“CS 374” Fall 2014 — Homework 0

Due Tuesday, September 2, 2014 at noon

• • • Some important course policies • • •

- **Each student must submit individual solutions for this homework.** You may use any source at your disposal—paper, electronic, or human—but you ***must*** cite ***every*** source that you use. See the academic integrity policies on the course web site for more details. For all future homeworks, groups of up to three students will be allowed to submit joint solutions.

- **Submit your solutions on standard printer/copier paper.** At the top of each page, please clearly print your name and NetID, and indicate your registered discussion section. Use both sides of the paper. If you plan to write your solutions by hand, please print the last three pages of this homework as templates. If you plan to typeset your homework, you can find a LaTeX template on the course web site; well-typeset homework will get a small amount of extra credit.

- **Submit your solutions in the drop boxes outside 1404 Siebel.** There is a separate drop box for each numbered problem. Don't staple your entire homework together. Don't give your homework to Jeff in class; he is fond of losing important pieces of paper.

- **Avoid the Three Deadly Sins!** There are a few dangerous writing (and thinking) habits that will trigger an automatic zero on any homework or exam problem. Yes, we are completely serious.

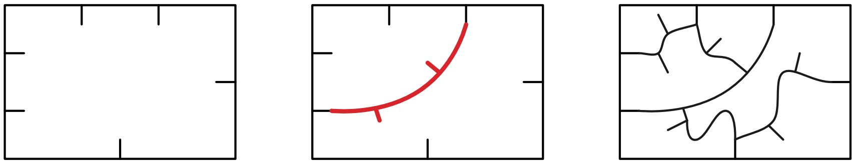
  - Give complete solutions, not just examples.
  - Declare all your variables.
  - Never use weak induction.

- Answering any homework or exam problem (or subproblem) in this course with “I don't know” *and nothing else* is worth 25% partial credit. We will accept synonyms like “No idea” or “WTF”, but you must write *something*.

See the course web site for more information.

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

1. The Terminal Game is a two-person game played with pen and paper. The game begins by drawing a rectangle with n "terminals" protruding into the rectangles, for some positive integer n, as shown in the figure below. On a player's turn, she selects two terminals, draws a simple curve from one to the other without crossing any other curve (or itself), and finally draws a new terminal on each side of the curve. A player loses if it is her turn and no moves are possible, that is, if no two terminals may be connected without crossing at least one other curve.



The initial setup. The first turn. No more moves.

Analyze this game, answering the following questions (and any more that you determine the answers to): When is it better to play first, and when it is better to play second? Is there always a winning strategy? What is the smallest number of moves in which you can defeat your opponent? Prove your answers are correct.

2. Herr Professor Doktor Georg von den Dschungel has a 23-node binary tree, in which each node is labeled with a unique letter of the German alphabet, which is just like the English alphabet with four extra letters: **Ä**, **Ö**, **Ü**, and **ß**. (Don't confuse these with **A**, **O**, **U**, and **B**!) Preorder and postorder traversals of the tree visit the nodes in the following order:

   - Preorder: **B K Ü E H L Z I Ö R C ß T S O A Ä D F M N U G**
   - Postorder: **H I Ö Z R L E C Ü S O T A ß K D M U G N F Ä B**

   (a) List the nodes in Professor von den Dschungel's tree in the order visited by an inorder traversal.

   (b) Draw Professor von den Dschungel's tree.

3. Recursively define a set L of strings over the alphabet $\{\mathtt{0},\mathtt{1}\}$ as follows:

   - The empty string ε is in L.
   - For any two strings x and y in L, the string $\mathtt{0}x\mathtt{1}y\mathtt{0}$ is also in L.
   - These are the only strings in L.

   (a) Prove that the string **000010101010010100** is in L.

   (b) Prove by induction that every string in L has exactly twice as many **0**s as **1**s.

   (c) Give an example of a string with exactly twice as many **0**s as **1**s that is *not* in L.

   Let $\#(a,w)$ denote the number of times symbol a appears in string w; for example,

   $$\#(\mathtt{0},\mathtt{000010101010010100}) = 12 \qquad \text{and} \qquad \#(\mathtt{1},\mathtt{000010101010010100}) = 6.$$

   You may assume without proof that $\#(a,xy) = \#(a,x) + \#(a,y)$ for any symbol a and any strings x and y.

4. **This is an extra credit problem. Submit your solutions in the drop box for problem 2 (but don't staple your solutions for 2 and 4 together).**

   A *perfect riffle shuffle*, also known as a *Faro shuffle*, is performed by cutting a deck of cards exactly in half and then *perfectly* interleaving the two halves. There are two different types of perfect shuffles, depending on whether the top card of the resulting deck comes from the top half or the bottom half of the original deck. An *out-shuffle* leaves the top card of the deck unchanged. After an in-shuffle, the original top card becomes the second card from the top. For example:

$$\mathit{OutShuffle}(\text{A}\spadesuit\ 2\spadesuit\ 3\spadesuit\ 4\spadesuit\ 5\heartsuit\ 6\heartsuit\ 7\heartsuit\ 8\heartsuit) = \text{A}\spadesuit\ 5\heartsuit\ 2\spadesuit\ 6\heartsuit\ 3\spadesuit\ 7\heartsuit\ 4\spadesuit\ 8\heartsuit$$

$$\mathit{InShuffle}(\text{A}\spadesuit\ 2\spadesuit\ 3\spadesuit\ 4\spadesuit\ 5\heartsuit\ 6\heartsuit\ 7\heartsuit\ 8\heartsuit) = 5\heartsuit\ \text{A}\spadesuit\ 6\heartsuit\ 2\spadesuit\ 7\heartsuit\ 3\spadesuit\ 8\heartsuit\ 4\spadesuit$$

   (If you are unfamiliar with playing cards, please refer to the Wikipedia article https://en.wikipedia.org/wiki/Standard_52-card_deck.)

   Suppose we start with a deck of 2^n distinct cards, for some non-negative integer n. What is the effect of performing exactly n perfect in-shuffles on this deck? Prove your answer is correct!

“CS 374” Fall 2014 — Homework 1

Due Tuesday, September 9, 2014 at noon

Groups of up to three students may submit common solutions for each problem in this homework and in all future homeworks. You are responsible for forming you own groups; you are welcome to advertise for group members on Piazza. You need not use the same group for every homework, or even for every problem in a single homework. Please clearly print the names and NetIDs of each of your group members at the top of each submitted solution, along with ***one*** discussion section where we should return your graded work. If you submit hand-written solutions, please use the last three pages of this homework as templates.

---

1. Give regular expressions for each of the following languages over the alphabet {0, 1}. You do not need to prove your answers are correct.

   (a) All strings with an odd number of 1s.

   (b) All strings with at most three 0s.

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

   (d) All strings in which every occurrence of the substring 00 occurs before every occurrence of the substring 11.

2. Recall that the ***reversal*** $\boldsymbol{w^R}$ of a string w is defined recursively as follows:

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

   The reversal L^R of a language L is defined as the set of reversals of all strings in L:

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

   (a) Prove that $(L^*)^R = (L^R)^*$ for every language L.

   (b) Prove that the reversal of any regular language is also a regular language. (You may assume part (a) even if you haven't proved it yet.)

   You may assume the following facts without proof:

   - $L^* \bullet L^* = L^*$ for every language L.
   - $(w^R)^R = w$ for every string w.
   - $(x \bullet y)^R = y^R \bullet x^R$ for all strings x and y.

   [Hint: Yes, all three proofs use induction, but induction on what? And yes, all ***three*** *proofs.]*

3. Describe context-free grammars for each of the following languages over the alphabet {0, 1}. Explain *briefly* why your grammars are correct; in particular, describe *in English* the language generated by each non-terminal in your grammars. (We are not looking for full formal proofs of correctness, but convincing evidence that *you* understand why your answers are correct.)

   (a) The set of all strings with more than twice as many 0s as 1s.

   (b) The set of all strings that are *not* palindromes.

   *(c) **[Extra credit]** The set of all strings that are *not* equal to ww for any string w.
   [Hint: $a + b = b + a$.]

“CS 374” Fall 2014 — Homework 2

Due Tuesday, September 16, 2014 at noon

Groups of up to three students may submit common solutions for each problem in this homework and in all future homeworks. You are responsible for forming you own groups; you are welcome to advertise for group members on Piazza. You need not use the same group for every homework, or even for every problem in a single homework. Please clearly print the names and NetIDs of each of your group members at the top of each submitted solution, along with ***one*** discussion section where we should return your graded work. If you submit hand-written solutions, please use the last three pages of this homework as templates.

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

   - Line comments: Strings of the form `// ... ↵`.
   - 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, the following strings are all valid C comments:

   - `/∗∗∗/`
   - `//⋄//⋄↵`
   - `/∗///⋄∗⋄↵∗∗/`
   - `/∗⋄//⋄↵⋄∗/`

   On the other hand, the following strings are *not* valid C comments:

   - `/∗/`
   - `//⋄//⋄↵⋄↵`
   - `/∗⋄/∗⋄∗/⋄∗/`

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

   (b) Describe a DFA that accepts the set of all strings composed entirely of blanks(`⋄`), newlines(`↵`), and C comments.

   You must explain *in English* how your DFAs work. Drawings or formal descriptions without English explanations will receive no credit, even if they are correct.

2. Construct a DFA for the following language over alphabet $\{0,1\}$:

$$L = \left\{ w \in \{0,1\}^* \;\middle|\; \begin{array}{l} \text{the number represented by binary string } w \text{ is divisible} \\ \text{by 19, but the length of } w \text{ is not a multiple of 23} \end{array} \right\}.$$

   You must explain *in English* how your DFA works. A formal description without an English explanation will receive no credit, even if it is correct. Don't even try to <u>draw</u> the DFA.

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

   (a) $\{w \in \{0\}^* \mid \text{length of } w \text{ is a perfect square; that is, } |w| = k^2 \text{ for some integer } k\}$.

   (b) $\{w \in \{0,1\}^* \mid \text{the number represented by } w \text{ as a binary string is a perfect square}\}$.

*4. **[Extra credit]** Suppose L is a regular language which guarantees to contain at least one palindrome. Prove that if an n-state DFA M accepts L, then L contains a palindrome of length polynomial in n. What is the polynomial bound you get?

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

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

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

The C++ commenting 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 comments.

“CS 374” Fall 2014 — Homework 3

Due Tuesday, September 23, 2014 at noon

- As usual, groups of up to three students may submit common solutions for this assignment. Each group should submit exactly ***one*** solution for each problem. Please clearly print the names and NetIDs of each of your group members at the top of each submitted solution, along with ***one*** discussion section where we should return your graded work. If you submit hand-written solutions, please use the last three pages of this handout as templates.

- If a question asks you to construct an NFA, you are welcome to use ε-transitions.

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

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

   (a) $(\mathtt{01} + \mathtt{10})^*(\mathtt{0} + \mathtt{1} + \varepsilon)$

   (b) $\mathtt{1}^* + (\mathtt{10})^* + (\mathtt{100})^*$

2. Prove that for any regular language L, the following languages are also regular:

   (a) $\textsc{Substrings}(L) := \{x \mid wxy \in L \text{ for some } w, y \in \Sigma^*\}$

   (b) $\textsc{Half}(L) := \{w \mid ww \in L\}$

   [Hint: Describe how to transform a DFA for L into NFAs for $\textsc{Substrings}(L)$ *and* $\textsc{Half}(L)$*. What do your NFAs have to guess? Don't forget to explain* ***in English*** *how your NFAs work.]*

3. Which of the following languages over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ are regular and which are not? Prove your answers are correct. Recall that Σ^+ denotes the set of all *nonempty* strings over Σ.

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

   (b) $\left\{wxx \;\middle|\; w, x \in \Sigma^+\right\}$

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

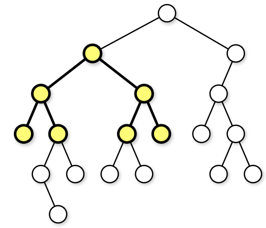
   (d) $\left\{wxxy \;\middle|\; w, x, y \in \Sigma^+\right\}$

“CS 374” Fall 2014 — Homework 4

Due Tuesday, October 7, 2014 at noon

---

1. 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.

2. Consider the following cruel and unusual sorting algorithm.

```
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 $A[1] \leftrightarrow A[2]$
  else
        for $i \leftarrow 1$ to $n/4$                 ⟨⟨swap 2nd and 3rd quarters⟩⟩
              swap $A[i + n/4] \leftrightarrow A[i + n/2]$
        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⟩⟩
```

Notice that the comparisons performed by the 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: Consider an array that contains $n/4$ 1s, $n/4$ 2s, $n/4$ 3s, and $n/4$ 4s. Why is this special case enough?]*

(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.

3. In the early 20th century, a German mathematician developed a variant of the Towers of Hanoi game, which quickly became known in the American literature as “Liberty Towers”.[1] In this variant, there is a row of $k \geq 3$ pegs, numbered in order from 1 to k. In a single turn, for any index i, you can move the smallest disk on peg i to either peg $i-1$ or peg $i+1$, subject to the usual restriction that you cannot place a bigger disk on a smaller disk. Your mission is to move a stack of n disks from peg 1 to peg k.

   (a) Describe and analyze a recursive algorithm for the case $k = 3$. ***Exactly*** how many moves does your algorithm perform?

   (b) Describe and analyze a recursive algorithm for the case $k = n + 1$ that requires at most $O(n^3)$ moves. To simplify the algorithm, assume that n is a power of 2. *[Hint: Use part (a).]*

   (c) **[Extra credit]** Describe and analyze a recursive algorithm for the case $k = n+1$ that requires at most $O(n^2)$ moves. Do *not* assume that n is a power of 2. *[Hint: Don’t use part (a).]*

   (d) **[Extra credit]** Describe and analyze a recursive algorithm for the case $k = \sqrt{n} + 1$ that requires at most a polynomial number of moves. To simplify the algorithm, assume that n is a power of 4. What polynomial bound do you get? *[Hint: Use part (a)!]*

   ★(e) **[Extra extra credit]** Describe and analyze a recursive algorithm for arbitrary n and k. How small must k be (as a function of n) so that the number of moves is bounded by a polynomial in n? (This is actually an open research problem, a phrase which here means “Nobody knows the best answer.”)

[1] No, not really. During World War I, many German-derived or Germany-related names were changed to more patriotic variants. For example, sauerkraut became “liberty cabbage”, hamburgers became “liberty sandwiches”, frankfurters became “Liberty sausages” or “hot dogs”, German measles became “liberty measles”, dachsunds became “liberty pups”, German shepherds became “Alsatians”, and pinochle (the card game) became “Liberty”. For more recent anti-French examples, see “freedom fries”, “freedom toast”, and “liberty lip lock”. Americans are weird.

“CS 374” Fall 2014 — Homework 5

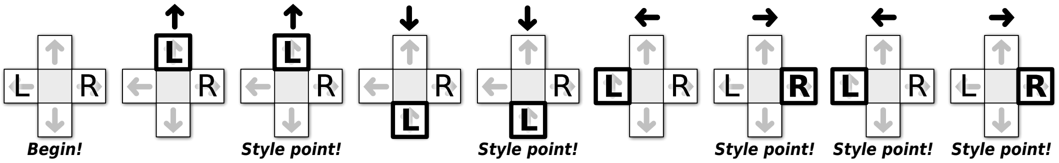
Due Tuesday, October 14, 2014 at noon

---

1. ***Dance Dance Revolution*** is a dance video game, first introduced in Japan by Konami in 1998. Players stand on a platform marked with four arrows, pointing forward, back, left, and right, arranged in a cross pattern. During play, the game plays a song and scrolls a sequence of n arrows (←, ↑, ↓, or →) from the bottom to the top of the screen. At the precise moment each arrow reaches the top of the screen, the player must step on the corresponding arrow on the dance platform. (The arrows are timed so that you’ll step with the beat of the song.)

   You are playing a variant of this game called “Vogue Vogue Revolution”, where the goal is to play perfectly but move as little as possible. When an arrow reaches the top of the screen, if one of your feet is already on the correct arrow, you are awarded one style point for maintaining your current pose. If neither foot is on the right arrow, you must move one (and *only* one) of your feet from its current location to the correct arrow on the platform. If you ever step on the wrong arrow, or fail to step on the correct arrow, or move more than one foot at a time, or move either foot when you are already standing on the correct arrow, or insult Beyoncé, all your style points are immediately taken away and you lose.

   How should you move your feet to maximize your total number of style points? For purposes of this problem, assume you always start with you left foot on ← and you right foot on →, and that you’ve memorized the entire sequence of arrows. For example, if the sequence is ↑↑↓↓←→←→, you can earn 5 style points by moving you feet as shown below:



   Describe and analyze an efficient algorithm to find the maximum number of style points you can earn during a given VVR routine. Your input is an array $\mathit{Arrow}[1\,..\,n]$ containing the sequence of arrows.

2. Recall that a *palindrome* is any string that is exactly the same as its reversal, like I, or DEED, or RACECAR, or AMANAPLANACATACANALPANAMA.

   Any string can be decomposed into a sequence of palindrome substrings. For example, the string BUBBASEESABANANA (“Bubba sees a banana.”) can be broken into palindromes in the following ways (among many others):

   BUB • BASEESAB • ANANA

   B • U • BB • A • SEES • ABA • NAN • A

   B • U • BB • A • SEES • A • B • ANANA

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

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

3. Suppose you are given a DFA $M = (\{0, 1\}, Q, s, A, \delta)$ and a binary string $w \in \{0, 1\}^*$.

   (a) Describe and analyze an algorithm that computes the longest subsequence of w that is accepted by M, or correctly reports that M does not accept any subsequence of w.

   *(b) **[Extra credit]** Describe and analyze an algorithm that computes the *shortest supersequence* of w that is accepted by M, or correctly reports that M does not accept any supersequence of w. (Recall that a string x is a supersequence of w if and only if w is a subsequence of x.)

   Analyze both of your algorithms in terms of the parameters $n = |w|$ and $k = |Q|$.

Rubric (for all dynamic programming problems): As usual, a score of x on the following 10-point scale corresponds to a score of $\lceil x/3 \rceil$ on the 4-point homework scale.

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 point for a clear **English** description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) **Automatic zero if the English description is missing.**
  - + 1 point for stating how to call your function to get the final answer.
  - + 1 point for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 points for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 4 points for details of the dynamic programming algorithm
  - + 1 point for describing the memoization data structure
  - + 2 points for describing a correct evaluation order; a clear picture is sufficient. If you use nested loops, be sure to specify the nesting order.
  - + 1 point for time analysis
- It is *not* necessary to state a space bound.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but this is not required for full credit***. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, data structure, or evaluation order. (But you still need to describe the underlying recursive function in English.)
- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.

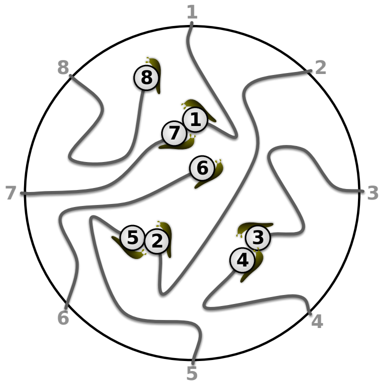
  We rarely include these target time bounds in the actual questions, because when we do include them, significantly more students turn in algorithms that meet the target time bound but don't work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).

“CS 374” Fall 2014 — Homework 6

Due Tuesday, October 21, 2014 at noon

---

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. Consider a weighted version of the class scheduling problem, where different classes offer different number of credit hours, which are of course totally unrelated to the duration of the class lectures. Given arrays $S[1..n]$ of start times, an array $F[1..n]$ of finishing times, and an array $H[1..n]$ of credit hours as input, your goal is to choose a set of non-overlapping classes with the largest possible number of credit hours.

   (a) Prove that the greedy algorithm described in class — Choose the class that ends first and recurse — does *not* always return the best schedule.

   (b) Describe an efficient algorithm to compute the best schedule.

   In addition to submitting a solution on paper as usual, please *individually* submit an electronic solution for this problem on CrowdGrader. Please see the course web page for detailed instructions.

3. Suppose you have just purchased a new type of hybrid car that uses fuel extremely efficiently, but can only travel 100 miles on a single battery. The car's fuel is stored in a single-use battery, which must be replaced after at most 100 miles. The actual fuel is virtually free, but the batteries are expensive and can only be installed by licensed battery-replacement technicians. Thus, even if you decide to replace your battery early, you must still pay full price for the new battery to be installed. Moreover, because these batteries are in high demand, no one can afford to own more than one battery at a time.

   Suppose you are trying to get from San Francisco to New York City on the new Inter-Continental Super-Highway, which runs in a direct line between these two cities. There are several fueling stations along the way; each station charges a different price for installing a new battery. Before you start your trip, you carefully print the Wikipedia page listing the locations and prices of every fueling station on the ICSH. Given this information, how do you decide the best places to stop for fuel?

   More formally, suppose you are given two arrays $D[1..n]$ and $C[1..n]$, where $D[i]$ is the distance from the start of the highway to the ith station, and $C[i]$ is the cost to replace your battery at the ith station. Assume that your trip starts and ends at fueling stations (so $D[1] = 0$ and $D[n]$ is the total length of your trip), and that your car starts with an empty battery (so you must install a new battery at station 1).

   (a) Describe and analyze a greedy algorithm to find the minimum number of refueling stops needed to complete your trip. Don't forget to prove that your algorithm is correct.

   (b) But what you really want to minimize is the total *cost* of travel. Show that your greedy algorithm in part (a) does *not* produce an optimal solution when extended to this setting.

   (c) Describe an efficient algorithm to compute the locations of the fuel stations you should stop at to minimize the total cost of travel.

“CS 374” Fall 2014 — Homework 7

Due Tuesday, October 28, 2014 at noon

1. You are standing next to a water pond, and you have three empty jars. Each jar holds a positive integer number of gallons; the capacities of the three jars may or may not be different. You want one of the jars (which one doesn’t matter) to contain exactly k gallons of water, for some integer k. You are only allowed to perform the following operations:

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

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

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

   Describe an efficient algorithm that finds the minimum number of operations required to obtain a jar containing exactly k gallons of water, or reports correctly that obtaining exactly k gallons of water is impossible, given the capacities of the three jars and a positive integer k as input. For example, given the four numbers $6, 10, 15$ and 13 as input, your algorithm should return the number 6 (for the sequence of operations listed above).

2. Consider a directed graph G, where each edge is colored either red, white, or blue. A walk[1] in G is called a *French flag walk* if its sequence of edge colors is red, white, blue, red, white, blue, and so on. More formally, a walk $v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k$ is a French flag path if, for every integer i, the edge $v_i \rightarrow v_{i+1}$ is red if $i \bmod 3 = 0$, white if $i \bmod 3 = 1$, and blue if $i \bmod 3 = 2$.

   Describe an efficient algorithm to find all vertices in a given edge-colored directed graph G that can be reached from a given vertex v through a French flag walk.

3. Suppose we are given a directed acyclic graph G where every edge e has a positive integer weight $w(e)$, along with two specific vertices s and t and a positive integer W.

   (a) Describe an efficient algorithm to find the *longest* path (meaning the largest number of edges) from s to t in G with total weight at most W. *[Hint: Use dynamic programming.]*

   (b) **[Extra credit]** Solve part (a) with a running time that does not depend on W.

[1] Recall that a ***walk*** in G is a sequence of vertices $v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k$, such that $v_{i-1} \rightarrow v_i$ is an edge in G for every index i. A ***path*** is a walk in which no vertex appears more than once.

“CS 374” Fall 2014 — Homework 8

Due Tuesday, November 4, 2014 at noon

---

1. After a grueling algorithms midterm, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Champaign-Urbana. Champaign-Urbana is currently suffering from a plague of zombies, so even though the bus stops have fences that *supposedly* keep the zombies out, you'd still like to spend as little time waiting at bus stops as possible. Unfortunately, there isn't a single bus that visits both your exam building and your home; you must transfer between buses at least once.

   Describe and analyze an algorithm to determine a sequence of bus rides from Siebel to your home, that minimizes the total time you spend waiting at bus stops. You can assume that there are b different bus lines, and each bus stops n times per day. Assume that the buses run exactly on schedule, that you have an accurate watch, and that walking between bus stops is too dangerous to even contemplate.

2. It is well known that the global economic collapse of 2017 was caused by computer scientists indiscriminately abusing weaknesses in the currency exchange market. *Arbitrage* was a money-making scheme that takes advantage of inconsistencies in currency exchange rates. Suppose a currency trader with \$1,000,000 discovered that 1 US dollar could be traded for 120 Japanese yen, 1 yen could be traded for 0.01 euros, and 1 euro could be traded for 1.2 US dollars. Then by converting his money from dollars to yen, then from yen to euros, and finally from euros back to dollars, the trader could instantly turn his \$1,000,000 into \$1,440,000! The cycle of currencies \$ → ¥ → € → \$ was called an ***arbitrage cycle***. Finding and exploiting arbitrage cycles before the prices were corrected required extremely fast algorithms. Of course, now that the entire world uses plastic bags as currency, such abuse is impossible.

   Suppose n different currencies are traded in the global currency market. You are given a two-dimensional array $Exch[1..n, 1..n]$ of exchange rates between every pair of currencies; for all indices i and j, one unit of currency i buys $Exch[i, j]$ units of currency j. (Do *not* assume that $Exch[i, j] \cdot Exch[j, i] = 1$.)

   (a) Describe an algorithm that computes an array $Most[1..n]$, where $Most[i]$ is the largest 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 array of currency exchange rates creates an arbitrage cycle.

3. Describe and analyze and algorithm to find the *second smallest spanning tree* of a given undirected graph G with weighted edges, that is, the spanning tree of G with smallest total weight except for the minimum spanning tree. Because the minimum spanning tree is haunted, or something.

“CS 374” Fall 2014 — Homework 9

Due Tuesday, November 18, 2014 at noon

The following questions ask you to describe various Turing machines. In each problem, give *both* a formal description of your Turing machine in terms of specific states, tape symbols, and transition functions *and* explain in English how your Turing machine works. In particular:

- Clearly specify what variant of Turing machine you are using: Number of tapes, number of heads, allowed head motions, halting conditions, and so on.
- Include the type signature of your machine's transition function. The standard model uses a transition function whose signature is $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$.
- If necessary, break your Turing machine into smaller functional pieces, and describe those pieces separately (both formally and in English).
- Use state names that convey their meaning/purpose.

---

1. Describe a Turing machine that computes the function $\lceil \log_2 n \rceil$. Given the string 1^n as input, for any positive integer n, your machine should return the string $1^{\lceil \log_2 n \rceil}$ as output. For example, given the input string 1111111111111 (thirteen 1s), your machine should output the string 1111, because $2^3 < 13 \le 2^4$.

2. A ***binary-tree*** Turing machine uses an infinite binary tree as its tape; that is, *every* cell in the tape has a left child and a right child. At each step, the head moves from its current cell to its Parent, its Left child, or to its Right child. Thus, the transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{P, L, R\}$. The input string is initially given along the left spine of the tape.

   Prove that any binary-tree Turing machine can be simulated by a standard Turing machine. That is, given any binary-tree Turing machine $M = (\Gamma, \square, \Sigma, Q, \mathsf{start}, \mathsf{accept}, \mathsf{reject}, \delta)$, describe a standard Turing machine $M' = (\Gamma', \square', \Sigma, Q', \mathsf{start}', \mathsf{accept}', \mathsf{reject}', \delta')$ that accepts and rejects exactly the same input strings as M. Be sure to describe how a single transition of M is simulated by M'.

   In addition to submitting paper solutions, please also electronically submit your solution to this problem on CrowdGrader.

3. **[Extra credit]**

   A ***tag***-Turing machine has two heads: one can only read, the other can only write. Initially, the read head is located at the left end of the tape, and the write head is located at the first blank after the input string. At each transition, the read head can either move one cell to the right or stay put, but the write head *must* write a symbol to its current cell and move one cell to the right. Neither head can ever move to the left.

   Prove that any standard Turing machine can be simulated by a tag-Turing machine. That is, given any standard Turing machine M, formally describe a tag-Turing machine M' that accepts and rejects exactly the same strings as M. Be sure to describe how a single transition of M is simulated by M'.

"CS 374" Fall 2014 — Homework 10

Due Tuesday, December 2, 2014 at noon

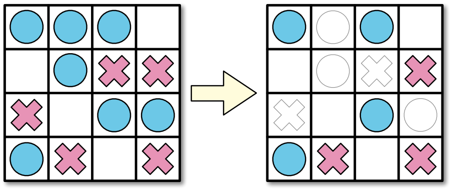
1. Consider the following problem, called BoxDepth: Given a set of n axis-aligned rectangles in the plane, how big is the largest subset of these rectangles that contain a common point?

