relaxed 3-coloring*** of a graph G assigns each vertex of G one of three colors (for example, red, green, and blue), such that ***at most one*** edge in G has both endpoints the same color.

   (a) Give an example of a graph that has a relaxed 3-coloring, but does not have a proper 3-coloring (where every edge has endpoints of different colors).

   (b) ***Prove*** that it is NP-hard to determine whether a given graph has a relaxed 3-coloring.

3. An ***ultra-Hamiltonian tour*** in G is a closed walk W that visits every vertex of G exactly once, except for *at most one* vertex that W visits more than once.

   (a) Give an example of a graph that contains a ultra-Hamiltonian tour, but does not contain a Hamiltonian cycle (which visits every vertex exactly once).

   (b) ***Prove*** that it is NP-hard to determine whether a given graph contains a ultra-Hamiltonian tour.

4. An ***infra-Hamiltonian cycle*** in G is a closed walk W that visits every vertex of G exactly once, except for *at most one* vertex that W does not visit at all.

   (a) Give an example of a graph that contains a infra-Hamiltonian cycle, but does not contain a Hamiltonian cycle (which visits every vertex exactly once).

   (b) ***Prove*** that it is NP-hard to determine whether a given graph contains a infra-Hamiltonian cycle.

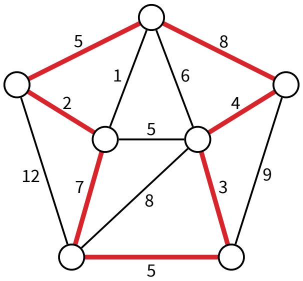
5. A ***quasi-satisfying assignment*** for a 3CNF boolean formula Φ is an assignment of truth values to the variables such that *at most one* clause in Φ does not contain a true literal. ***Prove*** that it is NP-hard to determine whether a given 3CNF boolean formula has a quasi-satisfying assignment.

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

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

8. A subset S of vertices in an undirected graph G is ***almost independent*** if at most 374 edges in G have both endpoints in S. Prove that finding the size of the largest almost-independent set of vertices in a given undirected graph is NP-hard.

9. Let G be an undirected graph with weighted edges. A ***heavy Hamiltonian cycle*** is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is more than half of the total weight of all edges in G. Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-hard.



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

10. (a) A ***tonian path*** in a graph G is a path that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian path is NP-hard.

    (b) A ***tonian cycle*** in a graph G is a cycle that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian cycle is NP-hard. *[Hint: Use part (a). Or not.]*

11. Prove that the following variants of SAT is NP-hard. *[Hint: Describe reductions from 3SAT.]*

    (a) Given a boolean formula Φ in conjunctive normal form, where *each variable appears in at most three clauses*, determine whether Φ has a satisfying assignment. *[Hint: First consider the variant where each variable appears in at most **five** clauses.]*

    (b) Given a boolean formula Φ in conjunctive normal form *and given one satisfying assignment for* Φ, determine whether Φ has at least one other satisfying assignment.

12. Jerry Springer and Maury Povich have decided not to compete with each other over scheduling guests during the next talk-show season. There is only one set of Weird People who either host would consider having on their show. The hosts want to divide the Weird People into two disjoint subsets: those to appear on Jerry's show, and those to appear on Maury's show. (Neither wants to "recycle" a guest that appeared on the other's show.)

Both Jerry and Maury have preferences about which Weird People they are particularly interested in. For example, Jerry wants at least one guest who fits the description "was abducted by a flying saucer". Thus, on his list of preferences, he writes "w_1 or w_3 or w_{45}", since weird people numbered 1, 3, and 45 are the only ones who fit that description. Jerry has other preferences as well, so he lists those also. Similarly, Maury might like to include at least one guest who "really enjoys Rice's theorem". Each potential guest may fall into any number of different categories, such as the person who enjoys Rice's theorem more than their involuntary flying-saucer voyage.

Jerry and Maury each prepare a list reflecting all of their preferences. Each list contains a collection of statements of the form "(w_i or w_j or w_k)". Your task is to prove that it is NP-hard to find an assignment of weird guests to the two shows that satisfies all of Jerry's preferences and all of Maury's preferences.

(a) The problem NoMixedClauses3Sat is the special case of 3Sat where the input formula cannot contain a clause with both a negated variable and a non-negated variable. Prove that NoMixedClauses3Sat is NP-hard. *[Hint: Reduce from the standard 3Sat problem.]*

(b) Describe a polynomial-time reduction from NoMixedClauses3Sat to 3Sat.