   (a) Describe a polynomial-time reduction from BoxDepth to MaxClique.

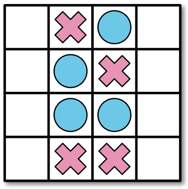
   (b) Describe and analyze a polynomial-time algorithm for BoxDepth. *[Hint: Don't try to optimize the running time; $O(n^3)$ is good enough.]*

   (c) Why don't these two results imply that P=NP?

2. 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, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.



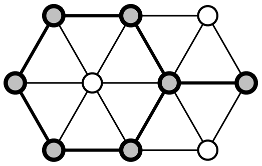
A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

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

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



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

   In addition to submitting paper solutions, please also electronically submit your solution to this problem on CrowdGrader.

4. **[Extra credit]** Describe a direct polynomial-time reduction from 4Color to 3Color. (This is significantly harder than the opposite direction, which you'll see in lab on Wednesday. Don't go through the Cook-Levin Theorem.)

“CS 374” Fall 2014 ✧ Homework 11

Due Tuesday, December 9, 2014 at noon

---

1. Recall that w^R denotes the reversal of string w; for example, $\texttt{TURING}^R = \texttt{GNIRUT}$. Prove that the following language is undecidable.

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

2. Let M be a Turing machine, let w be an arbitrary input string, and let s be an integer. We say that ***M* accepts *w* in space *s*** if, given w as input, M accesses only the first s cells on the tape and eventually accepts.

   (a) Prove that the following language is decidable:

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

   (b) Prove that the following language is undecidable:

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

3. ***[Extra credit]*** For each of the following languages, either prove that the language is decidable, or prove that the language is undecidable.

   (a) $L_0 = \left\{ \langle M \rangle \;\middle|\; \text{given any input string, } M \text{ eventually leaves its } \texttt{start} \text{ state} \right\}$

   (b) $L_1 = \left\{ \langle M \rangle \;\middle|\; M \text{ decides } L_0 \right\}$

   (c) $L_2 = \left\{ \langle M \rangle \;\middle|\; M \text{ decides } L_1 \right\}$

   (d) $L_3 = \left\{ \langle M \rangle \;\middle|\; M \text{ decides } L_2 \right\}$

   (e) $L_4 = \left\{ \langle M \rangle \;\middle|\; M \text{ decides } L_3 \right\}$

1. Prove that every non-negative integer can be represented as the sum of distinct powers of 2. ("Write it in binary" is not a proof; it's just a restatement of what you have to prove.)

2. Suppose you and your 8-year-old cousin Elmo decide to play a game with a rectangular bar of chocolate, which has been scored into an $n \times m$ grid of squares. You and Elmo alternate turns. On each turn, you or Elmo choose one of the available pieces of chocolate and break it along one of the grid lines into two smaller rectangles. Thus, at all times, each piece of chocolate is an $a \times b$ rectangle for some positive integers a and b; in particular, a 1×1 piece cannot be broken into smaller pieces. The game ends when all the pieces are individual squares. The winner is the player who breaks the last piece.

   Describe a strategy for winning this game. When should you take the first move, and when should you offer it to Elmo? On each turn, how do you decide which piece to break and where? Prove your answers are correct. *[Hint: Let's play a 3×3 game. You go first. Oh, and I'm kinda busy right now, so could you just play for me whenever it's my turn? Thanks.]*

3. **[To think about later]** Now consider a variant of the previous chocolate-bar game, where on each turn you can *either* break a piece into two smaller pieces *or eat a* 1×1 *piece*. This game ends when all the chocolate is gone. The winner is the player who eats the last bite of chocolate (*not* the player who eats the *most* chocolate). Describe a strategy for winning this game, and prove that your strategy works.

These lab problems ask you to prove some simple claims about recursively-defined string functions and concatenation. In each case, we want a self-contained proof by induction that relies on the formal recursive definitions, *not* on intuition. In particular, your proofs must refer to the formal recursive definition of string concatenation:

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

You may also use any of the following facts, which we proved in class:

Lemma 1: Concatenating nothing does nothing: For every string w, we have $w \bullet \varepsilon = w$.

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

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

---

1. Let $\#(a, w)$ denote the number of times symbol a appears in string w; for example,

$$\#(\mathtt{0}, \mathtt{000010101010010100}) = 12 \qquad \text{and} \qquad \#(\mathtt{1}, \mathtt{000010101010010100}) = 6.$$

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

   (b) Prove by induction that $\#(a, w \bullet z) = \#(a, w) + \#(a, z)$ for any symbol a and any strings w and z.

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

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

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

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

Give regular expressions that describe each of the following languages over the alphabet {**0**,**1**}. We won't get to all of these in section.

1. All strings containing at least three **0**s.

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

3. All strings containing the substring **000**.

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

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

6. All strings such that every substring **000** appears after every **1**.

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

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

Jeff showed the context-free grammars in class on Tuesday; 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.

$S \to A \mid B$	all strings $\texttt{0}^m\texttt{1}^n$ where $m \neq n$
$A \to \texttt{0}A \mid \texttt{0}C$	all strings $\texttt{0}^m\texttt{1}^n$ where $m > n$
$B \to B\texttt{1} \mid C\texttt{1}$	all strings $\texttt{0}^m\texttt{1}^n$ where $m < m$
$C \to \varepsilon \mid \texttt{0}C\texttt{1}$	all strings $\texttt{0}^n\texttt{1}^n$ for some integer n

---

Give context-free grammars for each of the following languages. For each grammar, describe *in English* the language for each non-terminal, and in the examples above. As usual, we won't get to all of these in section.

1. Binary palindromes: Strings over $\{\texttt{0},\texttt{1}\}$ that are equal to their reversals. For example: 00111100 and 0100010, but not 01100.

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

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

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

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.

6. Strings over $\{\texttt{0},\texttt{1}\}$ where the number of 0s is equal to the number of 1s.

7. Strings over $\{\texttt{0},\texttt{1}\}$ where the number of 0s is *not* equal to the number of 1s.

Construct DFA that accept each of the following languages over the alphabet $\{0,1\}$. We won't get to all of these in section.

---

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

   (b) $\emptyset$

   (c) $\{\epsilon\}$

2. Every string except 000.

3. All strings containing the substring 000.

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

5. All strings in which the reverse of the string is the binary representation of a integer divisible by 3.

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

Prove that each of the following languages is not regular.

1. Binary palindromes: Strings over $\{\mathtt{0},\mathtt{1}\}$ that are equal to their reversals. For example: **00111100** and **0100010**, but not **01100**. *[Hint: We did this in class.]*

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 **0**s is exactly twice the number of **1**s.

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.

6. $\left\{\mathtt{0}^{2^n} \,\middle|\, n \geq 0\right\}$ — Strings of **0**s whose length is a power of 2.

7. 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.

For each of the following languages over the alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$, either prove the language is regular (by giving an equivalent regular expression, DFA, or NFA) or prove that the language is not regular (using a fooling set argument). Exactly half of these languages are regular.

1. $\{\texttt{0}^n\texttt{1}\texttt{0}^n \mid n \geq 0\}$
2. $\{\texttt{0}^n\texttt{1}\texttt{0}^n w \mid n \geq 0 \text{ and } w \in \Sigma^*\}$
3. $\{w\texttt{0}^n\texttt{1}\texttt{0}^n x \mid w \in \Sigma^* \text{ and } n \geq 0 \text{ and } x \in \Sigma^*\}$
4. Strings in which the number of 0s and the number of 1s differ by at most 2.
5. Strings such that *in every prefix*, the number of 0s and the number of 1s differ by at most 2.
6. Strings such that *in every substring*, the number of 0s and the number of 1s differ by at most 2.

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. (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. 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. (a) 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?]*

   (b) ***To think about on your own:*** 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.

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, and ***prove*** that your algorithm is correct.

1. (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. 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. (a) 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?]*

   (b) ***To think about on your own:*** 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.

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

Describe recursive backtracking algorithms for the following 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

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

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).

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 the string `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 problems. For the first three, use the backtracking algorithms 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. Don't describe *how* to solve the problem at this stage; just describe *what* the problem actually is. Otherwise, the reader has no way to know what your recursive algorithm is *supposed* to compute.

   (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.

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? For example, the argument to RecFibo is always an integer between 0 and n.

   (b) **Analyze space and running time.** The number of possible distinct subproblems determines the space complexity of your memoized algorithm. To compute the time complexity, add up the running times of all possible subproblems, *ignoring the recursive calls*. For example, if we already know F_{i-1} and F_{i-2}, we can compute F_i in $O(1)$ time, so computing the first n Fibonacci numbers takes $O(n)$ time.

   (c) **Choose a data structure to memoize intermediate results.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. For some problems, however, a more complicated data structure is required.

   (d) **Identify dependencies between subproblems.** 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, this means you should consider the base cases first, then the subproblems that depends only on base cases, and so on. More formally, the dependencies you identified in the previous step define a partial order over the subproblems; in this step, you need to find a linear extension of that partial order. ***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.

1. 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 $\mathit{Wait}[1\,..\,n]$.

2. 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.

   $$\texttt{BANANA}_{\texttt{ANANAS}} \qquad \texttt{BAN}_{\texttt{ANA}}\texttt{ANA}_{\texttt{NAS}} \qquad \texttt{B}_{\texttt{AN}}\texttt{AN}_{\texttt{A}}\texttt{A}_{\texttt{NA}}\texttt{NA}_{\texttt{S}}$$

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

   $$\texttt{PRO}^{\texttt{D}}\texttt{G}^{\texttt{Y}}\texttt{R}^{\texttt{NAM}}\texttt{AMMI}^{\texttt{I}}\texttt{N}^{\texttt{C}}\texttt{G} \qquad {}^{\texttt{DY}}\texttt{PRO}^{\texttt{N}}\texttt{G}^{\texttt{A}}\texttt{R}^{\texttt{M}}\texttt{AMM}^{\texttt{IC}}\texttt{ING}$$

   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.

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. Don't describe *how* to solve the problem at this stage; just describe *what* the problem actually is. Otherwise, the reader has no way to know what your recursive algorithm is *supposed* to compute.

   (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.

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? For example, the argument to RECFIBO is always an integer between 0 and n.

   (b) **Analyze space and running time.** The number of possible distinct subproblems determines the space complexity of your memoized algorithm. To compute the time complexity, add up the running times of all possible subproblems, *ignoring the recursive calls*. For example, if we already know F_{i-1} and F_{i-2}, we can compute F_i in $O(1)$ time, so computing the first n Fibonacci numbers takes $O(n)$ time.

   (c) **Choose a data structure to memoize intermediate results.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. For some problems, however, a more complicated data structure is required.

   (d) **Identify dependencies between subproblems.** 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, this means you should consider the base cases first, then the subproblems that depends only on base cases, and so on. More formally, the dependencies you identified in the previous step define a partial order over the subproblems; in this step, you need to find a linear extension of that partial order. ***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.

1. 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:

   $$\begin{gathered} 2 \times (3 + (0 \times (6 \times (1 + (4 \times 2))))) = 6 \\ (((((2 \times 3) + 0) \times 6) \times 1) + 4) \times 2 = 80 \\ ((2 \times 3) + (0 \times 6)) \times (1 + (4 \times 2)) = 108 \\ (((2 \times 3) + 0) \times 6) \times ((1 + 4) \times 2) = 360 \end{gathered}$$

   Describe and analyze an algorithm to compute, given a list of integers separated by + and × signs, the largest 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 × if i is odd. Assume any arithmetic operation in your algorithm takes $O(1)$ time.

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. Don't describe *how* to solve the problem at this stage; just describe *what* the problem actually is. Otherwise, the reader has no way to know what your recursive algorithm is *supposed* to compute.

   (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.

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? For example, the argument to RecFibo is always an integer between 0 and n.

   (b) **Analyze space and running time.** The number of possible distinct subproblems determines the space complexity of your memoized algorithm. To compute the time complexity, add up the running times of all possible subproblems, *ignoring the recursive calls*. For example, if we already know F_{i-1} and F_{i-2}, we can compute F_i in $O(1)$ time, so computing the first n Fibonacci numbers takes $O(n)$ time.

   (c) **Choose a data structure to memoize intermediate results.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. For some problems, however, a more complicated data structure is required.

   (d) **Identify dependencies between subproblems.** 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, this means you should consider the base cases first, then the subproblems that depends only on base cases, and so on. More formally, the dependencies you identified in the previous step define a partial order over the subproblems; in this step, you need to find a linear extension of that partial order. ***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.

Recall the class scheduling problem described in lecture on Tuesday. We are given two arrays $S[1..n]$ and $F[1..n]$, where $S[i] < F[i]$ for each i, representing the start and finish times of n classes. Your goal is to find the largest number of classes you can take without ever taking two classes simultaneously. We showed in class that the following greedy algorithm constructs an optimal schedule:

Choose the course that *ends first*, discard all conflicting classes, and recurse.

But this is not the only greedy strategy we could have tried. For each of the following alternative greedy algorithms, either prove that the algorithm always constructs an optimal schedule, or describe a small input example for which the algorithm does not produce an optimal schedule. Assume that all algorithms break ties arbitrarily (that is, in a manner that is completely out of your control).

[Hint: Exactly three of these greedy strategies actually work.]

1. Choose the course x that *ends last*, discard classes that conflict with x, and recurse.

2. Choose the course x that *starts first*, discard all classes that conflict with x, and recurse.

3. Choose the course x that *starts last*, discard all classes that conflict with x, and recurse.

4. Choose the course x with *shortest duration*, discard all classes that conflict with x, and recurse.

5. Choose a course x that *conflicts with the fewest other courses*, discard all classes that conflict with x, and recurse.

6. If no classes conflict, choose them all. Otherwise, discard the course with *longest duration* and recurse.

7. If no classes conflict, choose them all. Otherwise, discard a course that *conflicts with the most other courses* and recurse.

8. Let x be the class with the *earliest start time*, and let y be the class with the *second earliest start time*.

   - If x and y are disjoint, choose x and recurse on everything but x.
   - If x completely contains y, discard x and recurse.
   - Otherwise, discard y and recurse.

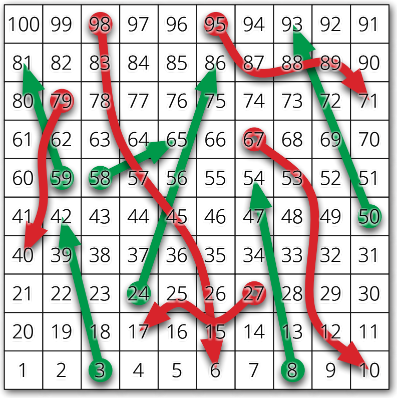
9. If any course x completely contains another course, discard x and recurse. Otherwise, choose the course y that *ends last*, discard all classes that conflict with y, and recurse.

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 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*?

---

1. ***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.

2. Let G be a connected 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 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).

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 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*?

---

1. Inspired by the previous lab, 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. 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 e has an associated toll of c_e dollars, where c_e is a positive integer. A teleport-way can be used multiple times, but the toll must be paid every time it is used.

   Judy wants to travel from galaxy u to galaxy v, but teleportation is not very pleasant and she would like to minimize the number of times she needs to teleport. However, she wants the total cost to be a multiple of five dollars, because carrying small bills is not pleasant either.

   (a) Describe and analyze an algorithm to compute the smallest number of times Judy needs to teleport to travel from galaxy u to galaxy v while the total cost is a multiple of five dollars.

   (b) Solve (a), but now assume that Judy has a coupon that allows her to waive the toll once.

Suppose we are given both an undirected graph G with weighted edges and a minimum spanning tree T of G.

1. Describe an efficient algorithm to update the minimum spanning tree when the weight of one edge $e \in T$ is decreased.

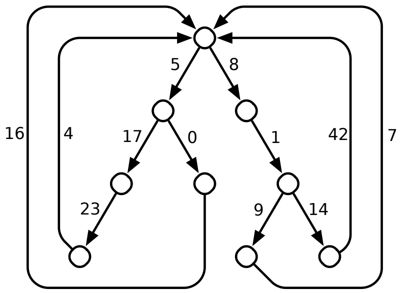
2. Describe an efficient algorithm to update the minimum spanning tree when the weight of one edge $e \notin T$ is increased.

3. Describe an efficient algorithm to update the minimum spanning tree when the weight of one edge $e \in T$ is increased.

4. Describe an efficient algorithm to update the minimum spanning tree when the weight of one edge $e \notin T$ is decreased.

In all cases, the input to your algorithm is the edge e and its new weight; your algorithms should modify T so that it is still a minimum spanning tree. Of course, we could just recompute the minimum spanning tree from scratch in $O(E \log V)$ time, but you can do better.

1. 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.



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.

2. 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. *[Hint: Build an appropriate graph from the input data and apply Dijkstra's algorithm.]*

Describe Turing machines that compute the following functions.

In particular, specify the transition functions $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$ for each machine either by writing out a table or by drawing a graph. Recall that $\delta(p, \texttt{\$}) = (q, \texttt{@}, +1)$ means that if the Turing machine is in state p and reads the symbol $ from the tape, then it will change to state q, write the symbol @ to the tape, and move one step to the right. In a *drawing* of a Turing machine, this transition is indicated by an edge from p to q with the label "$/@, +1".

Give your states short mnemonic names that suggest their purpose. Naming your states well won't just make it easier to understand; it will also make it easier to design.

1. DOUBLE: Given a string $w \in \{\texttt{0}, \texttt{1}\}^*$ as input, return the string ww as output.

2. POWER: Given a string of the form $\texttt{1}^n$ as input, return the string $\texttt{1}^{2^n}$ as output.

Describe how to simulate an arbitrary Turing machine to make it ***error-tolerant***. Specifically, given an arbitrary Turing machine M, describe a new Turing machine M' that accepts and rejects exactly the same strings as M, even though an evil pixie named Lenny will move the head of M' to an ***arbitrary*** location on the tape some finite number of *unknown* times during the execution of M'.

You do not have to describe M' in complete detail, but do give enough details that a seasoned Turing machine programmer could work out the remaining mechanical details.

As stated, this problem has no solution! If M halts on all inputs after a finite number of steps, then Lenny can make any substring of the input string completely invisible to M. For example, if the true input string is `INPUT-STRING`, Lenny can make M believe the input string is actually `IMPING`, by moving the head to the second `I` whenever it tries to move to `R`, and by moving the head to `P` when it tries to move to `U`. Because M halts after a finite number of steps, Lenny only has a finite number of opportunities to move the head.

In fact, with more care, Lenny can make M think the input string is *any* string that uses only symbols from the actual input string; if the true input string is `INPUT-STRING`, Lenny can make M believe the input string is actually `GRINNING-PUTIN-IS-GRINNING`.)

However, there are several different ways to rescue the problem. For each of the following restrictions on Lenny's behavior, and for any Turing machine M, one can design a Turing machine M' that simulates M despite Lenny's interference.

- Lenny can move the head only a *bounded* number of times. For example: Lenny can move the head at most 374 times.
- Whenever Lenny moves the head, he changes the state of the machine to a special error state `lenny`.
- Whenever Lenny moves the head, he moves it to the left end of the tape.
- Whenever Lenny moves the head, he moves it to a blank cell to the right of all non-blank cells.
- Whenever Lenny moves the head, he moves it to a cell containing a particular symbol in the input alphabet, say `0`.

Describe algorithms for the following problems. The input for each problem is string $\langle M, w\rangle$ that encodes a standard (one-tape, one-track, one-head) Turing machine M whose tape alphabet is $\{\texttt{0}, \texttt{1}, \square\}$ and a string $w \in \{\texttt{0}, \texttt{1}\}^*$.

1. Does M accept w after at most $|w|^2$ steps?

2. If we run M with input w, does M ever move its head to the right?

2½. If we run M with input w, does M ever move its head to the right twice in a row?

2¾. If we run M with input w, does M move its head to the right more than $2^{|w|}$ times?

3. If we run M with input w, does M ever change a symbol on the tape?

3½. If we run M with input w, does M ever change a $\square$ on the tape to either `0` or `1`?

4. If we run M with input w, does M ever leave its start state?

In contrast, as we will see later, the following problems are all undecidable!

1. Does M accept w?

1½. If we run M with input w, does M ever halt?

2. If we run M with input w, does M ever move its head to the right three times in a row?

3. If we run M with input w, does M ever change a $\square$ on the tape to `1`?

3½. If we run M with input w, does M ever change either `0` or `1` on the tape to $\square$?

4. If we run M with input w, does M ever reenter its start state?

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 $x_1, x_2, \ldots, x_n \in \{\text{TRUE}, \text{FALSE}\}$ 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 $x_1, x_2, \ldots, x_n \in \{\text{TRUE}, \text{FALSE}\}$ that make K output TRUE, or NONE if there are no such inputs.

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

2. Formally, ***valid 3-coloring*** of a graph $G = (V, E)$ is a function $c \colon V \to \{1, 2, 3\}$ such that $c(u) \neq c(v)$ for all $uv \in E$. Less formally, a valid 3-coloring assigns each vertex a color, which is either red, green, or blue, such that the endpoints of every edge have different colors.

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

   - INPUT: An undirected graph G.
   - OUTPUT: TRUE if G has a valid 3-coloring, and FALSE otherwise.

   Using this black box as a subroutine, describe an algorithm that solves the **3*-coloring problem*** *in polynomial time*:

   - INPUT: An undirected graph G.
   - OUTPUT: A valid 3-coloring of G, or NONE if there is no such coloring.

   *[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.
- 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 almost always requires two separate steps:
  - 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.

---

1. Recall 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. Recall that a *Hamiltonian cycle* in a graph G is a cycle that goes through every vertex of G exactly once. Now, a ***tonian cycle*** in a graph G is a cycle that goes through at least *half* of the vertices of G, and a ***Hamilhamiltonian circuit*** in a graph G is a closed walk that goes through every vertex in G exactly *twice*.

   (a) Prove that it is NP-hard to determine whether a given graph contains a tonian cycle.

   (b) Prove that it is NP-hard to determine whether a given graph contains a Hamilhamiltonian circuit.

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

- Choose a problem Y that you already know is NP-hard.
- 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 almost always requires two separate steps:
  - 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.

Recall that a *Hamiltonian cycle* in a graph G is a cycle that visits every vertex of G exactly once.

1. In class on Thursday, Jeff proved that it is NP-hard to determine whether a given *directed* graph contains a Hamiltonian cycle. Prove that it is NP-hard to determine whether a given *undirected* graph contains a Hamiltonian cycle.

2. A *double Hamiltonian circuit* in a graph G is a closed walk that goes through every vertex in G exactly *twice*. Prove that it is NP-hard to determine whether a given *undirected* graph contains a double Hamiltonian circuit.

Proving that a language L is undecidable by reduction requires several steps:

- Choose a language L' that you already know is undecidable. Typical choices for L' include:

$$\begin{aligned}
\textsc{Accept} &:= \big\{\langle M,w\rangle \bigm| M \text{ accepts } w\big\}\\
\textsc{Reject} &:= \big\{\langle M,w\rangle \bigm| M \text{ rejects } w\big\}\\
\textsc{Halt} &:= \big\{\langle M,w\rangle \bigm| M \text{ halts on } w\big\}\\
\textsc{Diverge} &:= \big\{\langle M,w\rangle \bigm| M \text{ diverges on } w\big\}\\
\textsc{NeverAccept} &:= \big\{\langle M\rangle \mid \textsc{Accept}(M) = \varnothing\big\}\\
\textsc{NeverReject} &:= \big\{\langle M\rangle \mid \textsc{Reject}(M) = \varnothing\big\}\\
\textsc{NeverHalt} &:= \big\{\langle M\rangle \mid \textsc{Halt}(M) = \varnothing\big\}\\
\textsc{NeverDiverge} &:= \big\{\langle M\rangle \mid \textsc{Diverge}(M) = \varnothing\big\}
\end{aligned}$$

- Describe an algorithm (really a Turing machine) M' that decides L', using a Turing machine M that decides L as a black box. Typically this algorithm has the following form:

  Given a string w, transform it into another string x,
  such that M accepts x if and only if $w \in L'$.

- Prove that your Turing machine is correct. This almost always requires two separate steps:
  - Prove that if M accepts w then $w \in L'$.
  - Prove that if M rejects w then $w \notin L'$.

Prove that the following languages are undecidable:

1. $\textsc{AcceptIllini} := \big\{\langle M\rangle \bigm| M \text{ accepts the string } \texttt{ILLINI}\big\}$

2. $\textsc{AcceptThree} := \big\{\langle M\rangle \bigm| M \text{ accepts exactly three strings}\big\}$

3. $\textsc{AcceptPalindrome} := \big\{\langle M\rangle \bigm| M \text{ accepts at least one palindrome}\big\}$

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

Rice's Theorem. *Let $\mathcal{X}$ be any nonempty proper subset of the set of acceptable languages. The language* $\textsc{AcceptIn}\mathcal{X} := \big\{\langle M\rangle \bigm| \textsc{Accept}(M) \in \mathcal{X}\big\}$ *is undecidable.*

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 languages are undecidable? How do you prove it?

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

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, check "True" if the statement is ***always*** true and "False" otherwise. Each correct answer is worth +1 point; each incorrect answer is worth −½ point; checking "I don't know" is worth +¼ point; and flipping a coin is (on average) worth +¼ point. You do ***not*** need to prove your answer is correct.

   Read each statement *very* carefully. Some of these are deliberately subtle.

   (a) If $2+2=5$, then Jeff is the Queen of England.

   (b) For all languages L_1 and L_2, the language $L_1 \cup L_2$ is regular.

   (c) For all languages $L \subseteq \Sigma^*$, if L is not regular, then $\Sigma^* \setminus L$ cannot be represented by a regular expression.

   (d) For all languages L_1 and L_2, if $L_1 \subseteq L_2$ and L_1 is regular, then L_2 is regular.

   (e) For all languages L_1 and L_2, if $L_1 \subseteq L_2$ and L_1 is not regular, then L_2 is not regular.

   (f) For all languages L, if L is regular, then L has no infinite fooling set.

   (g) The language $\left\{\mathtt{0}^m\mathtt{1}^n \;\middle|\; 0 \le m+n \le 374\right\}$ is regular.

   (h) The language $\left\{\mathtt{0}^m\mathtt{1}^n \;\middle|\; 0 \le m-n \le 374\right\}$ is regular.

   (i) For every language L, if the language $L^R = \left\{w^R \;\middle|\; w \in L\right\}$ is regular, then L is also regular. (Here w^R denotes the reversal of string w; for example, $(\mathtt{BACKWARD})^R = \mathtt{DRAWKCAB}$.)

   (j) Every context-free language is regular.

2. Let L be the set of strings in $\{\mathtt{0},\mathtt{1}\}^*$ in which every run of consecutive $\mathtt{0}$s has even length and every run of consecutive $\mathtt{1}$s has odd length.

   (a) Give a regular expression that represents L.

   (b) Construct a DFA that recognizes L.

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

3. For each of the following languages over the alphabet $\{\mathtt{0},\mathtt{1}\}$, either ***prove*** that the language is regular or ***prove*** that the language is not regular. ***Exactly one of these two languages is regular.***

   (a) The set of all strings in which the substrings $\mathtt{00}$ and $\mathtt{11}$ appear the same number of times.

   (b) The set of all strings in which the substrings $\mathtt{01}$ and $\mathtt{10}$ appear the same number of times.

   For example, both of these languages contain the string $\mathtt{1100001101101}$.

4. Consider the following recursive function:

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

For example, $stutter(\texttt{00101}) = \texttt{0000110011}$.

Prove that for any regular language L, the following languages are also regular.

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

(b) $\textsc{Stutter}^{-1}(L) := \{w \mid stutter(w) \in L\}$.

5. Recall that string concatenation and string reversal are formally defined as follows:

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

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

Prove that $(w \bullet x)^R = x^R \bullet w^R$, for all strings w and x. Your proof should be complete, concise, formal, and self-contained.

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, check "True" if the statement is ***always*** true and "False" otherwise. Each correct answer is worth +1 point; each incorrect answer is worth −½ point; checking "I don't know" is worth +¼ point; and flipping a coin is (on average) worth +¼ point. You do ***not*** need to prove your answer is correct.

   Read each statement *very* carefully. Some of these are deliberately subtle.

   (a) If $2+2=5$, then Jeff is not the Queen of England.

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

   (c) For all languages $L \subseteq \Sigma^*$, if L is can be represented by a regular expression, then $\Sigma^* \setminus L$ can also be represented by a regular expression.

   (d) For all languages L_1 and L_2, if L_2 is regular and $L_1 \subseteq L_2$, then L_1 is regular.

   (e) For all languages L_1 and L_2, if L_2 is not regular and $L_1 \subseteq L_2$, then L_1 is not regular.

   (f) For all languages L, if L is not regular, then every fooling set for L is infinite.

   (g) The language $\left\{\mathtt{0}^m\mathtt{1}\mathtt{0}^n \mid 0 \le n-m \le 374\right\}$ is regular.

   (h) The language $\left\{\mathtt{0}^m\mathtt{1}\mathtt{0}^n \mid 0 \le n+m \le 374\right\}$ is regular.

   (i) For every language L, if L is not regular, then the language $L^R = \left\{w^R \mid w \in L\right\}$ is also not regular. (Here w^R denotes the reversal of string w; for example, $(\mathtt{BACKWARD})^R = \mathtt{DRAWKCAB}$.)

   (j) Every context-free language is regular.

2. Let L be the set of strings in $\{\mathtt{0},\mathtt{1}\}^*$ in which every run of consecutive 0s has odd length and the total number of 1s is even.

   For example, the string 11110000010111000 is in L, because it has eight 1s and three runs of consecutive 0s, with lengths 5, 1, and 3.

   (a) Give a regular expression that represents L.

   (b) Construct a DFA that recognizes L.

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

3. For each of the following languages over the alphabet $\{\mathtt{0},\mathtt{1}\}$, either ***prove*** that the language is regular or ***prove*** that the language is not regular. ***Exactly one of these two languages is regular.***

   (a) The set of all strings in which the substrings 10 and 01 appear the same number of times.

   (b) The set of all strings in which the substrings 00 and 01 appear the same number of times.

   For example, both of these languages contain the string 1100001101101.

4. Consider the following recursive function:

$$odds(w) := \begin{cases} w & \text{if } |w| \le 1 \\ a \cdot odds(x) & \text{if } w = abx \text{ for some } a, b \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

Intuitively, *odds* removes every other symbol from the input string, starting with the second symbol. For example, $odds(\texttt{0101110}) = \texttt{0010}$.

Prove that for any regular language L, the following languages are also regular.

(a) $\textsc{Odds}(L) := \{odds(w) \mid w \in L\}$.

(b) $\textsc{Odds}^{-1}(L) := \{w \mid odds(w) \in L\}$.

5. Recall that string concatenation and string reversal are formally defined as follows:

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

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

Prove that $(w \bullet x)^R = x^R \bullet w^R$, for all strings w and x. Your proof should be complete, concise, formal, and self-contained. You may assume the following identities, which we proved in class:

- $w \bullet (x \bullet y) = (w \bullet x) \bullet y$ for all strings w, x, and y.
- $|w \bullet x| = |w| + |x|$ for all strings w and x.

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

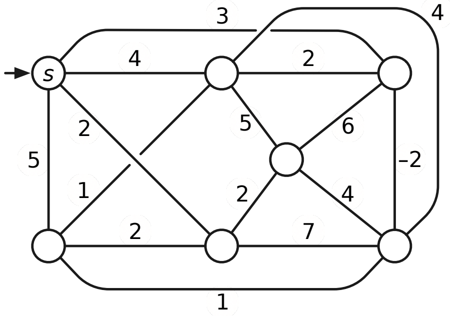
1. ***Clearly*** indicate the edges of the following spanning trees of the weighted graph pictured below. (Pretend that the person grading your exam has bad eyesight.) Some of these subproblems have more than one correct answer. Yes, that edge on the right has negative weight.

   (a) A depth-first spanning tree rooted at s

   (b) A breadth-first spanning tree rooted at s

   (c) A shortest-path tree rooted at s

   (d) A minimum spanning tree



2. 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

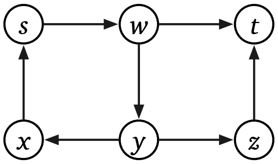
3	6	9	8	7	5	1	2	4

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.

3. 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.]

4. The new swap-puzzle 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.

5. Let G be a directed graph with weighted edges, and let s be a vertex of G. 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. Do ***not*** assume that G has no negative cycles.

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

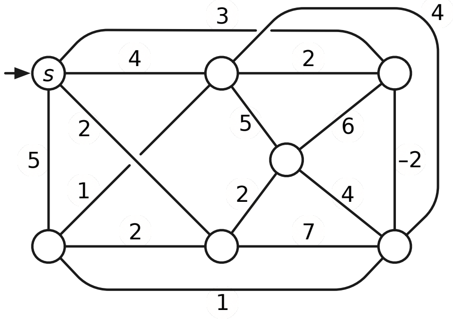
1. ***Clearly*** indicate the edges of the following spanning trees of the weighted graph pictured below. (Pretend that the person grading your exam has bad eyesight.) Some of these subproblems have more than one correct answer. Yes, that edge on the right has negative weight.

   (a) A depth-first spanning tree rooted at s

   (b) A breadth-first spanning tree rooted at s

   (c) A shortest-path tree rooted at s

   (d) A minimum spanning tree



2. 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.

3. Let G be a directed graph with weighted edges, and let s be a vertex of G. 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. Do ***not*** assume that G has no negative cycles.

4. 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. 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.)

“CS 374”: Algorithms and Models of Computation, Fall 2014

Final Exam — Version A — December 16, 2014

Name:	
NetID:	
Section:	1 2 3

#	1	2	3	4	5	6	Total
Score							
Max	20	10	10	10	10	10	70
Grader							

- ***Don’t panic!***
- Please print your name and your NetID and circle your discussion section in the boxes above.
- This is a closed-book, closed-notes, closed-electronics exam. If you brought anything except your writing implements and your two double-sided 8½" × 11" cheat sheets, please put it away for the duration of the exam. In particular, you may not use *any* electronic devices.
- **Please read the entire exam before writing anything.** Please ask for clarification if any question is unclear.
- **You have 180 minutes.**
- If you run out of space for an answer, continue on the back of the page, or on the blank pages at the end of this booklet, **but please tell us where to look.** Alternatively, feel free to tear out the blank pages and use them as scratch paper.
- **Please return your cheat sheets and all scratch paper with your answer booklet.**
- If you use a greedy algorithm, you must prove that it is correct to receive credit. **Otherwise, proofs are required only if we specifically ask for them.**
- As usual, answering any (sub)problem with “I don’t know” (and nothing else) is worth 25% partial credit. **Yes, even for problem 1.** Correct, complete, but suboptimal solutions are *always* worth more than 25%. A blank answer is not the same as “I don’t know”.
- ***Good luck!*** And have a great winter break!

1. For each of the following questions, indicate ***every*** correct answer by marking the "Yes" box, and indicate ***every*** incorrect answer by marking the "No" box. **Assume P ≠ NP.** If there is any other ambiguity or uncertainty, mark the "No" box. For example:

   [✓ Yes] No $2+2=4$

   Yes [✓ No] $x+y=5$

   Yes [✓ No] 3SAT can be solved in polynomial time.

   [✓ Yes] No Jeff is not the Queen of England.

   There are 40 yes/no choices altogether, each worth ½ point.

   (a) Which of the following statements is true for ***every*** language $L \subseteq \{\mathtt{0},\mathtt{1}\}^*$?

   Yes No L is non-empty.

   Yes No L is decidable or L is infinite (or both).

   Yes No L is accepted by some DFA with 42 states if and only if L is accepted by some NFA with 42 states.

   Yes No If L is regular, then $L \in$ NP.

   Yes No L is decidable if and only if its complement $\overline{L}$ is undecidable.

   (b) Which of the following computational models can be simulated by a deterministic Turing machine with three read/write heads, with at most polynomial slow-down in time, assuming P ≠ NP?

   Yes No A Java program

   Yes No A deterministic Turing machine with one head

   Yes No A deterministic Turing machine with 3 tapes, each with 5 heads

   Yes No A nondeterministic Turing machine with one head

   Yes No A nondeterministic finite-state automaton (NFA)

(c) Which of the following languages are decidable?

Yes	No	$\varnothing$
Yes	No	$\{\mathtt{0}^n\mathtt{1}^n\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$
Yes	No	$\{\langle M\rangle \mid M \text{ is a Turing machine}\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts } \langle M\rangle \bullet \langle M\rangle\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts a finite number of non-palindromes}\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts } \varnothing\}$
Yes	No	$\{\langle M, w\rangle \mid M \text{ accepts } w^R\}$
Yes	No	$\{\langle M, w\rangle \mid M \text{ accepts } w \text{ after at most } \lvert w\rvert^2 \text{ transitions}\}$
Yes	No	$\{\langle M, w\rangle \mid M \text{ changes a blank on the tape to a non-blank, given input } w\}$
Yes	No	$\{\langle M, w\rangle \mid M \text{ changes a non-blank on the tape to a blank, given input } w\}$

(d) Let M be a standard Turing machine (with a single one-track tape and a single head) that ***decides*** the regular language $\mathtt{0}^*\mathtt{1}^*$. Which of the following ***must*** be true?

Yes	No	Given an empty initial tape, M eventually halts.
Yes	No	M accepts the string 1111.
Yes	No	M rejects the string 0110.
Yes	No	M moves its head to the right at least once, given input 1100.
Yes	No	M moves its head to the right at least once, given input 0101.
Yes	No	M never accepts before reading a blank.
Yes	No	For some input string, M moves its head to the left at least once.
Yes	No	For some input string, M changes at least one symbol on the tape.
Yes	No	M always halts.
Yes	No	If M accepts a string w, it does so after at most $O(\lvert w\rvert^2)$ steps.

(e) Consider the following pair of languages:

- HamiltonianPath := $\{G \mid G \text{ contains a Hamiltonian path}\}$
- Connected := $\{G \mid G \text{ is connected}\}$

Which of the following ***must*** be true, assuming P≠NP?

Yes	No	Connected ∈ NP
Yes	No	HamiltonianPath ∈ NP
Yes	No	HamiltonianPath is decidable.
Yes	No	There is no polynomial-time reduction from HamiltonianPath to Connected.
Yes	No	There is no polynomial-time reduction from Connected to HamiltonianPath.

(f) Suppose we want to prove that the following language is undecidable.

$$\text{AlwaysHalts} := \{\langle M\rangle \mid M \text{ halts on every input string}\}$$

Rocket J. Squirrel suggests a reduction from the standard halting language

$$\text{Halt} := \{\langle M, w\rangle \mid M \text{ halts on inputs } w\}.$$

Specifically, suppose there is a Turing machine AH that decides AlwaysHalts. Rocky claims that the following Turing machine H decides Halt. Given an arbitrary encoding $\langle M, w\rangle$ as input, machine H writes the encoding $\langle M'\rangle$ of a new Turing machine M' to the tape and passes it to AH, where M' implements the following algorithm:

```
M'(x):
  if M accepts w
      reject
  if M rejects w
      accept
```

Which of the following statements is true for all inputs $\langle M, w\rangle$?

Yes	No	If M accepts w, then M' halts on every input string.
Yes	No	If M diverges on w, then M' halts on every input string.
Yes	No	If M accepts w, then AH accepts $\langle M'\rangle$.
Yes	No	If M rejects w, then H rejects $\langle M, w\rangle$.
Yes	No	H decides the language Halt. (That is, Rocky’s reduction is correct.)

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. Give a complete, formal, self-contained description of a DFA that accepts all strings in $\{0, 1\}^*$ containing at least ten 0s and at most ten 1s. Specifically:

   (a) What are the states of your DFA?

   (b) What is the start state of your DFA?

   (c) What are the accepting states of your DFA?

   (d) What is your DFA's transition function?

4. Suppose you are given three strings $A[1\,..\,n]$, $B[1\,..\,n]$, and $C[1\,..\,n]$. Describe and analyze an algorithm to find the maximum length of a common subsequence of all three strings. For example, given the input strings

   $A = \texttt{AxxBxxCDxEF}$, $B = \texttt{yyABCDyEyFy}$, $C = \texttt{zAzzBCDzEFz}$,

   your algorithm should output the number 6, which is the length of the longest common subsequence `ABCDEF`.

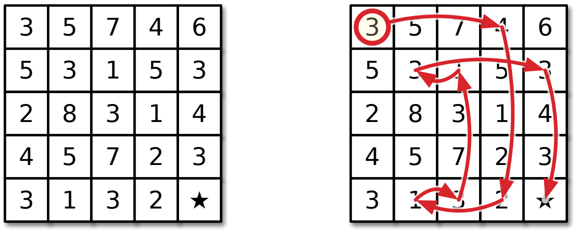
5. For each of the following languages over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, either ***prove*** that the language is regular, or ***prove*** that the language is not regular.

   (a) $\left\{ www \;\middle|\; w \in \Sigma^* \right\}$

   (b) $\left\{ wxw \;\middle|\; w, x \in \Sigma^* \right\}$

6. 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.

(scratch paper)

(scratch paper)

You may assume the following problems are 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 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?

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 an undirected graph G, is there a path in G that visits every vertex exactly once?

HAMILTONIANCYCLE: Given an undirected graph G, is there a cycle in G that visits every vertex exactly once?

DIRECTEDHAMILTONIANCYCLE: Given an directed graph G, is there a directed 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?

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?

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

You may assume the following languages are undecidable:

$$
\begin{aligned}
\textsc{SelfReject} &:= \{\langle M\rangle \mid M \text{ rejects } \langle M\rangle\}\\
\textsc{SelfAccept} &:= \{\langle M\rangle \mid M \text{ accepts } \langle M\rangle\}\\
\textsc{SelfHalt} &:= \{\langle M\rangle \mid M \text{ halts on } \langle M\rangle\}\\
\textsc{SelfDiverge} &:= \{\langle M\rangle \mid M \text{ does not halt on } \langle M\rangle\}\\
\textsc{Reject} &:= \{\langle M, w\rangle \mid M \text{ rejects } w\}\\
\textsc{Accept} &:= \{\langle M, w\rangle \mid M \text{ accepts } w\}\\
\textsc{Halt} &:= \{\langle M, w\rangle \mid M \text{ halts on } w\}\\
\textsc{Diverge} &:= \{\langle M, w\rangle \mid M \text{ does not halt on } w\}\\
\textsc{NeverReject} &:= \{\langle M\rangle \mid \textsc{Reject}(M) = \varnothing\}\\
\textsc{NeverAccept} &:= \{\langle M\rangle \mid \textsc{Accept}(M) = \varnothing\}\\
\textsc{NeverHalt} &:= \{\langle M\rangle \mid \textsc{Halt}(M) = \varnothing\}\\
\textsc{NeverDiverge} &:= \{\langle M\rangle \mid \textsc{Diverge}(M) = \varnothing\}
\end{aligned}
$$

“CS 374”: Algorithms and Models of Computation, Fall 2014

Final Exam (Version B) — December 16, 2014

Name:			
NetID:			
Section:	1	2	3

#	1	2	3	4	5	6	Total
Score							
Max	20	10	10	10	10	10	70
Grader							

- ***Don’t panic!***
- Please print your name and your NetID and circle your discussion section in the boxes above.
- This is a closed-book, closed-notes, closed-electronics exam. If you brought anything except your writing implements and your two double-sided 8½" × 11" cheat sheets, please put it away for the duration of the exam. In particular, you may not use *any* electronic devices.
- **Please read the entire exam before writing anything.** Please ask for clarification if any question is unclear.
- **You have 180 minutes.**
- If you run out of space for an answer, continue on the back of the page, or on the blank pages at the end of this booklet, **but please tell us where to look.** Alternatively, feel free to tear out the blank pages and use them as scratch paper.
- **Please return your cheat sheets and all scratch paper with your answer booklet.**
- If you use a greedy algorithm, you must prove that it is correct to receive credit. **Otherwise, proofs are required only if we specifically ask for them.**
- As usual, answering any (sub)problem with “I don’t know” (and nothing else) is worth 25% partial credit. **Yes, even for problem 1.** Correct, complete, but suboptimal solutions are *always* worth more than 25%. A blank answer is not the same as “I don’t know”.
- ***Good luck!*** And have a great winter break!

1. For each of the following questions, indicate ***every*** correct answer by marking the "Yes" box, and indicate ***every*** incorrect answer by marking the "No" box. **Assume P ≠ NP.** If there is any other ambiguity or uncertainty, mark the "No" box. For example:

   ☒ Yes □ No $2+2=4$

   □ Yes ☒ No $x+y=5$

   □ Yes ☒ No 3SAT can be solved in polynomial time.

   ☒ Yes □ No Jeff is not the Queen of England.

   There are 40 yes/no choices altogether, each worth ½ point.

(a) Which of the following statements is true for ***every*** language $L \subseteq \{0,1\}^*$?

□ Yes □ No L is non-empty.

□ Yes □ No L is decidable or L is infinite (or both).

□ Yes □ No L is accepted by some DFA with 42 states if and only if L is accepted by some NFA with 42 states.

□ Yes □ No If L is regular, then $L \in$ NP.

□ Yes □ No L is decidable if and only if its complement $\overline{L}$ is undecidable.

(b) Which of the following computational models can simulate a deterministic Turing machine with three read/write heads, with at most polynomial slow-down in time, assuming P ≠ NP?

□ Yes □ No A C++ program

□ Yes □ No A deterministic Turing machine with one head

□ Yes □ No A deterministic Turing machine with 3 tapes, each with 5 heads

□ Yes □ No A nondeterministic Turing machine with one head

□ Yes □ No A nondeterministic finite-state automaton (NFA)

(c) Which of the following languages are decidable?

Yes	No	$\varnothing$
Yes	No	$\{ww \mid w \text{ is a palindrome}\}$
Yes	No	$\{\langle M\rangle \mid M \text{ is a Turing machine}\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts } \langle M\rangle \bullet \langle M\rangle\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts an infinite number of palindromes}\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts } \varnothing\}$
Yes	No	$\{\langle M,w\rangle \mid M \text{ accepts } www\}$
Yes	No	$\{\langle M,w\rangle \mid M \text{ accepts } w \text{ after } \textbf{at least } \lvert w\rvert^2 \text{ transitions}\}$
Yes	No	$\{\langle M,w\rangle \mid M \text{ changes a non-blank on the tape to a blank, given input } w\}$
Yes	No	$\{\langle M,w\rangle \mid M \text{ changes a blank on the tape to a non-blank, given input } w\}$

(d) Let M be a standard Turing machine (with a single one-track tape and a single head) such that $\textsc{Accept}(M)$ is the regular language $\mathtt{0}^*\mathtt{1}^*$. Which of the following ***must*** be true?

Yes	No	Given an empty initial tape, M eventually halts.
Yes	No	M accepts the string `1111`.
Yes	No	M rejects the string `0110`.
Yes	No	M moves its head to the right at least once, given input `1100`.
Yes	No	M moves its head to the right at least once, given input `0101`.
Yes	No	M must read a blank before it accepts.
Yes	No	For some input string, M moves its head to the left at least once.
Yes	No	For some input string, M changes at least one symbol on the tape.
Yes	No	M always halts.
Yes	No	If M accepts a string w, it does so after at most $O(\lvert w\rvert^2)$ steps.

(e) Consider the following pair of languages:

- HamiltonianPath $:= \{G \mid G \text{ contains a Hamiltonian path}\}$
- Connected $:= \{G \mid G \text{ is connected}\}$

Which of the following ***must*** be true, assuming P≠NP?

Yes	No	Connected $\in$ NP
Yes	No	HamiltonianPath $\in$ NP
Yes	No	HamiltonianPath is undecidable.
Yes	No	There is a polynomial-time reduction from HamiltonianPath to Connected.
Yes	No	There is a polynomial-time reduction from Connected to HamiltonianPath.

(f) Suppose we want to prove that the following language is undecidable.

$$\text{AlwaysHalts} := \{\langle M\rangle \mid M \text{ halts on every input string}\}$$

Bullwinkle J. Moose suggests a reduction from the standard halting language

$$\text{Halt} := \{\langle M, w\rangle \mid M \text{ halts on inputs } w\}.$$

Specifically, suppose there is a Turing machine AH that decides AlwaysHalts. Bullwinkle claims that the following Turing machine H decides Halt. Given an arbitrary encoding $\langle M, w\rangle$ as input, machine H writes the encoding $\langle M'\rangle$ of a new Turing machine M' to the tape and passes it to AH, where M' implements the following algorithm:

```
$M'(x)$:
  if $M$ accepts $w$
      reject
  if $M$ rejects $w$
      accept
```

Which of the following statements is true for all inputs $\langle M, w\rangle$?

Yes	No	If M accepts w, then M' halts on every input string.
Yes	No	If M rejects w, then M' halts on every input string.
Yes	No	If M rejects w, then H rejects $\langle M, w\rangle$.
Yes	No	If M diverges on w, then H diverges on $\langle M, w\rangle$.
Yes	No	H does not correctly decide the language Halt. (That is, Bullwinkle's reduction is incorrect.)

2. A ***near-Hamiltonian cycle*** in a graph G is a closed walk in G that visits one vertex exactly twice and every other vertex exactly once.

   (a) Give an example of a graph that contains a near-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 near-Hamiltonian cycle.

3. Give a complete, formal, self-contained description of a DFA that accepts all strings in $\{0,1\}^*$ such that every fifth bit is 0 and the length is *not* divisible by 12. For example, your DFA should accept the strings 11110111101 and 11. Specifically:

   (a) What are the states of your DFA?

   (b) What is the start state of your DFA?

   (c) What are the accepting states of your DFA?

   (d) What is your DFA's transition function?

4. Suppose you are given three strings $A[1..n]$, $B[1..n]$, and $C[1..n]$. Describe and analyze an algorithm to find the maximum length of a 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.

5. 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.

   (a) $\left\{www \;\middle|\; w \in \Sigma^*\right\}$

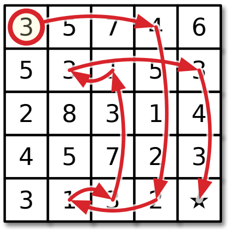
   (b) $\left\{wxw \;\middle|\; w, x \in \Sigma^*\right\}$

6. 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. In particular, if the current number is too large, you may not be able to move at all.

   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.

(scratch paper)

(scratch paper)

You may assume the following problems are 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 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?

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 an undirected graph G, is there a path in G that visits every vertex exactly once?

HamiltonianCycle: Given an undirected graph G, is there a cycle in G that visits every vertex exactly once?

DirectedHamiltonianCycle: Given an directed graph G, is there a directed 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?

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?

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

You may assume the following languages are undecidable:

$$\begin{aligned}
\textsc{SelfReject} &:= \big\{\langle M\rangle \bigm| M \text{ rejects } \langle M\rangle\big\}\\
\textsc{SelfAccept} &:= \big\{\langle M\rangle \bigm| M \text{ accepts } \langle M\rangle\big\}\\
\textsc{SelfHalt} &:= \big\{\langle M\rangle \bigm| M \text{ halts on } \langle M\rangle\big\}\\
\textsc{SelfDiverge} &:= \big\{\langle M\rangle \bigm| M \text{ does not halt on } \langle M\rangle\big\}\\
\textsc{Reject} &:= \big\{\langle M,w\rangle \bigm| M \text{ rejects } w\big\}\\
\textsc{Accept} &:= \big\{\langle M,w\rangle \bigm| M \text{ accepts } w\big\}\\
\textsc{Halt} &:= \big\{\langle M,w\rangle \bigm| M \text{ halts on } w\big\}\\
\textsc{Diverge} &:= \big\{\langle M,w\rangle \bigm| M \text{ does not halt on } w\big\}\\
\textsc{NeverReject} &:= \big\{\langle M\rangle \bigm| \textsc{Reject}(M) = \varnothing\big\}\\
\textsc{NeverAccept} &:= \big\{\langle M\rangle \bigm| \textsc{Accept}(M) = \varnothing\big\}\\
\textsc{NeverHalt} &:= \big\{\langle M\rangle \bigm| \textsc{Halt}(M) = \varnothing\big\}\\
\textsc{NeverDiverge} &:= \big\{\langle M\rangle \bigm| \textsc{Diverge}(M) = \varnothing\big\}
\end{aligned}$$

New CS 473: Algorithms, Spring 2015
Homework 0

Due Tuesday, January 27, 2015 at 5pm

- **This homework tests your familiarity with prerequisite material:** designing, describing, and analyzing elementary algorithms (at the level of CS 225); fundamental graph problems and algorithms (again, at the level of CS 225); and especially facility with recursion and induction. Notes on most of this prerequisite material are available on the course web page.
- **Each student must submit individual solutions for this homework.** For all future homeworks, groups of up to three students will be allowed to submit joint solutions.

☞ Some important course policies ☜

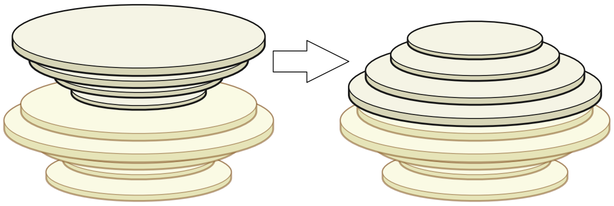
- **You may use any source at your disposal**—paper, electronic, or human—but you ***must*** cite ***every*** source that you use, and you ***must*** write everything yourself in your own words. See the academic integrity policies on the course web site for more details.
- **Submit your solutions on standard printer/copier paper.** Please use both sides of the paper. Please clearly print your name and NetID at the top of each page. If you plan to write your solutions by hand, please print the last four pages of this homework as templates. If you plan to typeset your homework, you can find a LaTeX template on the course web site; well-typeset homework will get a small amount of extra credit.
- **Start your solution to each numbered problem on a new sheet of paper.** Do not staple your entire homework together.
- **Submit your solutions in the drop boxes outside 1404 Siebel labeled "New CS 473".** There is a separate drop box for each numbered problem; if you put your solution in the wrong drop box, we won't grade it. Don't give your homework to Jeff in class; he is fond of losing important pieces of paper.
- **Avoid the Three Deadly Sins!** There are a few dangerous writing (and thinking) habits that will trigger an automatic zero on any homework or exam problem, unless your solution is nearly perfect otherwise. Yes, we are completely serious.
  - Always give complete solutions, not just examples.
  - Always declare all your variables.
  - Never use weak induction.
- Answering any homework or exam problem (or subproblem) in this course with "I don't know" *and nothing else* is worth 25% partial credit. We will accept synonyms like "No idea" or "WTF", but you must write *something*.

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. 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.



Flipping the top four pancakes.

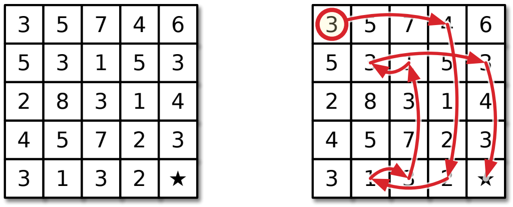
   (a) Describe an algorithm to sort an arbitrary stack of n pancakes, which uses as few flips as possible in the worst case. *Exactly* how many flips does your algorithm perform in the worst case?

   (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 in the worst case. *Exactly* how many flips does your algorithm perform in the worst case?

2. *[From last semester's CS 374 final exam]* 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. In particular, if the current number is too large, you may not be able to move at all.

   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.

[Hint: Build a graph. What are the vertices? What are the edges? Is the graph directed or undirected? Do the vertices or edges have weights? If so, what are they? What textbook problem do you need to solve on this graph? What textbook algorithm should you use to solve that problem? What is the running time of that algorithm as a function of n?]

3. (a) The ***Fibonacci numbers*** $\boldsymbol{F_n}$ are defined by the recurrence $F_n = F_{n-1} + F_{n-2}$, with base cases $F_0 = 0$ and $F_1 = 1$. Here are the first several Fibonacci numbers:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

Prove that every non-negative integer can be written as the sum of distinct, non-consecutive Fibonacci numbers. That is, if the Fibonacci number F_i appears in the sum, it appears exactly once, and its neighbors F_{i-1} and F_{i+1} do not appear at all. For example:

$$17 = F_7 + F_4 + F_2, \qquad 42 = F_9 + F_6, \qquad 54 = F_9 + F_7 + F_5 + F_3 + F_1.$$

(b) The Fibonacci sequence can be extended backward to negative indices by rearranging the defining recurrence: $F_n = F_{n+2} - F_{n+1}$. Here are the first several negative-index Fibonacci numbers:

F_{-10}	F_{-9}	F_{-8}	F_{-7}	F_{-6}	F_{-5}	F_{-4}	F_{-3}	F_{-2}	F_{-1}
−55	34	−21	13	−8	5	−3	2	−1	1

Prove that $F_{-n} = -F_n$ if and only if n is even.

(c) Prove that *every* integer—positive, negative, or zero—can be written as the sum of distinct, non-consecutive Fibonacci numbers *with negative indices*. For example:

$$17 = F_{-7} + F_{-5} + F_{-2}, \quad -42 = F_{-10} + F_{-7}, \quad 54 = F_{-9} + F_{-7} + F_{-5} + F_{-3} + F_{-1}.$$

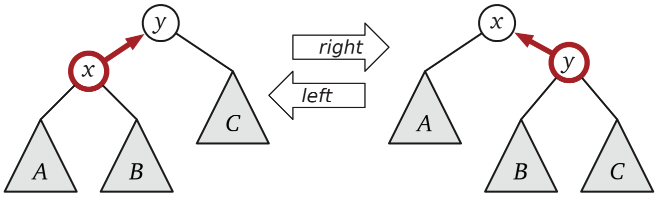
[Hint: Zero is both non-negative and even. Don't even think about using weak induction!]