13. The president of Sham-Poobanana University is planning An Unofficial St. Brigid's Day party for the university staff.[1] His staff has a hierarchical structure; that is, the supervisor relation forms a directed, acyclic graph, with the president as the only source, and with an edge from person i to person j in the graph if and only if person i is an immediate supervisor of person j. (Many staff members have multiple positions, and thus have several immediate supervisors.) In order to make the party fun for all guests, the president wants to ensure that if a person i attends, then none of i's immediate supervisors can attend.

By mining each staff member's email and social media accounts, Sham-Poobanana University Human Resources has determined a "party-hound" rating for each staff member, which is a non-negative real number reflecting how likely it is that the person will leave the party wearing a monkey suit and a lampshade.

Show that it is NP-hard to determine a guest-list that *maximizes* the sum of the party-hound ratings of all invited guests, subject to the supervisor constraint.

[Hint: This problem can be solved in polynomial time when the input graph is a tree!]

14. Prove that the following problem (which we call Match) is NP-hard. The input is a finite set S of strings, all of the same length n, over the alphabet $\{0, 1, 2\}$. The problem is to determine whether there is a string $w \in \{0, 1\}^n$ such that for every string $s \in S$, the strings s and w have the same symbol in at least one position.

For example, given the set $S = \{01220, 21110, 21120, 00211, 11101\}$, the correct output is True, because the string $w = 01001$ matches the first three strings of S in the second position, and matches the last two strings of S in the last position. On the other hand, given the set $S = \{00, 11, 01, 10\}$, the correct output is False.

[Hint: Describe a reduction from SAT (or 3SAT)]

[1] As I'm sure you already know, St. Brigid of Kildare is one of the patron saints of Ireland, chicken and dairy farmers, and academics.

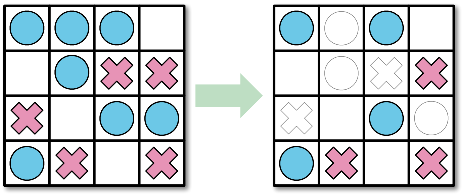
15. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions:

    (1) Every row contains at least one stone.

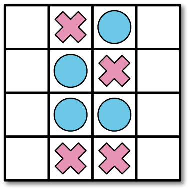
    (2) No column contains stones of both colors.

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

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



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

16. To celebrate the end of the semester, Professor Jarling wants to treat himself to an ice-cream cone at the *Polynomial House of Flavors*. For a fixed price, he can build a cone with as many scoops as he'd like. Because he has good balance (and because we want this problem to work out), Prof. Jarling can balance any number of scoops on top of the cone without it tipping over. He plans to eat the ice cream one scoop at a time, from top to bottom, and doesn't want more than one scoop of any flavor.

    However, he realizes that eating a scoop of bubblegum ice cream immediately after the scoop of potatoes-and-gravy ice cream would be unpalatable; these two flavors clearly should not be placed next to each other in the stack. He has other similar constraints; certain pairs of flavors cannot be adjacent in the stack.

    He'd like to get as much ice cream as he can for the one fee by building the tallest cone possible that meets his flavor-incompatibility constraints. Prove that Prof. Jarling's problem is NP-hard.

17. Prove that the following problems are NP-hard.

    (a) Given an undirected graph G, does G contain a simple path that visits all but 17 vertices?

    (b) Given an undirected graph G, does G have a spanning tree in which every node has degree at most 23?

    (c) Given an undirected graph G, does G have a spanning tree with at most 42 leaves?

18. Prove that the following problems are NP-hard.

    (a) Given an undirected graph G, is it possible to color the vertices of G with three different colors, so that at most 31337 edges have both endpoints the same color?

(b) Given an undirected graph G, is it possible to color the vertices of G with three different colors, so that each vertex has at most 8675309 neighbors with the same color?

19. At the end of every semester, Jeff needs to solve the following ExamDesign problem. He has a list of problems, and he knows for each problem which students will *really enjoy* that problem. He needs to choose a subset of problems for the exam such that for each student in the class, the exam includes at least one question that student will really enjoy. On the other hand, he does not want to spend the entire summer grading an exam with dozens of questions, so the exam must also contain as few questions as possible. Prove that the ExamDesign problem is NP-hard.