4. ***[Extra credit]*** Let T be a binary tree whose nodes store distinct numerical values. Recall that T is a ***binary search tree*** if and only if either (1) T is empty, or (2) T satisfies the following recursive conditions:

- The left subtree of T is a binary search tree.
- All values in the left subtree of T are smaller than the value at the root of T.
- The right subtree of T is a binary search tree.
- All values in the right subtree of T are larger than the value at the root of T.

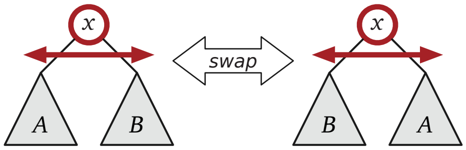
Describe and analyze an algorithm to transform an *arbitrary* binary tree T with distinct node values into a binary search tree, using ***only*** the following operations:

- Rotate an arbitrary node. Rotation is a local operation that decreases the depth of a node by one and increases the depth of its parent by one.



Left to right: right rotation at x. Right to left: left rotation at y.

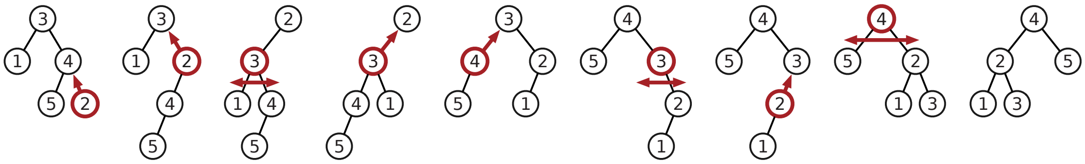
- Swap the left and right subtrees of an arbitrary node.



Swapping the subtrees of x

For both of these operations, some, all, or none of the subtrees A, B, and C may be empty.

The following example shows a five-node binary tree transforming into a binary search tree in eight operations:



"Sorting" a binary tree in eight steps: rotate 2, rotate 2, swap 3, rotate 3, rotate 4, swap 3, rotate 2, swap 4.

Your algorithm cannot directly modify parent or child pointers, and it cannot allocate new nodes or delete old nodes; the ***only*** way it can modify T is using rotations and swaps. On the other hand, you may *compute* anything you like for free, as long as that computation does not modify T. In other words, the running time of your algorithm is *defined* to be the number of rotations and swaps that it performs.

For full credit, your algorithm should use as few rotations and swaps as possible in the worst case. *[Hint: $O(n^2)$ operations is not too difficult, but we can do better.]*

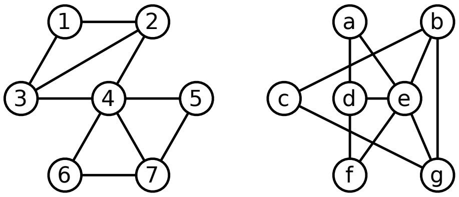
New CS 473: Algorithms, Spring 2015
Homework 1

Due Tuesday, February 3, 2015 at 5pm

For this and all future homeworks, groups of up to three students can submit joint solutions. Please print (or typeset) the name and NetID of every group member on the first page of each submission.

---

1. Two graphs are ***isomorphic*** if one can be transformed into the other just by relabeling the vertices. For example, the graphs shown below are isomorphic; the left graph can be transformed into the right graph by the relabeling $(1,2,3,4,5,6,7) \mapsto (c,g,b,e,a,f,d)$.



Two isomorphic graphs.

Consider the following related decision problems:

- GraphIsomorphism: Given two graphs G and H, determine whether G and H are isomorphic.
- EvenGraphIsomorphism: Given two graphs G and H, such that every vertex in G and every vertex in H has even degree, determine whether G and H are isomorphic.
- SubgraphIsomorphism: Given two graphs G and H, determine whether G is isomorphic to a subgraph of H.

(a) Describe a polynomial-time reduction from EvenGraphIsomorphism to GraphIsomorphism.

(b) Describe a polynomial-time reduction from GraphIsomorphism to EvenGraphIsomorphism.

(c) Describe a polynomial-time reduction from GraphIsomorphism to SubgraphIsomorphism.

(d) Prove that SubgraphIsomorphism is NP-complete.

(e) What can you conclude about the NP-hardness of GraphIsomorphism? Justify your answer.

[Hint: These are all easy!]

2. Prove that the following problems are NP-hard.

   (a) Given an undirected graph G, does G have a spanning tree with at most 473 leaves?

   (b) Given an undirected graph $G = (V, E)$, what is the size of the largest subset of vertices $S \subseteq V$ such that at most 2015 edges in E have both endpoints in S?

3. The Hamiltonian cycle problem has two closely related variants:

   - UNDIRECTEDHAMCYCLE: Given an *undirected* graph G, does G contain an *undirected* Hamiltonian cycle?
   - DIRECTEDHAMCYCLE: Given an *directed* graph G, does G contain a *directed* Hamiltonian cycle?

   This question asks you to prove that these two problems are essentially equivalent.

   (a) Describe a polynomial-time reduction from UNDIRECTEDHAMCYCLE to DIRECTEDHAMCYCLE.

   (b) Describe a polynomial-time reduction from DIRECTEDHAMCYCLE to UNDIRECTEDHAMCYCLE.

*4. ***[Extra Credit]*** Describe a direct polynomial-time reduction from 4COLOR to 3COLOR. (This is a lot harder than the opposite direction!)

New CS 473: Algorithms, Spring 2015
Homework 2

Due Tuesday, February 10, 2015 at 5pm

---

1. The ***maximum k-cut problem*** asks, given a graph G with edge weights and an integer k as input, to compute a partition of the vertices of G into k disjoint subsets $S_1, S_2, \ldots, S_k$ such that the sum of the weights of the edges that cross the partition (that is, have endpoints in different subsets) is as large as possible.

   (a) Describe an efficient $(1 - 1/k)$-approximation algorithm for this problem.

   (b) Now suppose we wish to minimize the sum of the weights of edges that do *not* cross the partition. What approximation ratio does your algorithm from part (a) achieve for the new problem? Justify your answer.

2. In the ***bin packing*** problem, we are given a set of n items, each with weight between 0 and 1, and we are asked to load the items into as few bins as possible, such that the total weight in each bin is at most 1. It's not hard to show that this problem is NP-Hard; this question asks you to analyze a few common approximation algorithms. In each case, the input is an array $W[1..n]$ of weights, and the output is the number of bins used.

```
NextFit(W[1..n]):
  $bins \gets 0$
  $Total[0] \gets \infty$
  for $i \gets 1$ to $n$
      if $Total[bins] + W[i] > 1$
          $bins \gets bins + 1$
          $Total[bins] \gets W[i]$
      else
          $Total[bins] \gets Total[bins] + W[i]$
  return $bins$
```

```
FirstFit(W[1..n]):
  $bins \gets 0$
  for $i \gets 1$ to $n$
      $j \gets 1$; $found \gets$ False
      while $j \le bins$ and $\neg found$
          if $Total[j] + W[i] \le 1$
              $Total[j] \gets Total[j] + W[i]$
              $found \gets$ True
          $j \gets j + 1$
      if $\neg found$
          $bins \gets bins + 1$
          $Total[bins] = W[i]$
  return $bins$
```

   (a) Prove that NextFit uses at most twice the optimal number of bins.

   (b) Prove that FirstFit uses at most twice the optimal number of bins.

   *(c) ***[Extra Credit]*** Prove that if the weight array W is initially sorted in decreasing order, then FirstFit uses at most $(4 \cdot OPT + 1)/3$ bins, where OPT is the optimal number of bins. The following facts may be useful (but you need to prove them if your proof uses them):

   - In the packing computed by FirstFit, every item with weight more than $1/3$ is placed in one of the first OPT bins.
   - FirstFit places at most $OPT - 1$ items outside the first OPT bins.

3. Consider the following greedy algorithm for the metric traveling salesman problem: Start at an arbitrary vertex, and then repeatedly travel to the closest unvisited vertex, until every vertex has been visited.

   (a) Prove that the approximation ratio for this algorithm is $O(\log n)$, where n is the number of vertices. *[Hint: Argue that the kth least expensive edge in the tour output by the greedy algorithm has weight at most* $\mathrm{OPT}/(n-k+1)$*; try* $k=1$ *and* $k=2$ *first.]*

   ★(b) ***[Extra Credit]*** Prove that the approximation ratio for this algorithm is $\Omega(\log n)$. That is, describe an infinite family of weighted graphs such that the greedy algorithm returns a Hamiltonian cycle whose weight is $\Omega(\log n)$ times the weight of the optimal TSP tour.

New CS 473: Algorithms, Spring 2015
Homework 3

Due Tuesday, February 17, 2015 at 5pm

---

1. A string w of parentheses **(** and **)** and brackets **[** and **]** is ***balanced*** if it satisfies one of the following conditions:

   - w is the empty string.
   - $w = (x)$ for some balanced string x
   - $w = [x]$ for some balanced string x
   - $w = xy$ for some balanced strings x and y

   For example, the string

   $$w = ([()][]())[()()]()$$

   is balanced, because $w = xy$, where

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

   Describe and analyze an algorithm to compute the length of a longest balanced subsequence of a given string of parentheses and brackets. Your input is an array $w[1..n]$, where $w[i] \in \{(,), [,]\}$ for every index i. (You may prefer to use different symbols instead of parentheses and brackets—for example, L, R, l, r or ◁, ▷, ◀, ▶—but please tell us what symbols you're using!)

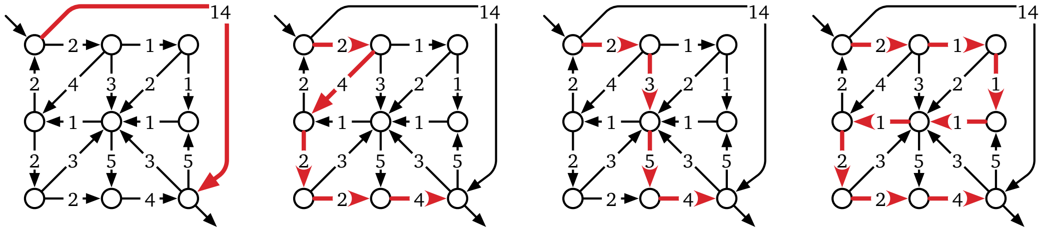
2. Congratulations! You've just been hired at internet giant Yeehaw! as the new party czar. The president of the company has asked you to plan the annual holiday party. Your task is to find exactly k employees to invite, including the president herself. Employees at Yeehaw! are organized into a strict hierarchy—a tree with the president at the root. The all-knowing oracles in Human Resources have determined two numerical values for every employee:

   - $\mathit{With}[i]$ measures much fun employee i would have at the party if their immediate supervisor is also invited.
   - $\mathit{Without}[i]$ measures how much fun employee i would have at the party if their immediate supervisor is *not* also invited.

   These values could be positive, negative, or zero, and $\mathit{With}[i]$ could be greater than, less than, or equal to $\mathit{Without}[i]$.

   Describe an algorithm that finds the set of k employees to invite that maximizes the sum of the k resulting "fun" values. The input to your algorithm is the tree T, the integer k, and the *With* and *Without* values for each employee. Assume that everyone invited to the party actually attends. Do *not* assume that T is a *binary* tree.

3. Although we typically speak of “the” shortest path between two nodes, 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 determine the *number* of shortest paths from a source vertex s to a target vertex t in an arbitrary directed graph G with weighted edges. You may assume that all edge weights are positive and that all necessary arithmetic operations can be performed in $O(1)$ time.

[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?]

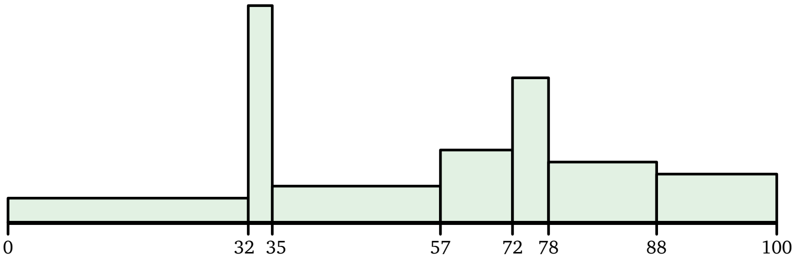
New CS 473: Algorithms, Spring 2015
Homework 4

Due Tuesday, February 24, 2015 at 5pm

Starting with this assignment, all homework must be submitted electronically via Moodle as separate PDF files, one for each numbered problem. Please see the course web site for more information.

---

1. Suppose we want to summarize a large set S of values—for example, course averages for students in CS 105—using a variable-width histogram. To construct a histogram, we choose a sorted sequence of ***breakpoints*** $b_0 < b_1 < \cdots < b_k$, such that every element of S lies between b_0 and b_k. Then for each interval $[b_{i-1}, b_i]$, the histogram includes a rectangle whose height is the number of elements of S that lie inside that interval.



A variable-width histogram with seven bins.

Unlike a standard histogram, which requires the intervals to have equal width, we are free to choose the breakpoints arbitrarily. For statistical purposes, it is useful for the *areas* of the rectangles to be as close to equal as possible. To that end, define the ***cost*** of a histogram to be the sum of the *squares* of the rectangle areas; we want to compute the histogram with minimum cost.

More formally, suppose we fix a sequence of breakpoints $b_0 < b_1 < \cdots < b_k$. For each index i, let n_i denote the number of input values in the ith interval:

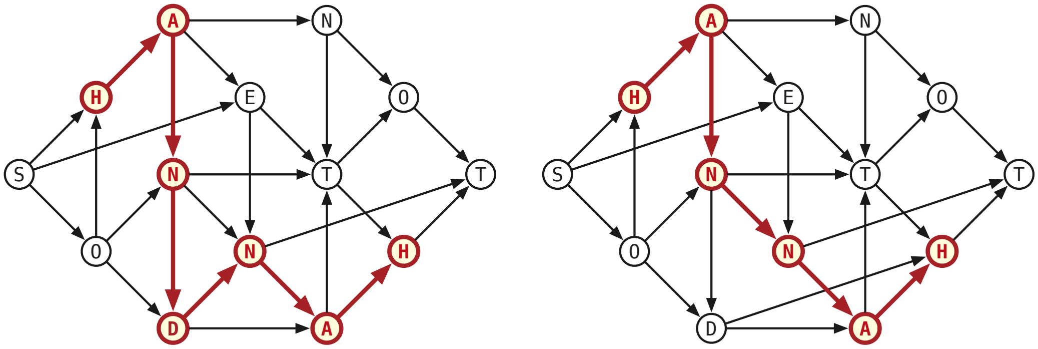
$$n_i := \#\left\{x \in S \;\middle|\; b_{i-1} \le x < b_i\right\}.$$

Then the ***cost*** of the resulting histogram is $\sum_{i=1}^{k}(n_i(b_i - b_{i-1}))^2$.

Describe and analyze an algorithm to compute a variable-width histogram with minimum cost for a given set of data values. Your input is an unsorted array $S[1..n]$ of ***distinct*** real numbers, all strictly between 0 and 1, and an integer k. Your algorithm should return a sorted array $B[0..k]$ of breakpoints that minimizes the cost of the resulting histogram, where $B[0] = 0$ and $B[k] = 1$, ***and every other breakpoint $B[i]$ is equal to some input value $S[j]$***.

2. Suppose we are given a directed acyclic graph G with labeled vertices. Every path in G has a label, which is a string obtained by concatenating the labels of its vertices in order. 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 graph on the left below, your algorithm should return the integer 7, which is the length of the palindrome `HANDNAH`; given the graph on the right, your algorithm should return the integer 6, which is the length of the palindrome `HANNAH`.



New CS 473: Algorithms, Spring 2015
Homework 5

Due Tuesday, March 10, 2015 at 5pm

All homework must be submitted electronically via Moodle as separate PDF files, one for each numbered problem. Please see the course web site for more information.

---

1. On their long journey from Denmark to England, Rosencrantz and Guildenstern amuse themselves by playing the following game with a fair coin. First Rosencrantz flips the coin over and over until it comes up tails. Then Guildenstern flips the coin over and over until he gets as many heads in a row as Rosencrantz got on his turn. Here are three typical games:

   Rosencrantz: **H** **H** T
   Guildenstern: H T **H** **H**

   Rosencrantz: T
   Guildenstern: (no flips)

   Rosencrantz: **H** **H** **H** T
   Guildenstern: T H H T H H T H T T **H** **H** **H**

   (a) What is the *exact* expected number of flips in one of Rosencrantz's turns?

   (b) Suppose Rosencrantz happens to flip k heads in a row on his turn. What is the *exact* expected number of flips in Guildenstern's next turn?

   (c) What is the *exact* expected total number of flips (by both Rosencrantz and Guildenstern) in a single game?

   Include formal proofs that your answers are correct. If you have to appeal to "intuition" or "common sense", your answer is almost certainly wrong!

2. Recall from class that a ***priority search tree*** is a binary tree in which every node has both a *search key* and a *priority*, arranged so that the tree is simultaneously a binary search tree for the keys and a min-heap for the priorities. A ***heater*** is a priority search tree in which the *priorities* are given by the user, and the *search keys* are distributed uniformly and independently at random in the real interval $[0,1]$. Intuitively, a heater is an "anti-treap".

   The following questions consider an n-node heater T whose priorities are the integers from 1 to n. Here we identify each node in T by its ***priority rank***, rather than by the rank of its search keys; for example, "node 5" means the node in T with the 5th smallest *priority*. In particular, the min-heap property implies that node 1 is the root of T. Finally, let i and j be arbitrary integers such that $1 \leq i < j \leq n$.

(a) What is the *exact* expected depth of node j in an n-node heater? Answering the following subproblems will help you:

i. Prove that in a uniformly random permutation of the $(i+1)$-element set $\{1, 2, \ldots, i, j\}$, elements i and j are adjacent with probability $2/(i+1)$.

ii. Prove that node i is an ancestor of node j with probability $2/(i+1)$. *[Hint: Use the previous question!]*

iii. What is the probability that node i is a *descendant* of node j? *[Hint: Do **not** use the previous question!]*

(b) Describe and analyze an algorithm to insert a new item into a heater. Express the expected running time of the algorithm in terms of the priority rank of the newly inserted item.

(c) Describe an algorithm to delete the minimum-priority item (the root) from an n-node heater. What is the expected running time of your algorithm?

3. Suppose we are given a two-dimensional array $M[1..n, 1..n]$ in which every row and every column is sorted in increasing order and no two elements are equal.

(a) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, compute the number of elements of M smaller than $M[i, j]$ and larger than $M[i', j']$.

(b) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, return an element of M chosen uniformly at random from the elements smaller than $M[i, j]$ and larger than $M[i', j']$. Assume the requested range is always non-empty.

(c) Describe and analyze a randomized algorithm to compute the median element of M in $O(n \log n)$ expected time.

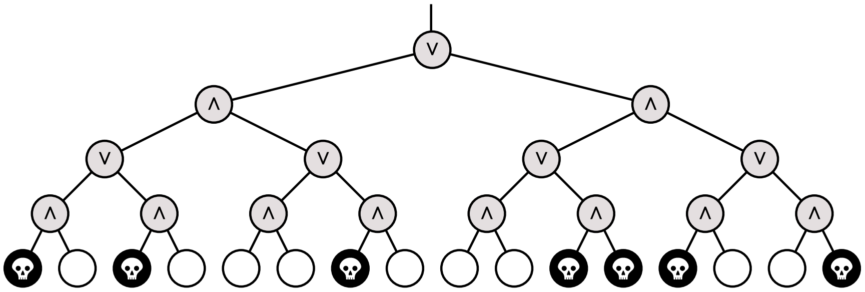
Assume you have access to a subroutine Random(k) that returns an integer chosen independently and uniformly at random from the set $\{1 2, \ldots, k\}$, given an arbitrary positive integer k as input.

New CS 473: Algorithms, Spring 2015
Homework 6

Due Tuesday, March 17, 2015 at 5pm

All homework must be submitted electronically via Moodle as separate PDF files, one for each numbered problem. Please see the course web site for more information.

1. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.



You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are Or gates, the nodes at odd levels (where it's Death's turn) are And gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for True and False. If the output at the top of the tree is True, then you can win and live forever! If the output at the top of the tree is False, you should challenge Death to a game of Twister instead.

(a) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy, but be specific!]*

*(b) ***[Extra credit]*** Prove that *any* deterministic algorithm that correctly determines whether you can win *must* examine every leaf in the tree. It follows that any correct algorithm for part (a) *must* take $\Omega(4^n)$ time. *[Hint: Let Death cheat, but not in a way that the algorithm can detect.]*

(c) Unfortunately, Death won't give you enough time to look at every node in the tree. Describe a *randomized* algorithm that determines whether you can win in $O(3^n)$ expected time. *[Hint: Consider the case $n = 1$.]*

*(d) ***[Extra credit]*** Describe and analyze a randomized algorithm that determines whether you can win in $O(c^n)$ expected time, for some explicit constant $c < 3$. Your analysis should yield an exact value for the constant c. *[Hint: You may not need to change your algorithm from part (b) at all!]*

2. A ***meldable priority queue*** stores a set of priorities from some totally-ordered universe (such as the integers) and supports the following operations:

   - MakeQueue: Return a new priority queue containing the empty set.
   - FindMin(Q): Return the smallest element of Q (if any).
   - DeleteMin(Q): Remove the smallest element in Q (if any).
   - Insert(Q, x): Insert element x into Q, if it is not already there.
   - DecreasePriority(Q, x, y): Replace an element $x \in Q$ with a new element $y < x$. (If $y \geq x$, the operation fails.) The input includes a pointer directly to the node in Q containing x.
   - Delete(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x.
   - Meld(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2. The elements of Q_1 and Q_2 could be arbitrarily intermixed; we do *not* assume, for example, that every element of Q_1 is smaller than every element of Q_2.

   A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a priority, along with pointers to its parent and two children. Meld can be implemented using the following randomized algorithm. The input consists of pointers to the roots of the two trees.

```
MELD(Q_1, Q_2):
  if $Q_1 = $ NULL then return $Q_2$
  if $Q_2 = $ NULL then return $Q_1$
  if $priority(Q_1) > priority(Q_2)$
      swap $Q_1 \leftrightarrow Q_2$
  with probability 1/2
      $left(Q_1) \leftarrow$ MELD($left(Q_1), Q_2$)
  else
      $right(Q_1) \leftarrow$ MELD($right(Q_1), Q_2$)
  return $Q_1$
```

   (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of Meld(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *[Hint: What is the expected length of a random root-to-leaf path in an n-node binary tree, where each left/right choice is made with equal probability?]*

   (b) Prove that Meld(Q_1, Q_2) runs in $O(\log n)$ time with high probability. *[Hint: You can use Chernoff bounds, but the simpler identity $\binom{c}{k} \leq (ce)^k$ is actually sufficient.]*

   (c) Show that each of the other meldable priority queue operations can be implemented with at most one call to Meld and $O(1)$ additional time. (It follows that every operation takes $O(\log n)$ time with high probability.)

New CS 473: Algorithms, Spring 2015
Homework 7

Due Tuesday, March 31, 2015 at 5pm

All homework must be submitted electronically via Moodle as separate PDF files, one for each numbered problem. Please see the course web site for more information.

1. ***Reservoir sampling*** is a method for choosing an item uniformly at random from an arbitrarily long stream of data; for example, the sequence of packets that pass through a router, or the sequence of IP addresses that access a given web page. Like all data stream algorithms, this algorithm must process each item in the stream quickly, using very little memory.

```
GetOneSample(stream $S$):
    $\ell \leftarrow 0$
    while $S$ is not done
        $x \leftarrow$ next item in $S$
        $\ell \leftarrow \ell + 1$
        if Random($\ell$) = 1
            sample ← x          (⋆)
    return sample
```

At the end of the algorithm, the variable ℓ stores the length of the input stream S; this number is *not* known to the algorithm in advance. If S is empty, the output of the algorithm is (correctly!) undefined.

Consider an arbitrary non-empty input stream S, and let n denote the (unknown) length of S.

(a) Prove that the item returned by GetOneSample(S) is chosen uniformly at random from S.

(b) What is the *exact* expected number of times that GetOneSample(S) executes line (⋆)?

(c) What is the *exact* expected value of ℓ when GetOneSample(S) executes line (⋆) for the *last* time?

(d) What is the *exact* expected value of ℓ when either GetOneSample(S) executes line (⋆) for the *second* time (or the algorithm ends, whichever happens first)?

(e) Describe and analyze an algorithm that returns a subset of k distinct items chosen uniformly at random from a data stream of length at least k. The integer k is given as part of the input to your algorithm. Prove that your algorithm is correct.

For example, if $k = 2$ and the stream contains the sequence ⟨♠, ♥, ♦, ♣⟩, the algorithm would return the subset {♦, ♠} with probability $1/6$.

New CS 473: Algorithms, Spring 2015
Homework 8

Due Tuesday, April 6, 2015 at 5pm

All homework must be submitted electronically via Moodle as separate PDF files, one for each numbered problem. Please see the course web site for more information.

1. Suppose you are given a directed graph $G = (V, E)$, two vertices s and t, a capacity function $c \colon E \to \mathbb{R}^+$, and a second function $f : E \to \mathbb{R}$.

   (a) Describe and analyze an efficient algorithm to determine whether f is a maximum (s, t)-flow in G.

   (b) Describe and analyze an efficient algorithm to determine whether f is the *unique* maximum (s, t)-flow in G.

   Do not assume anything about the function f.

2. A new assistant professor, teaching maximum flows for the first time, suggested the following greedy modification to the generic Ford-Fulkerson augmenting path algorithm. Instead of maintaining a residual graph, the greedy algorithm just reduces the capacity of edges along the augmenting path. In particular, whenever the algorithm saturates an edge, that edge is simply removed from the graph.

```
GreedyFlow($G, c, s, t$):
  for every edge $e$ in $G$
      $f(e) \leftarrow 0$

  while there is a path from $s$ to $t$ in $G$
      $\pi \leftarrow$ arbitrary path from $s$ to $t$ in $G$
      $F \leftarrow$ minimum capacity of any edge in $\pi$
      for every edge $e$ in $\pi$
          $f(e) \leftarrow f(e) + F$
          if $c(e) = F$
              remove $e$ from $G$
          else
              $c(e) \leftarrow c(e) - F$

  return $f$
```

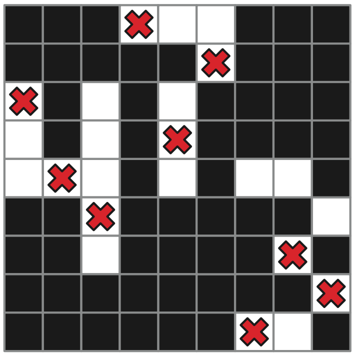
   (a) Prove that GreedyFlow does not always compute a maximum flow.

   (b) Prove that GreedyFlow is not even guaranteed to compute a good approximation to the maximum flow. That is, for any constant $\alpha > 1$, describe a flow network G such that the value of the maximum flow is more than α times the value of the flow computed by GreedyFlow. *[Hint: Assume that GreedyFlow chooses the worst possible path π at each iteration.]*

(c) Prove that for any flow network, if the Greedy Path Fairy tells you precisely which path π to use at each iteration, then GREEDYFLOW does compute a maximum flow. (Sadly, the Greedy Path Fairy does not actually exist.)

3. Suppose we are given an $n \times n$ square grid, some of whose squares are colored black and the rest white. Describe and analyze an algorithm to determine whether tokens can be placed on the grid so that

- every token is on a white square;
- every row of the grid contains exactly one token; and
- every column of the grid contains exactly one token.



Your input is a two dimensional array *IsWhite*$[1 \mathinner{..} n, 1 \mathinner{..} n]$ of booleans, indicating which squares are white. Your output is a single boolean. For example, given the grid above as input, your algorithm should return TRUE.

New CS 473: Algorithms, Spring 2015
Homework 9

Due Tuesday, April 21, 2015 at 5pm

All homework must be submitted electronically via Moodle as separate PDF files, one for each numbered problem. Please see the course web site for more information.

---

1. Suppose we are given an array $A[1..m][1..n]$ of real numbers. We want to *round* A to an integer array, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longmapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

   Describe and analyze an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

2. Suppose we are given a set of boxes, each specified by their height, width, and depth in centimeters. All three side lengths of every box lie strictly between 10cm and 20cm. As you should expect, one box can be placed inside another if the smaller box can be rotated so that its height, width, and depth are respectively smaller than the height, width, and depth of the larger box. Boxes can be nested recursively. Call a box *visible* if it is not inside another box.

   (a) Describe and analyse an algorithm to find the largest subset of the given boxes that can be nested so that only one box is visible.

   (b) Describe and analyze an algorithm to nest *all* the given boxes so that the number of visible boxes is as small as possible. *[Hint: Do **not** use part (a).]*

3. Describe and analyze an algorithm for the following problem, first posed and solved by the German mathematician Carl Jacobi in the early 1800s.[□]

> *Disponantur nn quantitates $h_k^{(i)}$ quaecunque in schema Quadrati, ita ut k habeantur n series horizontales et n series verticales, quarum quaeque est n terminorum. Ex illis quantitatibus eligantur n transversales, i.e. in seriebus horizontalibus simul atque verticalibus diversis positae, quod fieri potest $1.2\ldots n$ modis; ex omnibus illis modis quaerendum est is, qui summam n numerorum electorum suppeditet maximam.*

[□] Carl Gustav Jacob Jacobi. De investigando ordine systematis aequationum differentialum vulgarium cujuscunque. *J. Reine Angew. Math.* 64(4):297–320, 1865. Posthumously published by Carl Borchardt.

For the few students who are not fluent in mid-19th century academic Latin, here is a modern English translation of Jacobi's problem. Suppose we are given an $n \times n$ matrix M. Describe and analyze an algorithm that computes a permutation σ that maximizes the sum $\sum_i M_{i,\sigma(i)}$, or equivalently, permutes the columns of M so that the sum of the elements along the diagonal is as large as possible.

Please do not submit your solution in mid-19th century academic Latin.

꩜ New CS 473: Algorithms, Spring 2015 ꩜
Homework 10

Due Tuesday, April 28, 2015 at 5pm

This is the last graded homework.

---

1. Given points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ in the plane, the ***linear regression problem*** asks for real numbers a and b such that the line $y = ax + b$ fits the points as closely as possible, according to some criterion. The most common fit criterion is the ***L_2 error***, defined as follows:

$$\varepsilon_2(a, b) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

(This is the error metric *(ordinary/linear) least squares*.)

But there are several other ways of measuring how well a line fits a set of points, some of which can be optimized via linear programming.

(a) The ***L_1 error*** (or *total absolute deviation*) of the line $y = ax + b$ is the sum of the vertical distances from the given points to the line:

$$\varepsilon_1(a, b) = \sum_{i=1}^{n} \left| y_i - ax_i - b \right|.$$

Describe a linear program whose solution (a, b) describes the line with minimum L_1 error.

(b) The ***L_∞ error*** (or *maximum absolute deviation*) of the line $y = ax + b$ is the maximum vertical distance from any given point to the line::

$$\varepsilon_\infty(a, b) = \max_{i=1}^{n} \left| y_i - ax_i - b \right|.$$

Describe a linear program whose solution (a, b) describes the line with minimum L_∞ error.

2. (a) Give a linear-programming formulation of the maximum-cardinality bipartite matching problem. The input is a bipartite graph $G = (L \cup R, E)$, where every edge connects a vertex in L (“on the left”) with a vertex in R (“on the right”). The output is the largest matching in G. Your linear program should have one variable for each edge.

(b) Now dualize the linear program from part (a). What do the dual variables represent? What does the objective function represent? What problem is this!?

3. An ***integer program*** is a linear program with the additional constraint that the variables must take only integer values. Prove that deciding whether a given integer program has a feasible solution is NP-hard. *[Hint: Any NP-complete decision problem can be formulated as an integer program. Choose your favorite!]*

New CS 473: Algorithms, Spring 2015
Homework 11

This homework will not be graded.

1. In Homework 10, we considered several different problems that can be solved by reducing them to a linear programming problem:

   - Finding a line that fits a given set of n points in the plane with minimum L_1 error.
   - Finding a line that fits a given set of n points in the plane with minimum L_∞ error.
   - Finding the largest matching in a bipartite graph.
   - Finding the smallest vertex cover in a bipartite graph.

   The specific linear programs are described in the homework solutions. For each of these linear programs, answer the following questions in the language of the original problem:

   (a) What is a basis?

   (b) (For the line-fitting problems only:) How many different bases are there?

   (c) What is a *feasible* basis?

   (d) What is a *locally optimal* basis?

   (e) What is a pivot?

2. Let $G = (V, E)$ be an arbitrary directed graph with weighted vertices; vertex weights may be positive, negative, or zero. A ***prefix*** of G is a subset $P \subseteq V$, such that there is no edge $u \to v$ where $u \notin P$ but $v \in P$. A ***suffix*** of G is the complement of a prefix. Finally, an ***interval*** of G is the intersection of a prefix of G and a suffix of G. The weight of a prefix, suffix, or interval is the sum of the weights of its vertices.

   (a) Describe a linear program that characterizes the maximum-weight prefix of G. Your linear program should have one variable per vertex, indicating whether that vertex is or is not in the chosen prefix.

   (b) Describe a linear program that characterizes the maximum-weight interval of G.

   [Hint: Don't worry about the solutions to your linear programs being integral; they will be. If all vertex weights are negative, the maximum-weight interval is empty; if all vertex weights are positive, the maximum-weight interval contains every vertex.]

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. Recall that a boolean formula is in *conjunctive normal form* if it is the conjunction (AND) of a series of *clauses*, each of which is a disjunction (OR) of a series of literals, each of which is either a variable or the negation of a variable. Consider the following variants of SAT:

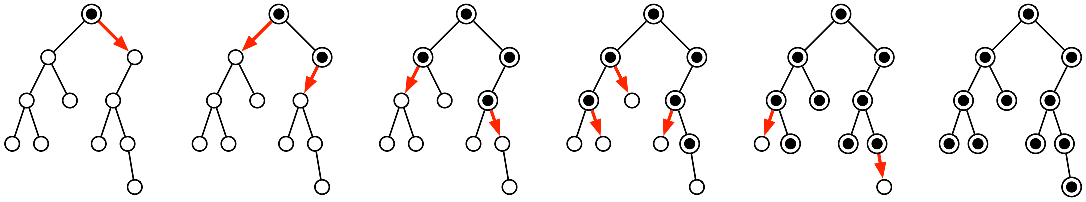
   - ***3SAT***: Given a boolean formula Φ in conjunctive normal form, such that every clause in Φ contains exactly *three* literals, is Φ satisfiable?
   - ***4SAT***: Given a boolean formula Φ in conjunctive normal form, such that every clause in Φ contains exactly *four* literals, is Φ satisfiable?

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

   (b) Describe a polynomial-time reduction from 4SAT to 3SAT.

   Don't forget to ***prove*** that your reductions are correct!

2. Suppose we need to distribute a message to all the nodes in a given binary tree. Initially, only the root node knows the message. In a single round, each node that knows the message is allowed (but not required) forward it to at most one of its children. Describe and analyze an algorithm to compute the minimum number of rounds required for the message to be delivered to all nodes in the tree.



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

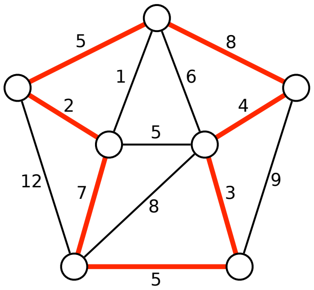
3. The ***maximum acyclic subgraph*** problem is defined as follows: The input is a directed graph $G = (V, E)$ with n vertices. Our task is to label the vertices from 1 to n so that the number of edges $u \rightarrow v$ with $label(u) < label(v)$ is as large as possible. Solving this problem exactly is NP-hard.

   (a) Describe and analyze an efficient 2-approximation algorithm for this problem.

   (b) ***Prove*** that the approximation ratio of your algorithm is at most 2.

   [Hint: Find an ordering of the vertices such that at least half of the edges point forward. Why is that enough?]

4. 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 at least 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.

5. Lenny Rutenbar, founding dean of the new Maximilian Q. Levchin College of Computer Science, has commissioned a series of snow ramps on the south slope of the Orchard Downs sledding hill[□] and challenged William (Bill) Sanders, head of the Department of Electrical and Computer Engineering, to a sledding contest. Bill and Lenny will both sled down the hill, each trying to maximize their air time. The winner gets to expand their department/college into both Siebel Center and the new ECE Building; the loser has to move their entire department/college under the Boneyard bridge next to Everitt Lab.

   Whenever Lenny or Bill 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.

   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 a sledder who takes the ith ramp will travel through the air. Describe and analyze an algorithm to determine the maximum total distance that Lenny or Bill can spend in the air.

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

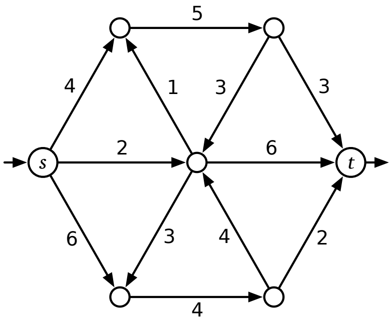
Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. Clearly indicate the following structures in the directed graph on the right. Some of these subproblems may have more than one correct answer.

   (a) A maximum (s,t)-flow f.

   (b) The residual graph of f.

   (c) A minimum (s,t)-cut.



2. Recall that a family $\mathcal{H}$ of hash functions is ***universal*** if $\Pr_{h\in\mathcal{H}}[h(x)=h(y)] \le 1/m$ for all distinct items $x \ne y$, where m is the size of the hash table. For any fixed hash function h, a ***collision*** is an unordered pair of distinct items $x \ne y$ such that $h(x)=h(y)$.

   Suppose we hash a set of n items into a table of size $m = 2n$, using a hash function h chosen uniformly at random from some universal family. Assume $\sqrt{n}$ is an integer.

   (a) ***Prove*** that the expected number of collisions is at most $n/4$.

   (b) ***Prove*** that the probability that there are at least $n/2$ collisions is at most $1/2$.

   (c) ***Prove*** that the probability that any subset of more than $\sqrt{n}$ items all hash to the same address is at most $1/2$. *[Hint: Use part (b).]*

   (d) ***[The actual exam question assumed only pairwise independence of hash values; under this weaker assumption, the claimed result is actually false. Everybody got extra credit for this part.]***

   Now suppose we choose h at random from a ***strongly 4-universal*** family of hash functions, which means for all distinct items w, x, y, z and all addresses i, j, k, l, we have

   $$\Pr_{h\in\mathcal{H}}\bigl[h(w)=i \,\wedge\, h(x)=j \,\wedge\, h(y)=k \,\wedge\, h(z)=\ell\bigr] = \frac{1}{m^4}.$$

   Prove that the probability that any subset of more than $\sqrt{n}$ items all hash to the same address is at most $O(1/n)$.

   [Hint: Use Markov's and Chebyshev's inequalities. All four statements have short elementary proofs.]

3. Suppose we have already computed a maximum flow f^* in a flow network G with *integer* capacities. Assume all flow values $f^*(e)$ are integers.

    (a) Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is *increased* by 1.

    (b) Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is *decreased* by 1.

    Your algorithms should be significantly faster than recomputing the maximum flow from scratch.

4. Let T be a treap with n vertices.

    (a) What is the *exact* expected number of leaves in T?

    (b) What is the *exact* expected number of nodes in T that have two children?

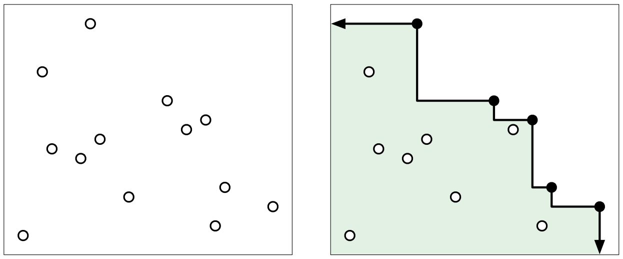
    (c) What is the *exact* expected number of nodes in T that have exactly one child?

    You do *not* need to prove that your answers are correct. *[Hint: What is the probability that the node with the kth smallest search key has no children, one child, or two children?]*

5. There is no problem 5.

Write your answers in the separate answer booklet.
You may take this question sheet with you when you leave.

1. Let S be an arbitrary set of n points in the plane. A point p in S is ***Pareto-optimal*** if no other point in S is both above and to the right. The ***staircase*** of S is the set of all points in the plane (not just in S) that have at least one point in S both above and to the right. All Pareto-optimal points lie on the boundary of the staircase.



A set of points in the plane and its staircase (shaded), with Pareto-optimal points in black.

   (a) Describe and analyze an algorithm that computes the staircase of S in $O(n \log n)$ time.

   (b) Suppose each point in S is chosen independently and uniformly at random from the unit square $[0,1] \times [0,1]$. What is the ***exact*** expected number of Pareto-optimal points in S?

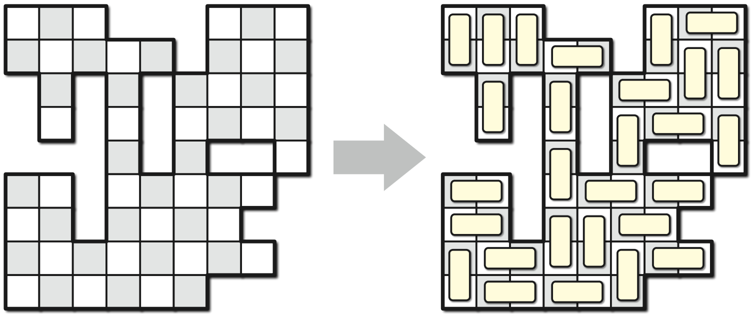
2. Let $G = (V, E)$ be a directed graph, in which every edge has capacity equal to 1 and some arbitrary cost. Edge costs could be positive, negative, or zero. Suppose you have just finished computing the minimum-cost circulation in this graph. Unfortunately, after all that work, now you realize that you recorded the direction of one of the edges incorrectly!

   Describe and analyze an algorithm to update the minimum-cost circulation in G when the direction of an arbitrary edge in G is reversed. The input to your algorithm consists of the directed graph G, the costs of edges in G, the minimum-cost circulation in G, and the edge to be reversed. Your algorithm should be faster than recomputing the minimum-cost circulation from scratch.

3. The ***chromatic number*** $\chi(G)$ of an undirected graph G is the minimum number of colors required to color the vertices, so that every edge has endpoints with different colors. Computing the chromatic number exactly is NP-hard, because 3COLOR is NP-hard.

   Prove that the following problem is also NP-hard: Given an arbitrary undirected graph G, return any integer between $\chi(G)$ and $\chi(G) + 473$.

4. Suppose you are given an $n \times n$ checkerboard with some of the squares deleted. You have a large set of dominos, just the right size to cover two squares of the checkerboard. Describe and analyze an algorithm to determine whether it is possible to tile the remaining squares with dominos—each domino must cover exactly two undeleted squares, and each undeleted square must be covered by exactly one domino.

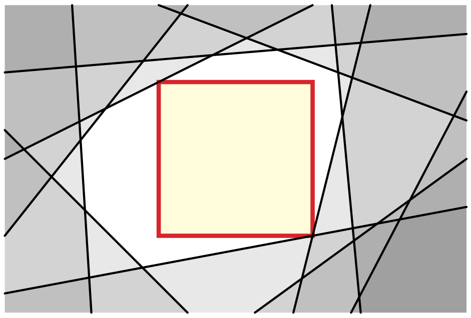


Your input is a two-dimensional array $\mathit{Deleted}[1\,..\,n, 1\,..\,n]$ of bits, where $\mathit{Deleted}[i, j] = \text{TRUE}$ if and only if the square in row i and column j has been deleted. Your output is a single bit; you do ***not*** have to compute the actual placement of dominos. For example, for the board shown above, your algorithm should return TRUE.

5. Recall from Homework 11 that a ***prefix*** of a directed graph $G = (V, E)$ is a subset $P \subseteq V$ of the vertices such that no edge $u \rightarrow v \in E$ has $u \notin P$ and $v \in P$.

   Suppose you are given a rooted tree T, with all edges directed away from the root; every vertex in T has a weight, which could be positive, negative, or zero. Describe and analyze a *self-contained* algorithm to compute the prefix of T with maximum total weight. *[Hint: Don't use linear programming.]*

6. Suppose we are given a sequence of n linear inequalities of the form $a_i x + b_i y \le c_i$; the set of all points (x, y) that satisfy these inequalities is a convex polygon P in the (x, y)-plane. Describe a linear program whose solution describes the largest square with horizontal and vertical sides that lies inside P. (You can assume that P is non-empty.)



CS 473 ✧ Spring 2016

Homework 0

Due Tuesday, January 26, 2016 at 5pm

- **This homework tests your familiarity with prerequisite material:** designing, describing, and analyzing elementary algorithms (at the level of CS 225); fundamental graph problems and algorithms (again, at the level of CS 225); and especially facility with recursion and induction. Notes on most of this prerequisite material are available on the course web page.
- **Each student must submit individual solutions for this homework.** For all future homeworks, groups of up to three students will be allowed to submit joint solutions.
- **Submit your solutions electronically on the course Moodle site as PDF files.**
  - Submit a separate file for each numbered problem.
  - You can find a LaTeX solution template on the course web site; please use it if you plan to typeset your homework.
  - If you must submit scanned handwritten solutions, use a black pen (not pencil) on blank white printer paper (not notebook or graph paper), use a high-quality scanner (not a phone camera), and print the resulting PDF file on a black-and-white printer to verify readability before you submit.

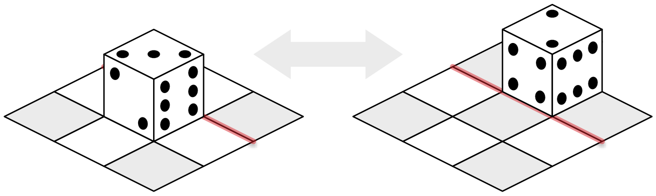
☞ Some important course policies ☜

- **You may use any source at your disposal**—paper, electronic, or human—but you ***must*** cite ***every*** source that you use, and you ***must*** write everything yourself in your own words. See the academic integrity policies on the course web site for more details.
- The answer ***"I don't know"*** (and *nothing* else) is worth 25% partial credit on any problem or subproblem, on any homework or exam, except for extra-credit problems. We will accept synonyms like "No idea" or "WTF" or "¯\(•_•)/¯", but you must write *something*.
- **Avoid the Three Deadly Sins!** There are a few dangerous writing (and thinking) habits that will trigger an ***automatic zero*** on any homework or exam problem, unless your solution is nearly perfect otherwise. Yes, we are completely serious.
  - Always give complete solutions, not just examples.
  - Always declare all your variables, in English.
  - Never use weak induction.

See the course web site for more information.

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

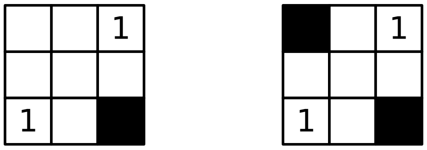
1. A ***rolling die maze*** is a puzzle involving a standard six-sided die (a cube with numbers on each side) and a grid of squares. You should imagine the grid lying on top of a table; the die always rests on and exactly covers one square. In a single step, you can *roll* the die 90 degrees around one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



Rolling a die.

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

For example, here are two rolling die mazes. Black squares are blocked; empty white squares are free. The maze on the left can be solved by placing the die on the lower left square with 1 pip on the top face, and then rolling it north, then north, then east, then east. The maze on the right is not solvable.



Two rolling die mazes. Only the maze on the left is solvable.

Describe and analyze an efficient algorithm to determine whether a given rolling die maze is solvable. Your input is a two-dimensional array $\mathit{Label}[1\,..\,n, 1\,..\,n]$, where each entry $\mathit{Label}[i, j]$ stores the label of the square in the ith row and jth column, where the label 0 means the square is free, and the label -1 means the square is blocked.

[Hint: Build a graph. What are the vertices? What are the edges? Is the graph directed or undirected? Do the vertices or edges have weights? If so, what are they? What textbook problem do you need to solve on this graph? What textbook algorithm should you use to solve that problem? What is the running time of that algorithm as a function of n? What does the number 24 have to do with anything?]

2. Describe and analyze fast algorithms for the following problems. The input for each problem is an unsorted array $A[1\,..\,n]$ of n arbitrary numbers, which may be positive, negative, or zero, and which are not necessarily distinct.

   (a) Are there two distinct indices $i < j$ such that $A[i] + A[j] = 0$?

   (b) Are there three distinct indices $i < j < k$ such that $A[i] + A[j] + A[k] = 0$?

   For example, if the input array is $[2, -1, 0, 4, 0, -1]$, both algorithms should return TRUE, but if the input array is $[4, -1, 2, 0]$, both algorithms should return FALSE. You do ***not*** need to prove that your algorithms are correct. *[Hint: The devil is in the details.]*

3. A ***binomial tree of order k*** is defined recursively as follows:

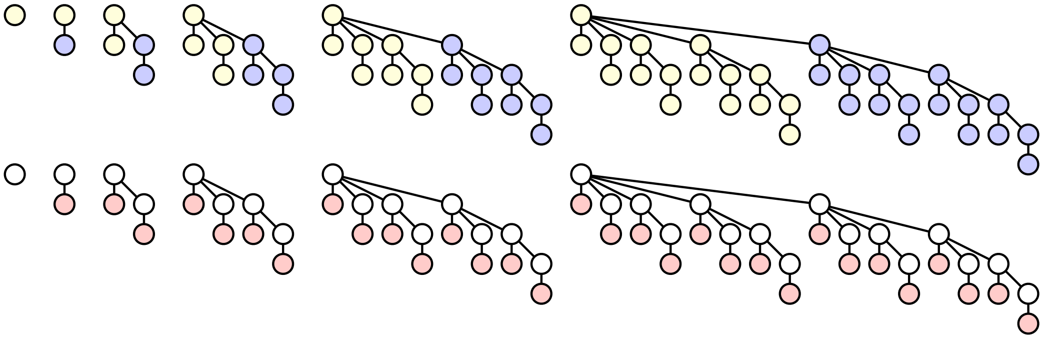
   - A binomial tree of order 0 is a single node.
   - For all $k > 0$, a binomial tree of order k consists of two binomial trees of order $k-1$, with the root of one tree connected as a new child of the root of the other. (See the figure below.)

   Prove the following claims:

   (a) For all non-negative integers k, a binomial tree of order k has exactly 2^k nodes.

   (b) For all positive integers k, attaching a new leaf to every node in a binomial tree of order $k-1$ results in a binomial tree of order k.

   (c) For all non-negative integers k and d, a binomial tree of order k has exactly $\binom{k}{d}$ nodes with depth d. (Hence the name!)

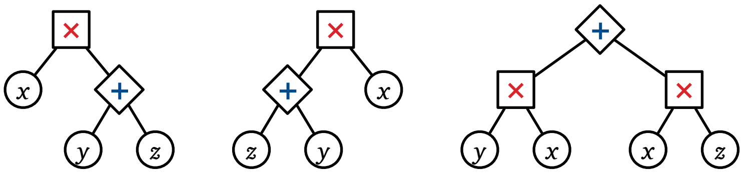


Binomial trees of order 0 through 5.
Top row: The recursive definition. Bottom row: The property claimed in part (b).

*4. *[Extra credit]* An ***arithmetic expression tree*** is a binary tree where every leaf is labeled with a variable, every internal node is labeled with an arithmetic operation, and every internal node has exactly two children. For this problem, assume that the only allowed operations are + and ×. Different leaves may or may not represent different variables.

Every arithmetic expression tree represents a function, transforming input values for the leaf variables into an output value for the root, by following two simple rules: (1) The value of any +-node is the sum of the values of its children. (2) The value of any ×-node is the product of the values of its children.

Two arithmetic expression trees are ***equivalent*** if they represent the same function; that is, the same input values for the leaf variables always leads to the same output value at both roots. An arithmetic expression tree is in ***normal form*** if the parent of every +-node (if any) is another +-node.



Three equivalent expression trees. Only the third expression tree is in normal form.

Prove that for any arithmetic expression tree, there is an equivalent arithmetic expression tree in normal form. *[Hint: This is harder than it looks.]*

CS 473 ✧ Spring 2016

Homework 1

Due Tuesday, February 2, 2016, at **8pm**

- Starting with this homework, groups of up to three students may submit joint solutions. **Group solutions must represent an honest collaborative effort by all members of the group.** Please see the academic integrity policies for more information.

- You are responsible for forming your own groups. Groups can change from homework to homework, or even from (numbered) problem to problem.

- Please make sure the names and NetIDs of *all* group members appear prominently at the top of the first page of each submission.

- Please only upload one submission per group for each problem. In the Online Text box on the problem submission page, you must type in the NetIDs of ***all*** group members, including the person submitting. See the Homework Policies for examples. Failure to enter all group NetIDs will delay (if not prevent) giving all group members the grades they deserve.

- For dynamic programming problems, a full-credit solution must include the following:
  - A clear English specification of the underlying recursive function. (For example: “Let $Edit(i, j)$ denote the edit distance between $A[1..i]$ and $B[1..j]$.”) Omitting the English description is a Deadly Sin, which will result in an automatic zero.
  - ***One*** of the following:
    - A correct recursive function or algorithm that computes the specified function, a clear description of the memoization structure, and a clear description of the iterative evaluation order.
    - Pseudocode for the final iterative dynamic programming algorithm.
  - The running time.

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

- Official solutions will provide target time bounds for full credit. Correct algorithms that are faster than the official solution will receive extra credit points; correct algorithms that are slower than the official solution will get partial credit. We rarely include these target time bounds in the actual questions, because when we do, more students submit fast but incorrect algorithms (worth 0/10 on exams) instead of correct but slow algorithms (worth 8/10 on exams).

---

1. Let's define a *summary* of two strings A and B to be a concatenation of substrings of the following form:

   - ▲`SNA` indicates a substring `SNA` of only the first string A.
   - ◆`FOO` indicates a common substring `FOO` of both strings.
   - ▼`BAR` indicates a substring `BAR` of only the second string B.

   A summary is *valid* if we can recover the original strings A and B by concatenating the appropriate substrings of the summary in order and discarding the delimiters ▲, ◆, and ▼. Each regular character has length 1, and each delimiter ▲, ◆, or ▼ has some fixed non-negative length Δ. The *length* of a summary is the sum of the lengths of its symbols.

   For example, each of the following strings is a valid summary of the strings `KITTEN` and `KNITTING`:

   - ◆`K`▼`N`◆`ITT`▲`E`▼`I`◆`N`▼`G` has length $9+7\Delta$.
   - ◆`K`▼`N`◆`ITT`▲`EN`▼`ING` has length $10+5\Delta$.
   - ◆`K`▲`ITTEN`▼`NITTING` has length $13+3\Delta$.
   - ▲`KITTEN`▼`KNITTING` has length $14+2\Delta$.

   Describe and analyze an algorithm that computes the length of the shortest summary of two given strings $A[1\,..\,m]$ and $B[1\,..\,n]$. The delimiter length Δ is also part of the input to your algorithm. For example:

   - Given strings `KITTEN` and `KNITTING` and $\Delta = 0$, your algorithm should return 9.
   - Given strings `KITTEN` and `KNITTING` and $\Delta = 1$, your algorithm should return 15.
   - Given strings `KITTEN` and `KNITTING` and $\Delta = 2$, your algorithm should return 18.

2. Suppose you are given a sequence of positive integers separated by plus (+) and minus (−) 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:

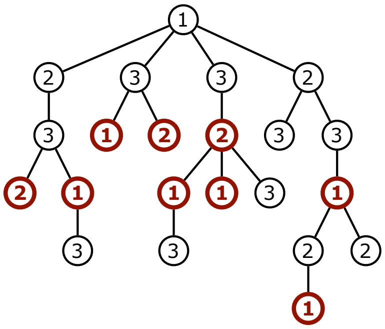
   $$1+3-2-5+1-6+7=-1$$
   $$(1+3-(2-5))+(1-6)+7=9$$
   $$(1+(3-2))-(5+1)-(6+7)=-17$$

   Describe and analyze an algorithm to compute the maximum possible value the expression can take by adding parentheses.

   You may only use parentheses to group additions and subtractions; in particular, you are not allowed to create implicit multiplication as in $1+3(-2)(-5)+1-6+7=33$.

3. The president of the Punxsutawney office of Giggle, Inc. has decided to give every employee a present to celebrate Groundhog Day! Each employee must receive one of three gifts: (1) an all-expenses-paid six-week vacation with Bill Murray,[□] (2) an all-the-Punxsutawney-pancakes-you-can-eat breakfast for two at Punxy Phil's Family Restaurant, or (3) a burning paper bag of groundhog poop. Corporate regulations prohibit any employee from receiving exactly the same gift as their direct supervisor. Unfortunately, any employee who receives a better gift than their direct supervisor will almost certainly be fired in a fit of jealousy.