20. Which of the following results would resolve the P vs. NP question? Justify each answer with a short sentence or two.

(a) The construction of a polynomial time algorithm for some problem in NP.

(b) A polynomial-time reduction from 3Sat to the language $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$.

(c) A polynomial-time reduction from $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ to 3Sat.

(d) A polynomial-time reduction from 3Color to MinVertexCover.

(e) The construction of a nondeterministic Turing machine that cannot be simulated by any deterministic Turing machine with the same running time.

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. **The final exam will include a copy of this list.**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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?

Turing Machines and Undecidability

The only undecidability questions on this semester's final exam will be True/False or short-answer, but the following problems might still be useful to build intuition.

For each of the following languages, either ***sketch*** an algorithm to decide that language or ***prove*** that the language is undecidable, using a diagonalization argument, a reduction argument, Rice's theorem, closure properties, or some combination of the above. Recall that w^R denotes the reversal of string w.

1. $\varnothing$
2. $\{\mathtt{0}^n\mathtt{1}^n\mathtt{2}^n \mid n \geq 0\}$
3. $\{A \in \{\mathtt{0},\mathtt{1}\}^{n\times n} \mid n \geq 0 \text{ and } A \text{ is the adjacency matrix of a dag with } n \text{ vertices}\}$
4. $\{A \in \{\mathtt{0},\mathtt{1}\}^{n\times n} \mid n \geq 0 \text{ and } A \text{ is the adjacency matrix of a 3-colorable graph with } n \text{ vertices}\}$
5. $\{\langle M\rangle \mid M \text{ accepts } \langle M\rangle^R\}$
6. $\{\langle M\rangle \mid M \text{ accepts } \langle M\rangle^R\} \cap \{\langle M\rangle \mid M \text{ rejects } \langle M\rangle^R\}$
7. $\{\langle M\rangle\mathtt{\#}w \mid M \text{ accepts } ww^R\}$
8. $\{\langle M\rangle \mid M \text{ accepts } \mathtt{RICESTHEOREM}\}$
9. $\{\langle M\rangle \mid M \text{ rejects } \mathtt{RICESTHEOREM}\}$
10. $\{\langle M\rangle \mid M \text{ accepts at least one palindrome}\}$
11. $\Sigma^* \setminus \{\langle M\rangle \mid M \text{ accepts at least one palindrome}\}$
12. $\{\langle M\rangle \mid M \text{ rejects at least one palindrome}\}$
13. $\{\langle M\rangle \mid M \text{ accepts exactly one string of length } \ell, \text{ for each integer } \ell \geq 0\}$
14. $\{\langle M\rangle \mid \textsc{Accept}(M) \text{ has an infinite fooling set}\}$
15. $\{\langle M\rangle\mathtt{\#}\langle M'\rangle \mid \textsc{Accept}(M) \cap \textsc{Accept}(M') \neq \varnothing\}$
16. $\{\langle M\rangle\mathtt{\#}\langle M'\rangle \mid \textsc{Accept}(M) \oplus \textsc{Reject}(M') \neq \varnothing\}$ — Here ⊕ means exclusive-or.

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

$$\begin{aligned}
\textsc{SelfReject} &:= \left\{\langle M\rangle \;\middle|\; M \text{ rejects } \langle M\rangle\right\}\\
\textsc{SelfAccept} &:= \left\{\langle M\rangle \;\middle|\; M \text{ accepts } \langle M\rangle\right\}\\
\textsc{SelfHalt} &:= \left\{\langle M\rangle \;\middle|\; M \text{ halts on } \langle M\rangle\right\}\\
\textsc{SelfDiverge} &:= \left\{\langle M\rangle \;\middle|\; M \text{ does not halt on } \langle M\rangle\right\}
\end{aligned}$$

$$\begin{aligned}
\textsc{Reject} &:= \left\{\langle M\rangle\#w \;\middle|\; M \text{ rejects } w\right\}\\
\textsc{Accept} &:= \left\{\langle M\rangle\#w \;\middle|\; M \text{ accepts } w\right\}\\
\textsc{Halt} &:= \left\{\langle M\rangle\#w \;\middle|\; M \text{ halts on } w\right\}\\
\textsc{Diverge} &:= \left\{\langle M\rangle\#w \;\middle|\; M \text{ does not halt on } w\right\}
\end{aligned}$$