   As Giggle-Punxsutawney's official gift czar, it's *your* job to decide which gift each employee receives. Describe an algorithm to distribute gifts so that the minimum number of people are fired. Yes, you can give the president groundhog poop.



A tree labeling with cost 9. The nine bold nodes have smaller labels than their parents.
The president got a vacation with Bill Murray. This is *not* the optimal labeling for this tree.

   More formally, you are given a rooted tree T, representing the company hierarchy, and you want to label each node in T with an integer 1, 2, or 3, so that every node has a different label from its parent. The *cost* of an labeling is the number of nodes that have smaller labels than their parents. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree T.

[□] The details of scheduling n distinct six-week vacations with Bill Murray, all in a single year, are left as an exercise for the reader.

CS 473 ✧ Spring 2016

Homework 2

Due Tuesday, February 9, 2016, at 8pm

---

1. *[Insert amusing story about distributing polling stations or cell towers or Starbucks or something on a long straight road in rural Iowa. Ha ha ha, how droll.]*

   More formally, you are given a sorted array $X[1\,..\,n]$ of distinct numbers and a positive integer k. A set of k intervals ***covers*** X if every element of X lies inside one of the k intervals. Your aim is to find k intervals $[a_1, z_1], [a_2, z_2], \ldots, [a_k, z_k]$ that cover X where the function $\sum_{i=1}^{k}(z_i - a_i)^2$ is as small as possible. Intuitively, you are trying to cover the points with k intervals whose lengths are as close to equal as possible.

   (a) Describe an algorithm that finds k intervals with minimum total squared length that cover X. The running time of your algorithm should be a simple function of n and k.

   (b) Consider the two-dimensional matrix $M[1\,..\,n, 1\,..\,n]$ defined as follows:

   $$M[i,j] = \begin{cases} (X[j] - X[i])^2 & \text{if } i \le j \\ \infty & \text{otherwise} \end{cases}$$

   Prove that M satisfies the ***Monge property***: $M[i,j] + M[i',j'] \le M[i,j'] + M[i',j]$ for all indices $i < i'$ and $j < j'$.

   (c) ***[Extra credit]*** Describe an algorithm that finds k intervals with minimum total squared length that cover X ***in $O(nk)$ time***. *[Hint: Solve part (a) first, then use part (b).]*

   We strongly recommend submitting your solution to part (a) separately, and only describing your changes to that solution for part (c).

2. The Doctor and River Song decide to play a game on a directed acyclic graph G, which has one source s and one sink t.□

   Each player has a token on one of the vertices of G. At the start of the game, The Doctor's token is on the source vertex s, and River's token is on the sink vertex t. The players alternate turns, with The Doctor moving first. On each of his turns, the Doctor moves his token forward along a directed edge; on each of her turns, River moves her token *backward* along a directed edge.

   If the two tokens ever meet on the same vertex, River wins the game. ("Hello, Sweetie!") If the Doctor's token reaches t or River's token reaches s before the two tokens meet, then the Doctor wins the game.

   Describe and analyze an algorithm to determine who wins this game, assuming both players play perfectly. That is, if the Doctor can win *no matter how River moves*, then your algorithm should output "Doctor", and if River can win *no matter how the Doctor moves*, your algorithm should output "River". (Why are these the only two possibilities?) The input to your algorithm is the graph G.

□possibly short for the Untempered **S**chism and the **T**ime Vortex, or the Shining World of the Seven Systems (otherwise known as Gallifrey) and Trenzalore, or Skaro and Telos, or something timey-wimey.

CS 473 ✧ Spring 2016

Homework 3

Due Tuesday, February 9, 2016, at 8pm

Unless a problem specifically states otherwise, you may assume a function Random that takes a positive integer k as input and returns an integer chosen uniformly and independently at random from $\{1, 2, \ldots, k\}$ in $O(1)$ time. For example, to flip a fair coin, you could call Random(2).

---

1. Suppose we want to write an efficient function RandomPermutation(n) that returns a permutation of the set $\{1, 2, \ldots, n\}$ chosen uniformly at random.

   (a) Prove that the following algorithm is ***not*** correct. *[Hint: There is a one-line proof!]*

```
RandomPermutation($n$):
  for $i \leftarrow 1$ to $n$
      $\pi[i] \leftarrow i$
  for $i \leftarrow 1$ to $n$
      swap $\pi[i] \leftrightarrow \pi[\text{Random}(n)]$
```

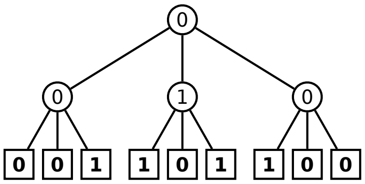
   (b) Consider the following implementation of RandomPermutation.

```
RandomPermutation($n$):
  for $i \leftarrow 1$ to $n$
      $\pi[i] \leftarrow$ NULL
  for $i \leftarrow 1$ to $n$
      $j \leftarrow \text{Random}(n)$
      while ($\pi[j]$ != NULL)
          $j \leftarrow \text{Random}(n)$
      $\pi[j] \leftarrow i$
  return $\pi$
```

   Prove that this algorithm is correct and analyze its expected running time.

   (c) Describe and analyze an implementation of RandomPermutation that runs in expected worst-case time $O(n)$.

2. A ***majority tree*** is a complete ternary tree in which every leaf is labeled either 0 or 1. The *value* of a leaf is its label; the *value* of any internal node is the majority of the values of its three children. For example, if the tree has depth 2 and its leaves are labeled $1, 0, 0, 0, 1, 0, 1, 1, 1$, the root has value 0.



A majority tree with depth 2.

It is easy to compute value of the root of a majority tree of depth n in $O(3^n)$ time, given the sequence of 3^n leaf labels as input, using a simple post-order traversal of the tree. Prove that this simple algorithm is optimal, and then describe a better algorithm. More formally:

(a) Prove that *any* deterministic algorithm that computes the value of the root of a majority tree *must* examine every leaf. *[Hint: Consider the special case $n = 1$. Recurse.]*

(b) Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some explicit constant $c < 3$. *[Hint: Consider the special case $n = 1$. Recurse.]*

3. A ***meldable priority queue*** stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

- MakeQueue: Return a new priority queue containing the empty set.
- FindMin(Q): Return the smallest element of Q (if any).
- DeleteMin(Q): Remove the smallest element in Q (if any).
- Insert(Q, x): Insert element x into Q, if it is not already there.
- DecreaseKey(Q, x, y): Replace an element $x \in Q$ with a smaller key y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x.
- Delete(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x.
- Meld(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2.

A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. Meld can be implemented using the following randomized algorithm:

```
Meld($Q_1, Q_2$):
  if $Q_1$ is empty return $Q_2$
  if $Q_2$ is empty return $Q_1$
  if $key(Q_1) > key(Q_2)$
      swap $Q_1 \leftrightarrow Q_2$
  with probability 1/2
      $left(Q_1) \leftarrow$ Meld($left(Q_1), Q_2$)
  else
      $right(Q_1) \leftarrow$ Meld($right(Q_1), Q_2$)
  return $Q_1$
```

(a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of Meld(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *[Hint: What is the expected length of a random root-to-leaf path in an n-node binary tree, where each left/right choice is made with equal probability?]*

(b) Prove that Meld(Q_1, Q_2) runs in $O(\log n)$ time with high probability.

(c) Show that each of the other meldable priority queue operations can be implemented with at most one call to Meld and $O(1)$ additional time. (It follows that each operation takes only $O(\log n)$ time with high probability.)

CS 473 ✧ Spring 2016

Homework 4

Due Tuesday, March 1, 2016, at 8pm

Unless a problem specifically states otherwise, you may assume a function Random that takes a positive integer k as input and returns an integer chosen uniformly and independently at random from $\{1, 2, \ldots, k\}$ in $O(1)$ time. For example, to flip a fair coin, you could call Random(2).

---

1. Suppose we are given a two-dimensional array $M[1..n, 1..n]$ in which every row and every column is sorted in increasing order and no two elements are equal.

   (a) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, compute the number of elements of M larger than $M[i, j]$ and smaller than $M[i', j']$.

   (b) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, return an element of M chosen uniformly at random from the elements larger than $M[i, j]$ and smaller than $M[i', j']$. Assume the requested range is always non-empty.

   (c) Describe and analyze a randomized algorithm to compute the median element of M in $O(n \log n)$ expected time.

2. ***Tabulated hashing*** uses tables of random numbers to compute hash values. Suppose $|\mathcal{U}| = 2^w \times 2^w$ and $m = 2^\ell$, so the items being hashed are pairs of w-bit strings (or $2w$-bit strings broken in half) and hash values are ℓ-bit strings.

   Let $A[0..2^w - 1]$ and $B[0..2^w - 1]$ be arrays of independent random ℓ-bit strings, and define the hash function $h_{A,B} : \mathcal{U} \to [m]$ by setting

   $$h_{A,B}(x, y) := A[x] \oplus B[y]$$

   where $\oplus$ denotes bit-wise exclusive-or. Let $\mathcal{H}$ denote the set of all possible functions $h_{A,B}$. Filling the arrays A and B with independent random bits is equivalent to choosing a hash function $h_{A,B} \in \mathcal{H}$ uniformly at random.

   (a) Prove that $\mathcal{H}$ is 2-uniform.

   (b) Prove that $\mathcal{H}$ is 3-uniform. *[Hint: Solve part (a) first.]*

   (c) Prove that $\mathcal{H}$ is ***not*** 4-uniform.

   Yes, "see part (b)" is worth full credit for part (a), but only if your solution to part (b) is correct.

CS 473 ✧ Spring 2016

Homework 5

Due Tuesday, March 1, 2016, at 8pm

Unless a problem specifically states otherwise, you may assume a function Random that takes a positive integer k as input and returns an integer chosen uniformly and independently at random from $\{1, 2, \ldots, k\}$ in $O(1)$ time. For example, to flip a fair coin, you could call Random(2).

---

1. ***Reservoir sampling*** is a method for choosing an item uniformly at random from an arbitrarily long stream of data.

```
GetOneSample(stream $S$):
  $\ell \leftarrow 0$
  while $S$ is not done
      $x \leftarrow$ next item in $S$
      $\ell \leftarrow \ell + 1$
      if Random($\ell$) = 1
          sample ← x        ($\star$)
  return sample
```

At the end of the algorithm, the variable ℓ stores the length of the input stream S; this number is *not* known to the algorithm in advance. If S is empty, the output of the algorithm is (correctly!) undefined. In the following, consider an arbitrary non-empty input stream S, and let n denote the (unknown) length of S.

(a) Prove that the item returned by GetOneSample(S) is chosen uniformly at random from S.

(b) Describe and analyze an algorithm that returns a subset of k distinct items chosen uniformly at random from a data stream of length at least k. The integer k is given as part of the input to your algorithm. Prove that your algorithm is correct.

For example, if $k = 2$ and the stream contains the sequence ⟨♠, ♥, ♦, ♣⟩, the algorithm should return the subset {♦, ♠} with probability $1/6$.

2. In this problem, we will derive a streaming algorithm that computes an accurate estimate $\widetilde{n}$ of the number of distinct items in a data stream S. Suppose S contains n unique items (but possible several copies of each item); the algorithm does *not* know n in advance. Given an accuracy parameter $0 < \varepsilon < 1$ and a confidence parameter $0 < \delta < 1$ as part of the input, our final algorithm will guarantee that $\Pr[|\widetilde{n} - n| > \varepsilon n] < \delta$.

   As a first step, fix a positive integer m that is large enough that we don't have to worry about round-off errors in the analysis. Our first algorithm chooses a hash function $h\colon \mathcal{U} \to [m]$ at random from a ***2-uniform*** family, computes the minimum hash value $\hbar = \min\{h(x) \mid x \in S\}$, and finally returns the estimate $\widetilde{n} = m/\hbar$.

   (a) Prove that $\Pr[\widetilde{n} > (1+\varepsilon)n] \le 1/(1+\varepsilon)$. *[Hint: Markov's inequality]*

   (b) Prove that $\Pr[\widetilde{n} < (1-\varepsilon)n] \le 1-\varepsilon$. *[Hint: Chebyshev's inequality]*

   (c) We can improve this estimator by maintaining the k smallest hash values, for some integer $k > 1$. Let $\widetilde{n}_k = k \cdot m/\hbar_k$, where $\hbar_k$ is the kth smallest element of $\{h(x) \mid x \in S\}$.

   Estimate the smallest value of k (as a function of the accuracy parameter ε) such that $\Pr[|\widetilde{n}_k - n| > \varepsilon n] \le 1/4$.

   (d) Now suppose we run d copies of the previous estimator in parallel to generate d independent estimates $\widetilde{n}_{k,1}, \widetilde{n}_{k,2}, \ldots, \widetilde{n}_{k,d}$, for some integer $d > 1$. Each copy uses its own independently chosen hash function, but they all use the same value of k that you derived in part (c). Let $\widetilde{N}$ be the *median* of these d estimates.

   Estimate the smallest value of d (as a function of the confidence parameter δ) such that $\Pr[|\widetilde{N} - n| > \varepsilon n] \le \delta$.

CS 473 ✧ Spring 2016

Homework 6

Due Tuesday, March 15, 2016, at 8pm

For problems that use maximum flows as a black box, a full-credit solution requires the following.

- A complete description of the relevant flow network, specifying the set of vertices, the set of edges (being careful about direction), the source and target vertices s and t, and the capacity of every edge. (If the flow network is part of the original input, just say that.)
- A description of the algorithm to construct this flow network from the stated input. This could be as simple as "We can construct the flow network in $O(n^3)$ time by brute force."
- A description of the algorithm to extract the answer to the stated problem from the maximum flow. This could be as simple as "Return TRUE if the maximum flow value is at least 42 and False otherwise."
- A proof that your reduction is correct. This proof will almost always have two components. For example, if your algorithm returns a boolean, you should prove that its TRUE answers are correct and that its FALSE answers are correct. If your algorithm returns a number, you should prove that number is neither too large nor too small.
- The running time of the overall algorithm, expressed as a function of the original input parameters, not just the number of vertices and edges in your flow network.
- You may assume that maximum flows can be computed in $O(VE)$ time. Do *not* regurgitate the maximum flow algorithm itself.

Reductions to other flow-based algorithms described in class or in the notes (for example: edge-disjoint paths, maximum bipartite matching, minimum-cost circulation) or to other standard graph problems (for example: reachability, minimum spanning tree, shortest paths) have similar requirements.

---

1. Suppose you are given a directed graph $G = (V, E)$, two vertices s and t in V, a capacity function $c\colon E \to \mathbb{R}^+$, and a second function $f : E \to \mathbb{R}$. Describe an algorithm to determine whether f is a maximum (s,t)-flow in G. Do not assume ***anything*** about the function f.

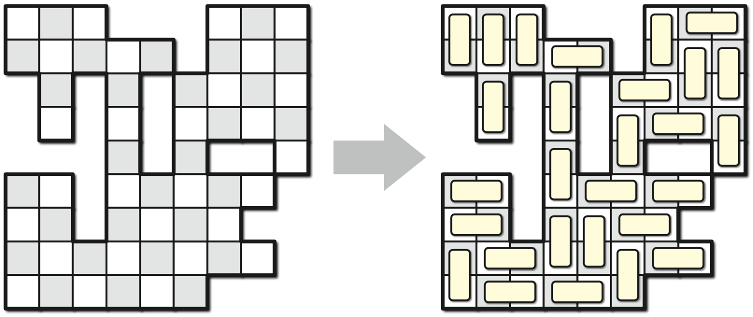
2. Suppose you have already computed a maximum flow f^* in a flow network G with ***integer*** edge capacities.

   (a) Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is increased by 1.

   (b) Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is decreased by 1.

   Both algorithms should be significantly faster than recomputing the maximum flow from scratch.

3. Suppose you are given an $n \times n$ checkerboard with some of the squares deleted. You have a large set of dominos, just the right size to cover two squares of the checkerboard. Describe and analyze an algorithm to determine whether it is possible to tile the board with dominos—each domino must cover exactly two undeleted squares, and each undeleted square must be covered by exactly one domino.



Your input is a two-dimensional array $Deleted[1..n, 1..n]$ of bits, where $Deleted[i, j] = \text{TRUE}$ if and only if the square in row i and column j has been deleted. Your output is a single bit; you do ***not*** have to compute the actual placement of dominos. For example, for the board shown above, your algorithm should return TRUE.

CS 473 ✧ Spring 2016

Homework 7

Due Tuesday, March 29, 2016, at 8pm

This is the last homework before Midterm 2.

1. Suppose we are given a two-dimensional array $A[1\,..\,m, 1\,..\,n]$ of non-negative real numbers. We would like to *round* A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longmapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

Describe and analyze an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding exists.

2. You're organizing the Third Annual UIUC Computer Science 72-Hour Dance Exchange, to be held all day Friday, Saturday, and Sunday in Siebel Center.[1] Several 30-minute sets of music will be played during the event, and a large number of DJs have applied to perform. You need to hire DJs according to the following constraints.

   - Exactly k sets of music must be played each day, and thus $3k$ sets altogether.
   - Each set must be played by a single DJ in a consistent musical genre (ambient, bubblegum, dancehall, horrorcore, trip-hop, Nashville country, Chicago blues, axé, laïkó, skiffle, shape note, Nitzhonot, J-pop, K-pop, C-pop, T-pop, 8-bit, Tesla coil, . . .).
   - Each genre must be played at most once per day.
   - Each DJ has given you a list of genres they are willing to play.
   - No DJ can play more than five sets during the entire event.

   Suppose there are n candidate DJs and g different musical genres available. Describe and analyze an efficient algorithm that either assigns a DJ and a genre to each of the $3k$ sets, or correctly reports that no such assignment is possible.

3. Describe and analyze an algorithm to determine, given an undirected[2] graph $G = (V, E)$ and three vertices $u, v, w \in V$ as input, whether G contains a simple path from u to w that passes through v.

[1] Efforts to secure overflow space in ECEB were sadly unsuccessful.

[2] This adjective is important; if the input graph were directed, this problem would be NP-hard.

CS 473 ✧ Spring 2016
Homework 8

Due Tuesday, April 12, 2016, at 8pm

You may assume the following results in your solutions:

- Maximum flows and minimum cuts can be computed in $O(VE)$ time.
- Minimum-cost flows can be computed in $O(E^2 \log^2 V)$ time.
- Linear programming problems with integer coefficients can be solved in polynomial time.

For problems that ask for a linear-programming formulation of some problem, a full credit solution requires the following components:

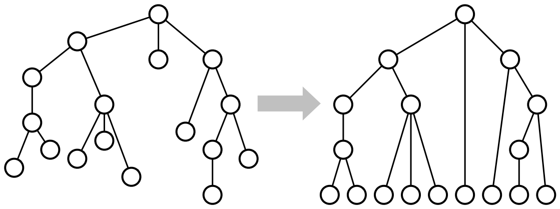
- A list of variables, along with a brief English description of each variable. (Omitting these English descriptions is a Deadly Sin.)
- A linear objective function (expressed either as minimization or maximization, whichever is more convenient), along with a brief English description of its meaning.
- A sequence of linear inequalities (expressed using $\le$, $=$, or $\ge$, whichever is more appropriate or convenient), along with a brief English description of each constraint.
- A proof that your linear programming formulation is correct, meaning that the optimal solution to the original problem can always be obtained from the optimal solution to the linear program. This may be very short.

It is ***not*** necessary to express the linear program in canonical form, or even in matrix form. Clarity is much more important than formality.

---

1. Suppose your are given a rooted tree T, where every edge e has two associated values: a non-negative *length* $\ell(e)$, and a *cost* $\$(e)$ (which could be positive, negative, or zero). Your goal is to add a non-negative *stretch* $s(e) \ge 0$ to the length of every edge e in T, subject to the following conditions:

   - Every root-to-leaf path π in T has the same total stretched length $\sum_{e \in \pi}(\ell(e) + s(e))$
   - The total *weighted* stretch $\sum_e s(e) \cdot \$(e)$ is as small as possible.



   (a) Describe an instance of this problem with no optimal solution.

   (b) Give a concise linear programming formulation of this problem. (For the instance described in part (a), your linear program will be unbounded.)

   (c) Suppose that for the given tree T and the given lengths and costs, the optimal solution to this problem is unique. Prove that in this optimal solution, we have $s(e) = 0$ for every edge on some longest root-to-leaf path in T. In other words, prove that the optimally stretched tree with the same depth as the input tree. *[Hint: What is a basis in your linear program? What is a **feasible** basis?]*

Problem 1(c) originally omitted the uniqueness assumption and asked for a proof that ***every*** optimal solution has an unstretched root-to-leaf path, but that more general claim is false. For example, if every edge has cost zero, there are optimal solutions in which every edge has positive stretch.

2. Describe and analyze an efficient algorithm for the following problem, first posed and solved by the German mathematician Carl Jacobi in the early 1800s.[□]

> *Disponantur nn quantitates $h_k^{(i)}$ quaecunque in schema Quadrati, ita ut k habeantur n series horizontales et n series verticales, quarum quaeque est n terminorum. Ex illis quantitatibus eligantur n transversales, i.e. in seriebus horizontalibus simul atque verticalibus diversis positae, quod fieri potest $1.2\ldots n$ modis; ex omnibus illis modis quaerendum est is, qui summam n numerorum electorum suppeditet maximam.*

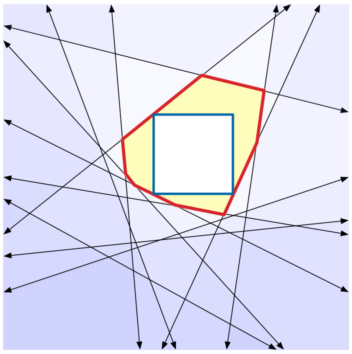
For those few students who are not fluent in mid-19th century academic Latin, here is a modern English translation of Jacobi's problem. Suppose we are given an $n \times n$ matrix M. Describe and analyze an algorithm that computes a permutation σ that maximizes the sum $\sum_{i=1}^{n} M_{i,\sigma(i)}$, or equivalently, permutes the columns of M so that the sum of the elements along the diagonal is as large as possible.

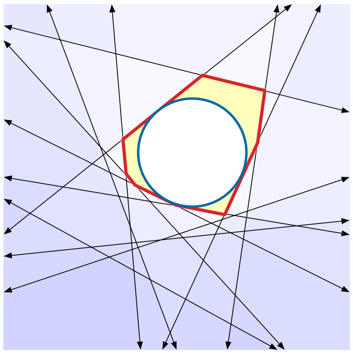
Please do not submit your solution in mid-19th century academic Latin.

3. Suppose we are given a sequence of n linear inequalities of the form $a_i x + b_i y \le c_i$. Collectively, these n inequalities describe a convex polygon P in the plane.

(a) Describe a linear program whose solution describes the largest square with horizontal and vertical sides that lies entirely inside P.

(b) Describe a linear program whose solution describes the largest circle that lies entirely inside P.





[□] Carl Gustav Jacob Jacobi. De investigando ordine systematis aequationum differentialum vulgarium cujuscunque. *J. Reine Angew. Math.* 64(4):297–320, 1865. Posthumously published by Carl Borchardt.

CS 473 ✧ Spring 2016

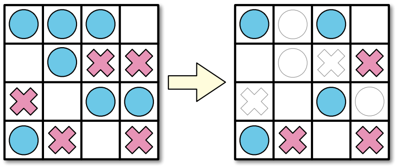
Homework 9

Due Tuesday, April 19, 2016, at 8pm

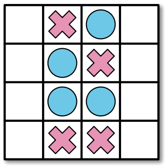
For problems that ask to prove that a given problem X is NP-hard, a full-credit solution requires the following components:

- Specify a known NP-hard problem Y, taken from the problems listed in the notes.
- Describe a polynomial-time algorithm for Y, using a black-box polynomial-time algorithm for X as a subroutine. Most NP-hardness reductions have the following form: Given an arbitrary instance of Y, describe how to transform it into an instance of X, pass this instance to a black-box algorithm for X, and finally, describe how to transform the output of the black-box subroutine to the final output. A cartoon with boxes may be helpful.
- Prove that your reduction is correct. As usual, correctness proofs for NP-hardness reductions usually have two components ("one for each f").

1. 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, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

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

2. Everyone's having a wonderful time at the party you're throwing, but now it's time to line up for ***The Algorithm March*** (アルゴリズムこうしん)! This dance was originally developed by the Japanese comedy duo Itsumo Kokokara (いつもここから) for the children's television show PythagoraSwitch (ピタゴラスイッチ). The Algorithm March is performed by a line of people; each person in line starts a specific sequence of movements one measure later than the person directly in front of them. Thus, the march is the dance equivalent of a musical round or canon, like "Row Row Row Your Boat".□ Proper etiquette dictates that each marcher must know the person directly in front of them in line, lest a minor mistake during lead to horrible embarrassment between strangers.

Suppose you are given a complete list of which people at your party know each other. Prove that it is NP-hard to determine the largest number of party-goers that can participate in the Algorithm March. You may assume without loss of generality that there are no ninjas at your party.

□そろそろおわりかな。そろそろおわりかな。そろそろおわりかな。

CS 473 ✧ Spring 2016

Homework 10

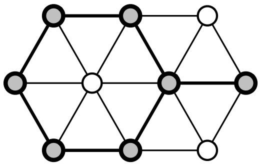
Due Tuesday, April 26, 2016, at 8pm

This is the last graded homework of the semester.

---

1. A *double-Hamiltonian circuit* in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Prove that determining whether whether a given undirected graph contains a double-Hamiltonian circuit is NP-hard.

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



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

3. Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, whether G is 3-colorable. Describe and analyze a **polynomial-time** algorithm that either computes a proper 3-coloring of a given graph or correctly reports that no such coloring exists, using the magic black box as a subroutine. *[Hint: The input to the magic black box is a graph. Just a graph. Vertices and edges. Nothing else.]*

CS 473 ✧ Spring 2016

Homework 11

Solutions will be released on Tuesday, May 3, 2016.

This homework will not be graded.
However, material covered by this homework *may* appear on the final exam.

---

1. The ***linear arrangement*** problem asks, given an n-vertex directed graph as input, for an ordering $v_1, v_2, \ldots, v_n$ of the vertices that maximizes the number of forward edges: directed edges $v_i \rightarrow v_j$ such that $i < j$. Describe and analyze an efficient 2-approximation algorithm for this problem. (Solving this problem exactly is NP-hard.)

2. Let $G = (V, E)$ be an undirected graph, each of whose vertices is colored either red, green, or blue. An edge in G is *boring* if its endpoints have the same color, and *interesting* if its endpoints have different colors. The *most interesting 3-coloring* is the 3-coloring with the maximum number of interesting edges, or equivalently, with the fewest boring edges. Computing the most interesting 3-coloring is NP-hard, because the standard 3-coloring problem is a special case.

   (a) Let $wow(G)$ denote the number of interesting edges in the most interesting 3-coloring of G. Suppose we independently assign each vertex in G a *random* color from the set $\{\text{red}, \text{green}, \text{blue}\}$. Prove that the expected number of interesting edges is at least $\frac{2}{3} wow(G)$.

   (b) Prove that with high probability, the expected number of interesting edges is at least $\frac{1}{2} wow(G)$. *[Hint: Use Chebyshev's inequality. But wait... How do we know that we* ***can*** *use Chebyshev's inequality?]*

   (c) Let $zzz(G)$ denote the number of boring edges in the most interesting 3-coloring of a graph G. Prove that it is NP-hard to approximate $zzz(G)$ within a factor of $10^{10^{100}}$.

3. Suppose we want to schedule a give set of n jobs on on a machine containing a row of p identical processors. Our input consists of two arrays $duration[1..n]$ and $width[1..n]$. A valid schedule consists of two arrays $start[1..n]$ and $first[1..n]$ that satisfy the following constraints:

   - $start[j] \geq 0$ for all j.
   - The jth job runs on processors $first[j]$ through $first[j] + width[j] - 1$, starting at time $start[j]$ and ending at time $start[j] + duration[j]$.
   - No processor can run more than one job simultaneously.

   The *makespan* of a schedule is the largest finishing time: $\max_j(start[j] + duration[j])$. Our goal is to compute a valid schedule with the smallest possible makespan.

   (a) Prove that this scheduling problem is NP-hard.

(b) Describe a polynomial-time algorithm that computes a 3-approximation of the minimum makespan of the given set of jobs. That is, if the minimum makespan is M, your algorithm should compute a schedule with makespan at most $3M$. You may assume that p is a power of 2. *[Hint: Assume that p is a power of 2.]*

(c) Describe an algorithm that computes a 3-approximation of the minimum makespan of the given set of jobs ***in $O(n \log n)$ time***. Again, you may assume that p is a power of 2.

These are the standard 10-point rubrics that we will use for certain types of exam questions. When these problems appear in the homework, a score of x on this 10-point scale corresponds to a score of $\lceil x/3 \rceil$ on the 4-point homework scale.