$$\begin{aligned}
\textsc{NeverReject} &:= \left\{\langle M\rangle \;\middle|\; \textsc{Reject}(M) = \varnothing\right\}\\
\textsc{NeverAccept} &:= \left\{\langle M\rangle \;\middle|\; \textsc{Accept}(M) = \varnothing\right\}\\
\textsc{NeverHalt} &:= \left\{\langle M\rangle \;\middle|\; \textsc{Halt}(M) = \varnothing\right\}\\
\textsc{NeverDiverge} &:= \left\{\langle M\rangle \;\middle|\; \textsc{Diverge}(M) = \varnothing\right\}
\end{aligned}$$

CS/ECE 374 A ✧ Fall 2021

Final Exam

December 15, 2021

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 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 $c \in Z^d$, 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 $T(n) = \mathbf{4}T(n/\mathbf{2}) + O(n^2)$ is $T(n) = O(n^2)$.

(b) The solution to the recurrence $T(n) = \mathbf{2}T(n/\mathbf{4}) + O(n^2)$ 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:

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

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 ***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:

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

(For concreteness, assume that in both of these languages, graphs are represented by their adjacency matrices.) The language 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) Acyclic is NP-hard.

(b) $\textsc{HalfInd} \setminus \textsc{Acyclic} \in \mathrm{P}$
(Recall that $X \setminus Y$ is the subset of elements of X that are not in Y.)

(c) HalfInd is decidable.

(d) A polynomial-time reduction from HalfInd to Acyclic would imply P=NP.

(e) A polynomial-time reduction from Acyclic to 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.

---

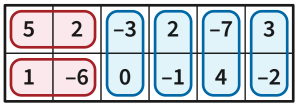
3. Suppose you are asked to tile a $2 \times n$ grid of squares with dominos (1×2 rectangles). Each domino must cover exactly two grid squares, either horizontally or vertically, and each grid square must be covered by exactly one domino.

   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.

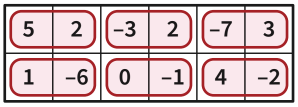
   Describe and analyze an efficient algorithm to compute the largest possible value of a domino tiling of a given $2 \times n$ grid. Your input is an array $Points[1..2, 1..n]$ of point values.

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

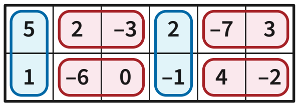
5	2	−3	2	−7	3
1	−6	0	−1	4	−2



value = —6



value = 2



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 Φ 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 Φ 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 Δ. (The value of Δ 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 $Elevation[1..n, 1..n]$ of elevation values, the starting point s, the target point t, and the parameter Δ.

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:

$$00011111110000 = 000 \bullet 1111111 \bullet 0000$$

   (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 ε, but does not contain either 001100 or 1111110.

   - 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 0110001111 and 1100000 and 000 and the empty string ε, but does not contain either 000111 or 100011.

   Prove that L_b is not a regular language.

CS/ECE 374 A ✧ Fall 2021

Conflict Final Exam

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 $c \in Z^d$, 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 $T(n) = \mathbf{2}T(n/\mathbf{4}) + O(n^2)$ is $T(n) = O(n^2)$.

(b) The solution to the recurrence $T(n) = \mathbf{4}T(n/\mathbf{2}) + O(n^2)$ 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:

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

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:

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

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

(a) $\text{Acyclic} \in \text{NP}$

(b) $\text{Acyclic} \cap \text{DirHamPath} \in \text{P}$

(c) DirHamPath is decidable.

(d) A polynomial-time reduction from DirHamPath to Acyclic would imply P=NP.

(e) A polynomial-time reduction from Acyclic to 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 $\text{P} \neq \text{NP}$?

(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

A:	7	5	4	1	2	3	3	2	3	1	4	2
B:	5	1	2	4	7	3	5	2	4	6	3	1

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

A:	2	3	5	1	3
B:	3	4	1	2	1

   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:

$$32492 \to \overset{57^2}{3249} \to \overset{18^2}{324} \to 24 \to \overset{2^2}{4} \to \varepsilon.$$

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 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 00011111110000 is the concatenation of three runs:

$$00011111110000 = 000 \bullet 1111111 \bullet 0000$$

(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 ε, but neither language contains 000111 or 100011 or 0000.