Proof by Induction

- 2 points for stating a valid **strong** induction hypothesis.
  - The inductive hypothesis need not be stated explicitly if it is a mechanical translation of the theorem (that is, "Assume $P(k)$ for all $k < n$" when the theorem is "$P(n)$ for all n") *and* it is applied correctly. However, if the proof requires a stronger induction hypothesis ("Assume $P(k)$ and $Q(k)$ for all $k < n$") then it must be stated explicitly.
  - By course policy, ***stating a weak inductive hypothesis triggers an automatic zero***, unless the proof is otherwise *perfect*.
  - ***Ambiguous*** induction hypotheses like "Assume the statement is true for all $k < n$." are not valid. *What* statement? The theorem you're trying to prove doesn't use the variable k, so that can't possibly be the statement you mean.
  - ***Meaningless*** induction hypotheses like "Assume that k is true for all $k < n$" are not valid. Only propositions can be true or false; k is an integer, not a proposition.
  - ***False*** induction hypotheses like "Assume that $k < n$ for all k" are not valid. The inequality $k < n$ does *not* hold for all k, because it does not hold when $k = n + 5$.
- 1 point for explicit and clearly exhaustive case analysis.
  - No penalty for overlapping or redundant cases. However, mistakes in redundant cases are still penalized.
- 2 points for the base case(s).
- 2 point for correctly applying the *stated* inductive hypothesis.
  - It is not possible to correctly apply an invalid inductive hypothesis.
  - No credit for correctly applying a different induction hypothesis than the one stated.
- 3 points for other details of the inductive case(s).

Dynamic Programming

- **6 points for a correct recurrence**, described either using functional notation or as pseudocode for a recursive algorithm.
  - + 1 point for a clear **English** description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) ***Automatic zero if the English description is missing.***
  - + 1 point for stating how to call your recursive function to get the final answer.
  - + 1 point for the base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 points for the recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- **4 points for iterative details**
  - + 1 point for describing the memoization data structure; a clear picture may be sufficient.
  - + 2 points for describing a correct evaluation order; a clear picture may be sufficient. If you use nested loops, be sure to specify the nesting order.
  - + 1 point for running time
- Proofs of correctness are not required for full credit on exams, unless the problem specifically asks for one.
- Do not analyze (or optimize) space.
- 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 says otherwise.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative psuedocode 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. However, you ***must*** give an English description of the underlying recursive function.
- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is ***scaled*** to the new maximum score. All points above 10 are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we have included them, significantly more students turned in algorithms that meet the target time bound but didn't work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).

Graph Reductions

For problems solved by reducing them to a standard graph algorithm covered either in class or in a prerequisite class (for example: shortest paths, topological sort, minimum spanning trees, maximum flows, bipartite maximum matching, vertex-disjoint paths, . . .):

- **1 point for listing the vertices of the graph.** (If the original input is a graph, describing how to modify that graph is fine.)
- **1 point for listing the edges of the graph**, including whether the edges are directed or undirected. (If the original input is a graph, describing how to modify that graph is fine.)
- **1 point for describing appropriate weights** and/or lengths and/or capacities and/or costs and/or demands and/or whatever for the vertices and edges.
- **2 points for an explicit description of the problem being solved on that graph.** (For example: "We compute the maximum number of vertex-disjoint paths in G from v to z.")
- **3 points for other algorithmic details**, assuming the rest of the reduction is correct.
  + 1 point for describing how to build the graph from the original input (for example: "by brute force")
  + 1 point for describing the algorithm you use to solve the graph problem (for example: "Orlin's algorithm" or "as described in class")
  + 1 point for describing how to extract the output for the original problem from the output of the graph algorithm.
- **2 points for the running time**, expressed in terms of the original input parameters, not just V and E.
- **If the problem explicitly asks for a proof of correctness,** divide all previous points in half and add **5 points for proof of correctness.** These proofs almost always have two parts; for example, for algorithms that return TRUE or FALSE:
  - 2½ points for proving that if your algorithm returns TRUE, then the correct answer is TRUE.
  - 2½ points for proving that if your algorithm returns FALSE, then the correct answer is FALSE.

  These proofs do not need to be as detailed as in the homeworks; we are really just looking for compelling evidence that *you* understand why your reduction is correct.
- It is still possible to get partial credit for an incorrect algorithm. For example, if you describe an algorithm that sometimes reports false positives, but you prove that all FALSE answers are correct, you would still get 2½ points for half of the correctness proof.

NP-Hardness Reductions

For problems that ask "***Prove*** that X is NP-hard":

- **4 points for the polynomial-time reduction:**
  - 1 point for explicitly naming the NP-hard problem Y to reduce from. You may use any of the problems listed in the lecture notes; a list of NP-hard problems will appear on the back page of the exam.
  - 2 points for describing the polynomial-time algorithm to transform arbitrary instances of Y into inputs to the black-box algorithm for X
  - 1 point for describing the polynomial-time algorithm to transform the output of the black-box algorithm for X into the output for Y.
  - Reductions that call the black-box algorithm for X more than once are perfectly acceptable. You do *not* need to explicitly analyze the running time of your resulting algorithm for Y, but it must be polynomial in the size of the input instance of Y.

- **6 points for the proof of correctness. This is the entire point of the problem.** These proofs always have two parts; for example, if X and Y are both decision problems:
  - 3 points for proving that your reduction transforms positive instances of Y into positive instances of X.
  - 3 points for proving that your reduction transforms negative instances of Y into negative instances of X.

  These proofs do not need to be as detailed as in the homeworks; however, it must be clear that you have at least considered all possible cases. We are really just looking for compelling evidence that *you* understand why your reduction is correct.

- It is still possible to get partial credit for an incorrect reduction. For example, if you describe a reduction that sometimes reports false positives, but you prove that all FALSE answers are correct, you would still get 3 points for half of the correctness proof.

- Zero points for reducing X ***to*** some NP-hard problem Y.

- Zero points for attempting to solve X.

Approximation Algorithms

For problems that ask you to describe a polynomial-time approximation algorithm for some NP-hard problem X, analyze its approximation ratio, and prove that your approximation analysis is correct:

- **4 points for the actual approximation algorithm.** You do not need to analyze the running time of your algorithm (unless we explicitly ask for the running time), but it must clearly run in polynomial time. If we give you the algorithm, ignore this part and scale the rest of the rubric up to 10 points.
- **2 points for stating the correct approximation ratio.** If we give you the approximation ratio, ignore this part and scale the rest of the rubric up to 10 points.
- **4 points for proving that the stated approximation ratio is correct.** If we do not *explicitly* ask for a proof, ignore this part and scale the rest of the rubric up to 10 points.

For example, suppose we give you an algorithm and ask for its approximation ratio, but we do not explicitly ask for a proof. If the given algorithm is a 3-approximation algorithm, then you would get full credit for writing “3”.

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. For any positive integer n, the nth ***Fibonacci string*** F_n is defined recursively as follows, where $x \bullet y$ denotes the concatenation of strings x and y:

$$\begin{aligned} F_1 &:= \texttt{0} \\ F_2 &:= \texttt{1} \\ F_n &:= F_{n-1} \bullet F_{n-2} \quad \text{for all } n \geq 3 \end{aligned}$$

For example, $F_3 = \texttt{10}$ and $F_4 = \texttt{101}$.

(a) What is F_8?

(b) ***Prove*** that every Fibonacci string except F_1 starts with `1`.

(c) ***Prove*** that no Fibonacci string contains the substring `00`.

2. You have reached the inevitable point in the semester where it is no longer possible to finish all of your assigned work without pulling at least a few all-nighters. The problem is that pulling successive all-nighters will burn you out, so you need to pace yourself (or something).

Let's model the situation as follows. There are n days left in the semester. For simplicity, let's say you are taking one class, there are no weekends, there is an assignment due every single day until the end of the semester, and you will only work on an assignment the day before it is due. For each day i, you know two positive integers:

- $Score[i]$ is the score you will earn on the ith assignment if you do *not* pull an all-nighter the night before.
- $Bonus[i]$ is the number of additional points you could potentially earn if you *do* pull an all-nighter the night before.

However, pulling multiple all-nighters in a row has a price. If you turn in the ith assignment immediately after pulling k consecutive all-nighters, your actual score for that assignment will be $(Score[i] + Bonus[i])/2^{k-1}$.

Design and analyze an algorithm that computes the maximum total score you can achieve, given the arrays $Score[1..n]$ and $Bonus[1..n]$ as input.

3. The following algorithm finds the smallest element in an unsorted array. The subroutine Shuffle randomly permutes the input array A; every permutation of A is equally likely.

```
RandomMin($A[1..n]$):
    $min \leftarrow \infty$
    Shuffle($A$)
    for $i \leftarrow 1$ to $n$
        if $A[i] < min$
            $min \leftarrow A[i]$        $(\star)$
    return $min$
```

In the following questions, assume all elements in the input array $A[\,]$ are distinct.

(a) In the worst case, how many times does RandomMin execute line $(\star)$?

(b) For each index i, let $X_i = 1$ if line $(\star)$ is executed in the ith iteration of the for loop, and let $X_i = 0$ otherwise. What is $\Pr[X_i = 1]$? *[Hint: First consider $i = 1$ and $i = n$.]*

(c) What is the *exact* expected number of executions of line $(\star)$?

(d) ***Prove*** that line $(\star)$ is executed $O(\log n)$ times with high probability, *assuming* the variables X_i are mutually independent.

(e) **[Extra credit]** ***Prove*** that the variables X_i are mutually independent.
[Hint: Finish the rest of the exam first!]

4. Your eight-year-old cousin Elmo decides to teach his favorite new card game to his baby sister Daisy. At the beginning of the game, n cards are dealt face up in a long row. Each card is worth some number of points, which may be positive, negative, or zero. Then Elmo and Daisy take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, each player can decide which of the two cards to take. When the game ends, the player that has collected the most points wins.

Daisy isn't old enough to get this whole "strategy" thing; she's just happy to play with her big brother. When it's her turn, she takes the either leftmost card or the rightmost card, each with probability $1/2$.

Elmo, on the other hand, *really* wants to win. Having never taken an algorithms class, he follows the obvious greedy strategy—when it's his turn, Elmo *always* takes the card with the higher point value.

Describe and analyze an algorithm to determine Elmo's expected score, given the initial sequence of n cards as input. Assume Elmo moves first, and that no two cards have the same value.

For example, suppose the initial cards have values $1, 4, 8, 2$. Elmo takes the 2, because it's larger than 1. Then Daisy takes either 1 or 8 with equal probability. If Daisy takes the 1, then Elmo takes the 8; if Daisy takes the 8, then Elmo takes the 4. Thus, Elmo's expected score is $2 + (8 + 4)/2 = 8$.

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.
Red text reflects corrections or clarifications given during the actual exam.

1. Suppose we insert n distinct items into an initially empty hash table of size $m \gg n$, using an ***ideal random*** hash function h. Recall that a collision is a set of two distinct items $\{x, y\}$ in the table such that $h(x) = h(y)$.

   (a) What is the exact expected number of collisions?

   (b) Estimate the probability that there are no collisions. *[Hint: Use Markov's inequality.]*

   (c) Estimate the largest value of n such that the probability of having no collisions is at least $1 - 1/n$. Your answer should have the form $n = O(f(m))$ for some simple function f.

   (d) Fix an integer $k > 1$. A ***k-way collision*** is a set of k distinct items $\{x_1, \dots, x_k\}$ that all have the same hash value: $h(x_1) = h(x_2) = \cdots = h(x_k)$. Estimate the largest value of n such that the probability of having no k-way collisions is at least $1 - 1/n$. Your answer should have the form $n = O(f(m, k))$ for some simple function f. *[Hint: You may want to repeat parts (a) and (b).]*

2. Quentin, Alice, and the other Brakebills Physical Kids are planning an excursion through the Neitherlands to Fillory. The Neitherlands is a vast, deserted city composed of several plazas, each containing a single fountain that can magically transport people to a different world. Adjacent plazas are connected by gates, which have been cursed by the Beast. The gates open for only five minutes every hour, all at the same time. During those five minutes, if more than one person passes through any single gate, the Beast will detect their presence.□ However, people can safely pass through *different* gates at the same time. Moreover, anyone attempting to pass through more than one gate in the same five-minute period will turn into a niffin.□

   You are given a map of the Neitherlands, which is a graph G with a vertex for each fountain and an edge for each gate, with the fountains to Earth and Fillory clearly marked; you are also given a positive integer h. Describe and analyze an algorithm to compute the maximum number of people that can walk from the Earth fountain to the Fillory fountain in h hours, without anyone alerting the Beast or turning into a niffin.

□This is very bad.
□This is very bad.

3. Recall that a ***Bloom filter*** is an array $B[1..m]$ of bits, together with a collection of k independent ideal random hash functions $h_1, h_2, \ldots, h_k$. To insert an item x into a Bloom filter, we set $B[h_i(x)] \leftarrow 1$ for every index i. To test whether an item x belongs to a set represented by a Bloom filter, we check whether $B[h_i(x)] = 1$ for every index i. This algorithm always returns TRUE if x is in the set, but may return either TRUE or FALSE when x is not in the set. Thus, there may be false positives, but no false negatives.

   If there are n distinct items stored in the Bloom filter, then the probability of a false positive is $(1-p)^k$, where $p \approx e^{-kn/m}$ is the probability that $B[j] = 0$ for any particular index j. In particular, if we set $k = (m/n)\ln 2$, then $p = 1/2$, and the probability of a false positive is $(1/2)^{(m/n)\ln 2} \approx (0.61850)^{m/n}$.

   After months spent lovingly crafting a Bloom filter of size m for a set S of n items, using exactly $k = (m/n)\ln 2$ hash functions (so $p = 1/2$), your boss tells you that you must reduce the size of your Bloom filter from m bits down to $m/2$ bits. Unfortunately, you no longer have the original set S, and your company's product ships tomorrow; you have to do something quick and dirty. Fortunately, your boss has a couple of ideas.

   (a) First your boss suggests simply discarding half of the Bloom filter, keeping only the subarray $B[1..m/2]$. Describe an algorithm to check whether a given item x is an element of the original set S, using only this smaller Bloom filter. As usual, if $x \in S$, your algorithm ***must*** return TRUE.

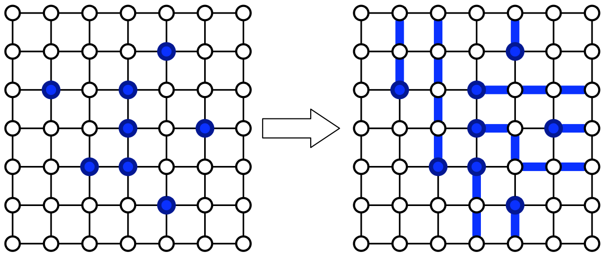
   (b) What is the probability that your algorithm returns TRUE when $x \notin S$?

   (c) Next your boss suggests merging the two halves of your old Bloom filter, defining a new array $B'[1..m/2]$ by setting $B'[i] \leftarrow B[i] \vee B[i + m/2]$ for all i. Describe an algorithm to check whether a given item x is an element of the original set S, using only this smaller Bloom filter B'. As usual, if $x \in S$, your algorithm ***must*** return TRUE.

   (d) What is the probability that your algorithm returns TRUE when $x \notin S$?

4. An $n \times n$ grid is an undirected graph with n^2 vertices organized into n rows and n columns. We denote the vertex in the ith row and the jth column by (i, j). Every vertex (i, j) has exactly four neighbors $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, and $(i, j+1)$, except the *boundary* vertices, for which $i = 1$, $i = n$, $j = 1$, or $j = n$.

   Let $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ be distinct vertices, called *terminals*, in the $n \times n$ grid. The ***escape problem*** is to determine whether there are m vertex-disjoint paths in the grid that connect these terminals to any m distinct boundary vertices. Describe and analyze an efficient algorithm to solve the escape problem.



A positive instance of the escape problem, and its solution.

Write your answers in the separate answer booklet.

Please return this question handout and your cheat sheets with your answers.

1. Let $G = (V, E)$ be an arbitrary undirected graph. A ***triple-Hamiltonian circuit*** in G is a closed walk in G that visits every vertex of G exactly *three* times. ***Prove*** that it is NP-hard to determine whether a given undirected graph has a triple-Hamiltonian circuit. *[Hint: Modify your reduction for double-Hamiltonian circuits from Homework 10.]*

2. Marie-Joseph Paul Yves Roch Gilbert du Motier, Marquis de Lafayette, colonial America's favorite fighting Frenchman, needs to choose a subset of his ragtag volunteer army of m soldiers to complete a set of n important tasks, like "go to France for more funds" or "come back with more guns". Each task requires a specific set of skills, such as "knows what to do in a trench" or "ingenuitive and fluent in French". For each task, exactly k soldiers are qualified to complete that task.

   Unfortunately, Lafayette's soldiers are extremely lazy. For each task, if Lafayette chooses more than one soldier qualified for that task, each of them will assume that someone else will take on that task, and so the task will never be completed. A task will be completed if and only if exactly one of the chosen soldiers has the necessary skills for that task.

   So Lafayette needs to choose a subset S of soldiers that maximizes the number of tasks for which *exactly one* soldier in S is qualified. Not surprisingly, Lafayette's problem is NP-hard.

   (a) Suppose Lafayette chooses each soldier independently with probability p. What is the *exact* expected number of tasks that will be completed, in terms of p and k?

   (b) What value of p maximizes this expected value?

   (c) Describe a randomized polynomial-time $O(1)$-approximation algorithm for Lafayette's problem. What is the expected approximation ratio for your algorithm?

3. Suppose we are given a set of n rectangular boxes, each specified by their height, width, and depth in centimeters. All three dimensions of each box lie strictly between 10cm and 20cm, and all $3n$ dimensions are distinct. As you might expect, one box can be nested inside another if the first box can be rotated so that is is smaller in every dimension than the second box. Boxes can be nested recursively, but two boxes cannot be nested side-by-side inside a third box. A box is *visible* if it is not nested inside another box.

   Describe and analyze an algorithm to nest the boxes, so that the number of visible boxes is as small as possible.

4. Hercules Mulligan, a tailor spyin' on the British government, has determined a set of routes and towns that the British army plans to use to move their troops from Charleston, South Carolina to Yorktown, Virginia. (He took their measurements, information, and then he smuggled it.) The American revolutionary army wants to set up ambush points in some of these towns, so that every unit of the British army will face at least one ambush before reaching Yorktown. On the other hand, General Washington wants to leave as many troops available as possible to help defend Yorktown when the British army inevitably arrives.

   Describe an efficient algorithm that computes the smallest number of towns where the revolutionary army should set up ambush points. The input to your algorithm is Mulligan's graph of towns (vertices) and routes (edges), with Charleston and Yorktown clearly marked.

5. Consider the following randomized algorithm to approximate the smallest vertex cover in an undirected graph $G = (V, E)$. For each vertex $v \in V$, define the *priority* of v to be a real number between 0 and 1, chosen independently and uniformly at random. Finally, let S be the subset of vertices with higher priority than at least one of their neighbors:

$$S := \left\{ v \in V \;\middle|\; \mathit{priority}(v) > \min_{uv \in E} \mathit{priority}(u) \right\}$$

   (a) What is the probability that the set S is a vertex cover of G? ***Prove*** your answer is correct. (Your proof should be *short*.)

   (b) Suppose the input graph G is a cycle of length n. What is the *exact* expected size of S?

   (c) Suppose the input graph G is a ***star***: a tree with one vertex of degree $n-1$ and $n-1$ vertices of degree 1. What is the *exact* probability that S is the *smallest* vertex cover of G?

   (d) Again, suppose G is a star. Suppose we run the randomized algorithm N times, generating a sequence of subsets $S_1, S_2, \dots, S_N$. How large must N be to guarantee with high probability that some S_i is the minimum vertex cover of G?

6. 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 *profit*$[1..n]$ and *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 $i + \mathit{skip}[i]$.

   Design and analyze an algorithm that determines the maximum total profit Burr can secure from these n cases, using his two arrays as input.

CS/ECE 374 ✧ Fall 2016

Homework 0

Due Tuesday, August 30, 2016 at 8pm

- **Each student must submit individual solutions for this homework.** For all future homeworks, groups of up to three students can submit joint solutions.
- **Submit your solutions electronically on the course Gradescope site 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, not just a phone camera).
- You are ***not*** required to sign up on Gradescope (or Piazza) with your real name and your illinois.edu email address; you may use any email address and alias of your choice. However, to give you credit for the homework, we need to know who Gradescope thinks you are. **Please fill out the web form linked from the course web page.**

☞ Some important course policies ☜

- **You may use any source at your disposal**—paper, electronic, or human—but you ***must*** cite ***every*** source that you use, and you ***must*** write everything yourself in your own words. See the academic integrity policies on the course web site for more details.
- The answer ***"I don't know"*** (and *nothing* else) is worth 25% partial credit on any required problem or subproblem, on any homework or exam. We will accept synonyms like "No idea" or "WTF" or "¯\(•_•)/¯", but you must write *something*.
- **Avoid the Three Deadly Sins!** Any homework or exam solution that breaks any of the following rules will be given an ***automatic zero***, unless the solution is otherwise perfect. Yes, we really mean it. We're not trying to be scary or petty (Honest!), but we do want to break a few common bad habits that seriously impede mastery of the course material.
  - Always give complete solutions, not just examples.
  - Always declare all your variables, in English. In particular, always describe the specific problem your algorithm is supposed to solve.
  - Never use weak induction.

See the course web site for more information.

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

1. The famous Czech professor Jiřina Z. Džunglová has a favorite 23-node binary tree, in which each node is labeled with a unique letter of the alphabet. Preorder and inorder traversals of the tree visit the nodes in the following order:

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

   (a) List the nodes in Professor Džunglová's tree in post-order.

   (b) Draw Professor Džunglová's tree.

2. The ***complement*** $\boldsymbol{w^c}$ of a string $w \in \{\mathtt{0},\mathtt{1}\}^*$ is obtained from w by replacing every $\mathtt{0}$ in w with a $\mathtt{1}$ and vice versa; for example, $\mathtt{111011000100}^c = \mathtt{000100111011}$. The complement function is formally defined as follows:

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

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

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

   Your proofs must be formal and self-contained, and they must invoke the *formal* definitions of length $|w|$, concatenation $x \bullet y$, and complement w^c. Do not appeal to intuition!

3. Recursively define a set L of strings over the alphabet $\{\mathtt{0},\mathtt{1}\}$ as follows:

   - The empty string ε is in L.
   - For all strings x and y in L, the string $\mathtt{0}x\mathtt{1}y$ is also in L.
   - For all strings x and y in L, the string $\mathtt{1}x\mathtt{0}y$ is also in L.
   - These are the only strings in L.

   Let $\#(a, w)$ denote the number of times symbol a appears in string w; for example,

$$\#(\mathtt{0}, \mathtt{01000110111001}) = \#(\mathtt{1}, \mathtt{01000110111001}) = 7.$$

   (a) Prove that the string $\mathtt{01000110111001}$ is in L.

   (b) Prove by induction that every string in L has exactly the same number of $\mathtt{0}$s and $\mathtt{1}$s. (You may assume without proof that $\#(a, xy) = \#(a, x) + \#(a, y)$ for any symbol a and any strings x and y.)

   (c) Prove by induction that L contains every string with the same number of $\mathtt{0}$s and $\mathtt{1}$s.

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

Solved Problems

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

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

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

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

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

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

In parts (b) and (c), you may assume without proof that $(x \cdot y)^R = y^R \bullet x^R$ and $(x^R)^R = x$ for all strings x and y.

Solution:

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

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

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

(b) Let w be an arbitrary palindrome.
Assume that $x = x^R$ for every palindrome x such that $|x| < |w|$.
There are three cases to consider (mirroring the three cases in the definition):

- 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 $w = w^R$.
- Suppose $w = axa$ for some symbol $a \in \Sigma$ and some palindrome $x \in P$. Then

$$\begin{aligned} w^R &= (a \cdot x \bullet a)^R \\ &= (x \bullet a)^R \bullet a && \text{by definition of reversal} \\ &= a^R \bullet x^R \bullet a && \text{You said we could assume this.} \\ &= a \bullet x^R \bullet a && \text{by definition of reversal} \\ &= a \bullet x \bullet a && \text{by the inductive hypothesis} \\ &= w && \text{by assumption} \end{aligned}$$

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

Rubric: 4 points: standard induction rubric (scaled)

(c) Let w be an arbitrary string such that $w = w^R$.

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

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

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

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

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

  Because $a = b$, we have $w = ay^R a$ and $w^R = aya$.
  The recursive definition of string equality implies $y^R a = ya$.
  It immediately follows that $(y^R a)^R = (ya)^R$.
  Known properties of reversal imply $(y^R a)^R = a(y^R)^R = ay$ and $(ya)^R = ay^R$.
  It follows that $ay^R = ay$, and therefore $y = y^R$.
  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).

- No penalty for jumping from $aya = ay^R a$ directly to $y = y^R$.

■

Rubric (induction): For problems worth 10 points:

- + 1 for explicitly considering an *arbitrary* object
- + 2 for a valid **strong** induction hypothesis
  - – **Deadly Sin!** Automatic zero for stating a weak induction hypothesis, unless the rest of the proof is *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 cases overlap (for example:
- + 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.

CS/ECE 374 ✧ Fall 2016

Homework 1

Due Tuesday, September 6, 2016 at 8pm

Starting with this homework, groups of up to three people can submit joint solutions. Each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the Gradescope names and email addresses of each group member. In addition, whoever submits the homework must tell Gradescope who their other group members are.

---

1. For each of the following languages over the alphabet $\{0, 1\}$, give a regular expression that describes that language, and briefly argue why your expression is correct.

   (a) All strings that end with the suffix `01010101`.

   (b) All strings except `111`.

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

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

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

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

2. This problem considers two special classes of regular expressions.

   - A regular expression R is ***plus-free*** if and only if it never uses the $+$ operator.
   - A regular expression R is ***top-plus*** if and only if either
     - R is plus-free, or
     - $R = S + T$, where S and T are top-plus.

   For example, $1((0^*10)^*1)^*0$ is plus-free and (therefore) top-plus; $01^*0 + 10^*1 + \varepsilon$ is top-plus but not plus-free, and $0(0+1)^*(1+\varepsilon)$ is neither top-plus nor plus-free.

   Recall that two regular expressions R and S are ***equivalent*** if they describe exactly the same language: $L(R) = L(S)$.

   (a) Prove that for any top-plus regular expressions R and S, there is a top-plus regular expression that is equivalent to RS. *[Hint: Use the fact that $(A+B)(C+D)$ and $AC+AD+BC+BD$ are equivalent, for all regular expressions A, B, C, and D.]*

   (b) Prove that for any top-plus regular expression R, there is a ***plus-free*** regular expression S such that R^* and S^* are equivalent. *[Hint: Use the fact that $(A+B)^*$ is equivalent to $(A^*B^*)^*$, for all regular expressions A and B.]*

   (c) Prove that for any regular expression, there is an equivalent top-plus regular expression.

3. Let L be the set of all strings in $\{0, 1\}^*$ that contain exactly two occurrences of the substring 001.

   (a) Describe a DFA that over the alphabet $\Sigma = \{0, 1\}$ that accepts the language L. Argue that your machine accepts every string in L and nothing else, by explaining what each state in your DFA *means*.

   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.

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

Solved problem

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

   - /⋆/
   - //◇//◇↲◇↲
   - /⋆◇/⋆◇⋆/◇⋆/

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

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

   You must explain *in English* how your DFAs work. Drawings or formal descriptions without English explanations will receive no credit, even if they are correct.

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

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

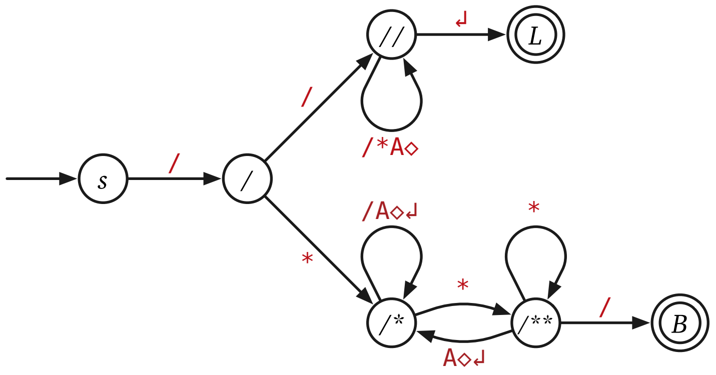
For example, the string "/⋆\\\"⋆/"/⋆"/⋆\"/⋆"⋆/ is a valid string literal (representing the 5-character string /⋆\"\⋆/, which is itself a valid block comment!) followed immediately by a valid block comment. ***For this homework question, just pretend that the characters ’, ", and \ don’t exist.***

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

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

Solution:

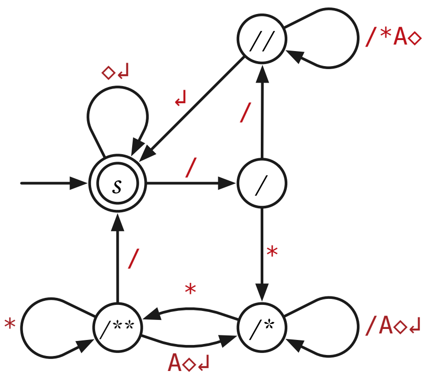
(a) The following eight-state DFA recognizes the language of C comments. All missing transitions lead to a hidden reject state.



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 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 read a complete block comment.

(b) 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 $\star$ after the opening $/\star$.
- /** — We are reading a block comment, and we just read a $\star$ after the opening $/\star$. ■

Rubric: 10 points = 5 for each part, using the standard DFA design rubric (scaled)

Rubric (DFA design): 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.
  - **For drawings:** Use an arrow from nowhere to indicate s, and doubled circles to indicate accepting states A. If $A = \varnothing$, say so explicitly. If your drawing omits a reject state, say so explicitly. **Draw neatly!** If we can't read your solution, we can't give you credit for it,.
  - **For text descriptions:** You can describe the transition function either using a 2d array, using mathematical notation, or using an algorithm.
  - **For product constructions:** You must give a complete description of the states and transition functions of the DFAs you are combining (as either drawings or text), together with the accepting states of the product DFA.
- **Homework only:** 4 points for *briefly* and correctly explaining the purpose of each state *in English*. This is how you justify that your DFA is correct.
  - For product constructions, explaining the states in the factor DFAs is enough.
  - **Deadly Sin:** ("Declare your variables.") No credit for the problem if the English description is missing, *even if the DFA is correct*.
- 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 accept 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.
- DFA drawings with too many states may be penalized. DFA drawings with *significantly* too many states may get no credit at all.
- Half credit for describing an NFA when the problem asks for a DFA.

CS/ECE 374 ✧ Fall 2016

Homework 2

Due Tuesday, September 13, 2016 at 8pm

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

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

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

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

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

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

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

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

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

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

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

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

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

   (a) $\left\{w \in (\mathtt{0}+\mathtt{1})^* \;\middle|\; |\#(\mathtt{0},w) - \#(\mathtt{1},w)| < 5\right\}$

   (b) Strings in $(\mathtt{0}+\mathtt{1})^*$ in which the substrings $\mathtt{00}$ and $\mathtt{11}$ appear the same number of times.

   (c) $\left\{\mathtt{0}^m\mathtt{1}\mathtt{0}^n \;\middle|\; n/m \text{ is an integer}\right\}$

3. Let L be an arbitrary regular language.

   (a) Prove that the language $\mathit{palin}(L) := \{w \mid ww^R \in L\}$ is also regular.

   (b) Prove that the language $\mathit{drome}(L) := \{w \mid w^R w \in L\}$ is also regular.

Solved problem

4. Let L be an arbitrary regular language. 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:

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

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

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. ■

Rubric: 5 points =

- + 1 for a formal, complete, and unambiguous description of a DFA or NFA
  - No points for the rest of the problem if this is missing.
- + 3 for a correct NFA
  - −1 for a single mistake in the description (for example a typo)
- + 1 for a *brief* English justification. We explicitly do *not* want a formal proof of correctness, but we do want one or two sentences explaining how the NFA works.

CS/ECE 374 ✧ Fall 2016

Homework 3

Due Tuesday, September 20, 2016 at 8pm

Groups of up to three people can submit joint solutions. Each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the Gradescope names and email addresses of each group member. In addition, whoever submits the homework must tell Gradescope who their other group members are.

---

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

   - An NFA that accepts the same language, obtained using Thompson's recursive algorithm
   - An equivalent DFA, obtained using the incremental subset construction. For each state in your DFA, identify the corresponding subset of states in your NFA. Your DFA should have no unreachable states.

   (a) $(\mathtt{00}+\mathtt{11})^*(\mathtt{0}+\mathtt{1}+\varepsilon)$

   (b) $\mathtt{1}^*+(\mathtt{01})^*+(\mathtt{001})^*$

2. Give context-free grammars for the following languages, and clearly explain how they work and the role of each nonterminal. Grammars can be very difficult to understand; if the grader does not understand how your construction is intended to generate the language, then you will receive no credit.

   (a) In any string, a ***block*** (also called a ***run***) is a maximal non-empty substring of identical symbols. For example, the string 0111000011001 has six blocks: three blocks of 0s of lengths 1, 4, and 2, and three blocks of 1s of lengths 3, 2, and 1.

   Let L be the set of all strings in $\{\mathtt{0},\mathtt{1}\}^*$ that contain two blocks of 0s of equal length. For example, L contains the strings 01101111 and 01001011100010 but does not contain the strings 000110011011 and 00000000111.

   (b) $L = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w \text{ is } \textit{not} \text{ a palindrome}\}$.

3. Let $L = \{\mathtt{0}^i\mathtt{1}^j\mathtt{2}^k \mid k = i + j\}$.

   (a) Show that L is context-free by describing a grammar for L.

   (b) Prove that your grammar G is correct. As usual, you need to prove both $L \subseteq L(G)$ and $L(G) \subseteq L$.

Solved problem

4. Let L be the set of all strings over $\{0,1\}^*$ with exactly twice as many 0s as 1s.

   (a) Describe a CFG for the language L.
   [Hint: For any string u define $\Delta(u) = \#(0,u) - 2\#(1,u)$. Introduce intermediate variables that derive strings with $\Delta(u) = 1$ and $\Delta(u) = -1$ and use them to define a non-terminal that generates L.]

   Solution: $S \to \varepsilon \mid SS \mid 00S1 \mid 0S1S0 \mid 1S00$ ■

   (b) Prove that your grammar G is correct. As usual, you need to prove both $L \subseteq L(G)$ and $L(G) \subseteq L$.
   [Hint: Let $u_{\leq i}$ denote the prefix of u of length i. If $\Delta(u) = 1$, what can you say about the smallest i for which $\Delta(u_{\leq i}) = 1$? How does u split up at that position? If $\Delta(u) = -1$, what can you say about the smallest i such that $\Delta(u_{\leq i}) = -1$?]

   Solution: (Hopefully you recognized this as a more advanced version of HW0 problem 3.) We separately prove $L \subseteq L(G)$ and $L(G) \subseteq L$ as follows:

   Claim 1. *$L(G) \subseteq L$, that is, every string in $L(G)$ has exactly twice as many 0s as 1s.*

   Proof: As suggested by the hint, for any string u, let $\Delta(u) = \#(0,u) - 2\#(1,u)$. We need to prove that $\Delta(w) = 0$ for every string $w \in L(G)$.

   Let w be an arbitrary string in $L(G)$, and consider an arbitrary derivation of w of length k. Assume that $\Delta(x) = 0$ for every string $x \in L(G)$ that can be derived with fewer than k productions.□ There are five cases to consider, depending on the first production in the derivation of w.

   - If $w = \varepsilon$, then $\#(0,w) = \#(1,w) = 0$ by definition, so $\Delta(w) = 0$.
   - Suppose the derivation begins $S \rightsquigarrow SS \rightsquigarrow^* w$. Then $w = xy$ for some strings $x, y \in L(G)$, each of which can be derived with fewer than k productions. The inductive hypothesis implies $\Delta(x) = \Delta(y) = 0$. It immediately follows that $\Delta(w) = 0$.□
   - Suppose the derivation begins $S \rightsquigarrow 00S1 \rightsquigarrow^* w$. Then $w = 00x1$ for some string $x \in L(G)$. The inductive hypothesis implies $\Delta(x) = 0$. It immediately follows that $\Delta(w) = 0$.
   - Suppose the derivation begins $S \rightsquigarrow 1S00 \rightsquigarrow^* w$. Then $w = 1x00$ for some string $x \in L(G)$. The inductive hypothesis implies $\Delta(x) = 0$. It immediately follows that $\Delta(w) = 0$.
   - Suppose the derivation begins $S \rightsquigarrow 0S1S1 \rightsquigarrow^* w$. Then $w = 0x1y0$ for some strings $x, y \in L(G)$. The inductive hypothesis implies $\Delta(x) = \Delta(y) = 0$. It immediately follows that $\Delta(w) = 0$.

   In all cases, we conclude that $\Delta(w) = 0$, as required. □

□Alternatively: Consider the *shortest* derivation of w, and assume $\Delta(x) = 0$ for every string $x \in L(G)$ such that $|x| < |w|$.

□Alternatively: Suppose the *shortest* derivation of w begins $S \rightsquigarrow SS \rightsquigarrow^* w$. Then $w = xy$ for some strings $x, y \in L(G)$. Neither x or y can be empty, because otherwise we could shorten the derivation of w. Thus, x and y are both shorter than w, so the induction hypothesis implies. . . . We need some way to deal with the decompositions $w = \varepsilon \bullet w$ and $w = w \bullet \varepsilon$, which are both consistent with the production $S \to SS$, without falling into an infinite loop.

Claim 2. *$L \subseteq L(G)$; that is, G generates every binary string with exactly twice as many* `0`*s as* `1`*s.*

Proof: As suggested by the hint, for any string u, let $\Delta(u) = \#(0, u) - 2\#(1, u)$. For any string u and any integer $0 \le i \le |u|$, let $\boldsymbol{u_i}$ denote the ith symbol in u, and let $\boldsymbol{u_{\le i}}$ denote the prefix of u of length i.

Let w be an arbitrary binary string with twice as many `0`s as `1`s. Assume that G generates every binary string x that is shorter than w and has twice as many `0`s as `1`s. There are two cases to consider:

- If $w = \varepsilon$, then $\varepsilon \in L(G)$ because of the production $S \to \varepsilon$.
- Suppose w is non-empty. To simplify notation, let $\Delta_i = \Delta(w_{\le i})$ for every index i, and observe that $\Delta_0 = \Delta_{|w|} = 0$. There are several subcases to consider:
  - Suppose $\Delta_i = 0$ for some index $0 < i < |w|$. Then we can write $w = xy$, where x and y are non-empty strings with $\Delta(x) = \Delta(y) = 0$. The induction hypothesis implies that $x, y \in L(G)$, and thus the production rule $S \to SS$ implies that $w \in L(G)$.
  - Suppose $\Delta_i > 0$ for all $0 < i < |w|$. Then w must begin with `00`, since otherwise $\Delta_1 = -2$ or $\Delta_2 = -1$, and the last symbol in w must be `1`, since otherwise $\Delta_{|w|-1} = -1$. Thus, we can write $w = \mathtt{00}x\mathtt{1}$ for some binary string x. We easily observe that $\Delta(x) = 0$, so the induction hypothesis implies $x \in L(G)$, and thus the production rule $S \to \mathtt{00}S\mathtt{1}$ implies $w \in L(G)$.
  - Suppose $\Delta_i < 0$ for all $0 < i < |w|$. A symmetric argument to the previous case implies $w = \mathtt{1}x\mathtt{00}$ for some binary string x with $\Delta(x) = 0$. The induction hypothesis implies $x \in L(G)$, and thus the production rule $S \to \mathtt{1}S\mathtt{00}$ implies $w \in L(G)$.
  - Finally, suppose none of the previous cases applies: $\Delta_i < 0$ and $\Delta_j > 0$ for some indices i and j, but $\Delta_i \ne 0$ for all $0 < i < |w|$.

    Let i be the smallest index such that $\Delta_i < 0$. Because Δ_j either increases by 1 or decreases by 2 when we increment j, for all indices $0 < j < |w|$, we must have $\Delta_j > 0$ if $j < i$ and $\Delta_j < 0$ if $j \ge i$.

    In other words, there is a *unique* index i such that $\Delta_{i-1} > 0$ and $\Delta_i < 0$. In particular, we have $\Delta_1 > 0$ and $\Delta_{|w|-1} < 0$. Thus, we can write $w = \mathtt{0}x\mathtt{1}y\mathtt{0}$ for some binary strings x and y, where $|\mathtt{0}x\mathtt{1}| = i$.

    We easily observe that $\Delta(x) = \Delta(y) = 0$, so the inductive hypothesis implies $x, y \in L(G)$, and thus the production rule $S \to \mathtt{0}S\mathtt{1}S\mathtt{0}$ implies $w \in L(G)$.

In all cases, we conclude that G generates w. □

Together, Claim 1 and Claim 2 imply $L = L(G)$. ■

Rubric: 10 points:

- part (a) = 4 points. As usual, this is not the only correct grammar.
- part (b) = 6 points = 3 points for ⊆ + 3 points for ⊇, each using the standard induction template (scaled).

CS/ECE 374 ✧ Fall 2016

Homework 4

Due Tuesday, October 4, 2016 at 8pm

---

1. Consider the following restricted variant of the Tower of Hanoi puzzle. The pegs are numbered 0, 1, and 2, and your task is to move a stack of n disks from peg 1 to peg 2. However, you are forbidden to move any disk *directly* between peg 1 and peg 2; *every* move must involve peg 0.

   Describe an algorithm to solve this version of the puzzle in as few moves as possible. *Exactly* how many moves does your algorithm make?

2. Consider the following cruel and unusual sorting algorithm.

```
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 $A[1] \leftrightarrow A[2]$
  else
      for $i \leftarrow 1$ to $n/4$                 ⟨⟨swap 2nd and 3rd quarters⟩⟩
          swap $A[i+n/4] \leftrightarrow A[i+n/2]$
      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⟩⟩
```

   Notice that the comparisons performed by the 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: Consider an array that contains $n/4$ 1s, $n/4$ 2s, $n/4$ 3s, and $n/4$ 4s. Why is this special case enough? 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.

3. You are a visitor at a political convention (or perhaps a faculty meeting) with n delegates. Each delegate is a member of exactly one political party. It is impossible to tell which political party any delegate belongs to. In particular, you will be summarily ejected from the convention if you ask. However, you *can* determine whether any pair of delegates belong to the *same* party or not simply by introducing them to each other. Members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.

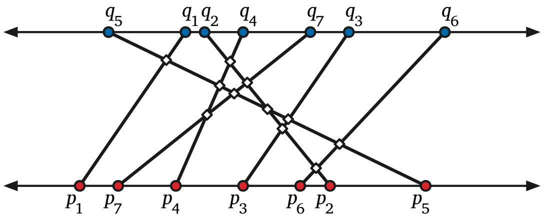
   (a) Suppose more than half of the delegates belong to the same political party. Describe and analyze an efficient algorithm that identifies every member of this majority party.

   (b) Now suppose precisely p political parties are present and one party has a plurality: more delegates belong to that party than to any other party. Please present a procedure to pick out the people from the plurality party as parsimoniously as possible.□ Do *not* assume that $p = O(1)$.

□Describe and analyze an efficient algorithm that identifies every member of the plurality party.

Solved Problem

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



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

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

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

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

- Recursively count inversions in (and sort) $Q[1\,..\,\lfloor n/2 \rfloor]$.
- Recursively count inversions in (and sort) $Q[\lfloor n/2 \rfloor + 1\,..\,n]$.
- Count inversions $Q[i] > Q[j]$ where $i \le \lfloor n/2 \rfloor$ and $j > \lfloor n/2 \rfloor$ as follows:
  - Color the elements in the Left half $Q[1\,..\,n/2]$ bLue.
  - Color the elements in the Right half $Q[n/2 + 1\,..\,n]$ Red.
  - Merge $Q[1\,..\,n/2]$ and $Q[n/2 + 1\,..\,n]$, maintaining their colors.
  - For each blue element $Q[i]$, 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 $total$
```

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

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

We can further optimize this algorithm by observing that *count* is always equal to $j - m - 1$. (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 $j > n$
          $B[k] \gets A[i]$; $i \gets i+1$; $total \gets total + \mathbf{j - m - 1}$
      else if $i > m$
          $B[k] \gets A[j]$; $j \gets j+1$
      else if $A[i] < A[j]$
          $B[k] \gets A[i]$; $i \gets i+1$; $total \gets total + \mathbf{j - m - 1}$
      else
          $B[k] \gets A[j]$; $j \gets j+1$
  for $k \gets 1$ to $n$
      $A[k] \gets B[k]$
  return $total$
```

The modified MERGE algorithm still runs in $O(n)$ time, so the running time of the resulting modified mergesort still obeys the recurrence $T(n) = 2T(n/2) + O(n)$. We conclude that the overall running time is $O(n \log n)$, as required. ■

Rubric: 10 points = 2 for base case + 3 for divide (split and recurse) + 3 for conquer (merge and count) + 2 for time analysis. Max 3 points for a correct $O(n^2)$-time algorithm. This is neither the only way to correctly describe this algorithm nor the only correct $O(n \log n)$-time algorithm. No proof of correctness is required.

CS/ECE 374 ✧ Fall 2016

Homework 5

Due Tuesday, October 11, 2016 at 8pm

---

1. For each of the following problems, the input consists of two arrays $X[1..k]$ and $Y[1..n]$ where $k \leq n$.

   (a) Describe and analyze an algorithm to determine whether X occurs as two ***disjoint*** subsequences of Y, where "disjoint" means the two subsequences have no indices in common. For example, the string PPAP appears as two disjoint subsequences in the string PENPINEAPPLEAPPLEPEN, but the string PEEPLE does not.

   (b) Describe and analyze an algorithm to compute the number of occurrences of X as a subsequence of Y. For example, the string PPAP appears exactly 23 times as a subsequence of the string PENPINEAPPLEAPPLEPEN. If all characters in X and Y are equal, your algorithm should return $\binom{n}{k}$. For purposes of analysis, assume that each arithmetic operation takes $O(1)$ time.

2. You are driving a bus along a long straight highway, full of rowdy, hyper, thirsty students and an endless supply of soda. Each minute that each student is on your bus, that student drinks one ounce of soda. Your goal is to drive all students home, so that the total volume of soda consumed by the students is as small as possible.

   Your bus begins at an exit (probably not at either end) with all students on board and moves at a constant speed of 37.4 miles per hour. Each student needs to be dropped off at a highway exit. You may reverse directions as often as you like; for example, you are allowed to drive forward to the next exit, let some students off, then turn around and drive back to the previous exit, drop more students off, then turn around again and drive to further exits. (Assume that at each exit, you can stop the bus, drop off students, and if necessary turn around, all instantaneously.)

   Describe an efficient algorithm to take the students home so that they drink as little soda as possible. Your algorithm will be given the following input:

   - A sorted array $L[1..n]$, where $L[i]$ is the Location of the ith exit, measured in miles from the first exit; in particular, $L[1] = 0$.
   - An array $N[1..n]$, where $N[i]$ is the Number of students you need to drop off at the ith exit
   - An integer *start* equal to the index of the starting exit.

   Your algorithm should return the total volume of soda consumed by the students when you drive the optimal route.□

□Non-US students are welcome to assume kilometers and liters instead of miles and ounces. Late 18th-century French students are welcome to use decimal minutes.

3. *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 below, the player can score $8-6+7-3+4 = 10$ points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. (This is *not* the best possible score for these values.)

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

   (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) In the Canadian version of this game, appropriately called *Vankin's Kilometer*, the player can move the token either one square down, one square right, *or one square left* in each turn. However, to prevent infinite scores, the token cannot land on the same square more than once. Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin's Kilometer, given the $n \times n$ array of values as input.□

□If we also allowed upward movement, the resulting game (Vankin's Fathom?) would be NP-hard.

Solved Problem

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

$${}^{\texttt{BANANA}}{}_{\texttt{ANANAS}} \qquad {}^{\texttt{BAN}}{}_{\texttt{ANA}}{}^{\texttt{ANA}}{}_{\texttt{NAS}} \qquad {}^{\texttt{B}}{}_{\texttt{AN}}{}^{\texttt{AN}}{}_{\texttt{A}}{}^{\texttt{A}}{}_{\texttt{NA}}{}^{\texttt{NA}}{}_{\texttt{S}}$$

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

$${}_{\texttt{PRO}}{}^{\texttt{D}}{}_{\texttt{G}}{}^{\texttt{Y}}{}_{\texttt{R}}{}^{\texttt{NAM}}{}_{\texttt{AMMI}}{}^{\texttt{I}}{}_{\texttt{N}}{}^{\texttt{C}}{}_{\texttt{G}} \qquad {}^{\texttt{DY}}{}_{\texttt{PRO}}{}^{\texttt{N}}{}_{\texttt{G}}{}^{\texttt{A}}{}_{\texttt{R}}{}^{\texttt{M}}{}_{\texttt{AMM}}{}^{\texttt{IC}}{}_{\texttt{ING}}$$

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

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

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

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

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

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

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

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

Standard dynamic programming rubric. For problems worth 10 poins:

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 point for a clear **English** description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) **Automatic zero if the English description is missing.**
  - + 1 point for stating how to call your function to get the final answer.
  - + 1 point for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 points for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 4 points for details of the dynamic programming algorithm
  - + 1 point for describing the memoization data structure
  - + 2 points for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested loops, be sure to specify the nesting order.
  - + 1 point for time analysis
- It is *not* necessary to state a space bound.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem says otherwise.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative psuedocode 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 still need to describe the underlying recursive function in English.)
- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we have included them, significantly more students turned in algorithms that meet the target time bound but didn't work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).

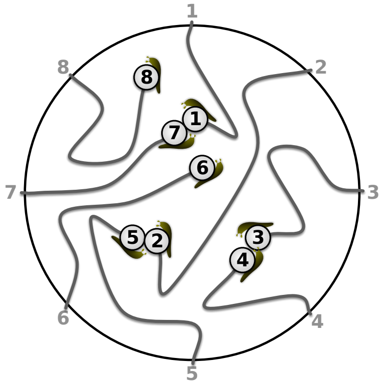
CS/ECE 374 ✧ Fall 2016

Homework 6

Due Tuesday, October 16, 2016 at 8pm

---

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. 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.

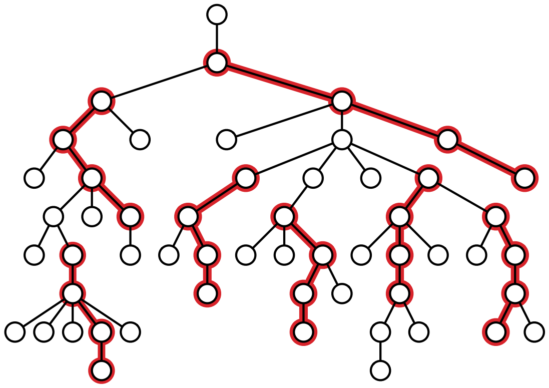
   (c) Five years later, Elmo has become a *significantly* stronger player. Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against a *perfect* opponent. *[Hint: What is a perfect opponent?]*

3. One day, Alex got tired of climbing in a gym and decided to take a very large group of climber friends outside to climb. The climbing area where they went, had a huge wide boulder, not very tall, with various marked hand and foot holds. Alex quickly determined an "allowed" set of moves that her group of friends can perform to get from one hold to another.

   The overall system of holds can be described by a rooted tree T with n vertices, where each vertex corresponds to a hold and each edge corresponds to an allowed move between holds. The climbing paths converge as they go up the boulder, leading to a unique hold at the summit, represented by the root of T.□

   Alex and her friends (who are all excellent climbers) decided to play a game, where as many climbers as possible are simultaneously on the boulder and each climber needs to perform a sequence of *exactly* k moves. Each climber can choose an arbitrary hold to start from, and all moves must move away from the ground. Thus, each climber traces out a path of k edges in the tree T, all directed toward the root. However, no two climbers are allowed to touch the same hold; the paths followed by different climbers cannot intersect at all.

   Describe and analyze an efficient algorithm to compute the maximum number of climbers that can play this game. More formally, you are given a rooted tree T and an integer k, and you want to find the largest possible number of disjoint paths in T, where each path has length k. For full credit, do ***not*** assume that T is a binary tree. For example, given the tree T below and $k = 3$ as input, your algorithm should return the integer 8.



Seven disjoint paths of length k=3 in a rooted tree.
This is *not* the largest such set of paths in this tree.

□Q: Why do computer science professors think trees have their roots at the top?
A: Because they've never been outside!

Solved Problems

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

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

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

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

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

Solution: Suppose $A[1\,..\,n]$ is the input string. For all indices i and j, we write $\boldsymbol{A[i] \sim A[j]}$ to indicate that $A[i]$ and $A[j]$ are matching delimiters: Either $A[i] = ($ and $A[j] =)$ or $A[i] = [$ and $A[j] =]$.

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

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

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

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

■

Rubric: 10 points, standard dynamic programming rubric

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

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

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

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

We need to compute $\mathit{MaxFunYes}(root)$. 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 $v.yes$ and $v.no$ to each node v in the tree. The values at each node depend only on the vales at its children, so we can compute all $2n$ values using a postorder traversal of T.

```
BESTPARTY(T):
    COMPUTEMAXFUN(T.root)
    return T.root.yes
```

```
COMPUTEMAXFUN(v):
    $v.yes \leftarrow v.fun$
    $v.no \leftarrow 0$
    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![1]) The algorithm spends $O(1)$ time at each node, and therefore runs in $\boldsymbol{O(n)}$ ***time*** altogether. ■

[1] 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 $\mathit{MaxFun}(v)$ 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:

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

(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!□)

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

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

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

CS/ECE 374 ✧ Fall 2016

Homework 7

Due Tuesday, October 25, 2016 at 8pm

If you use a greedy algorithm, you ***must*** prove that it is correct, or you will get zero points ***even if your algorithm is correct***.

---

1. You've been hired to store a sequence of n books on shelves in a library. The order of the books is fixed by the cataloging system and cannot be changed; each shelf must store a contiguous interval of the given sequence of books. You are given two arrays $H[1\,..\,n]$ and $T[1\,..\,n]$, where $H[i]$ and $T[i]$ are respectively the height and thickness of the ith book in the sequence. All shelves in this library have the same length L; the total thickness of all books on any single shelf cannot exceed L.

   (a) Suppose all the books have the same height h (that is, $H[i] = h$ for all i) and the shelves have height larger than h, so each book fits on every shelf. Describe and analyze a greedy algorithm to store the books in as few shelves as possible. *[Hint: The algorithm is obvious, but why is it correct?]*

   (b) That was a nice warmup, but now here's the real problem. In fact the books have different heights, but you can adjust the height of each shelf to match the tallest book on that shelf. (In particular, you can change the height of any empty shelf to zero.) Now your task is to store the books so that the sum of the heights of the shelves is as small as possible. Show that your greedy algorithm from part (a) does *not* always give the best solution to this problem.

   (c) Describe and analyze an algorithm to find the best assignment of books to shelves as described in part (b).

2. Consider a directed graph G, where each edge is colored either red, white, or blue. A walk□ in G is called a *French flag walk* if its sequence of edge colors is red, white, blue, red, white, blue, and so on. More formally, a walk $v_0 \to v_1 \to \cdots \to v_k$ is a French flag walk if, for every integer i, the edge $v_i \to v_{i+1}$ is red if $i \bmod 3 = 0$, white if $i \bmod 3 = 1$, and blue if $i \bmod 3 = 2$.

   Describe an efficient algorithm to find all vertices in a given edge-colored directed graph G that can be reached from a given vertex v through a French flag walk.

□Recall that a ***walk*** in a directed graph G is a sequence of vertices $v_0 \to v_1 \to \cdots \to v_k$, such that $v_{i-1} \to v_i$ is an edge in G for every index i. A ***path*** is a walk in which no vertex appears more than once.

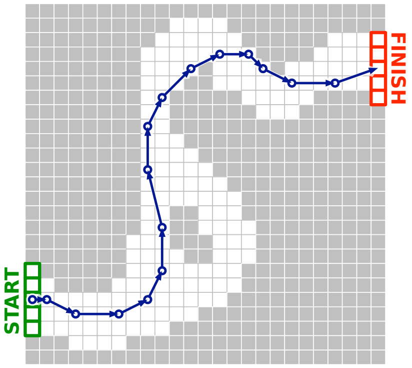
3. ***Racetrack*** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.[1] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion 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.

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 (for the sequence of operations listed above).

   Solution: Let A, B, C denote the capacities of the three jars. We reduce the problem to breadth-first search in the following directed graph:

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

- $\begin{Bmatrix} (a,0,b+c) & \text{if } b+c \le C \\ (a,b+c-C,C) & \text{if } b+c \ge C \end{Bmatrix}$ — pouring from the second jar into the third
- $\begin{Bmatrix} (a+c,b,0) & \text{if } a+c \le A \\ (A,b,a+c-A) & \text{if } a+c \ge A \end{Bmatrix}$ — pouring from the third jar into the first
- $\begin{Bmatrix} (a,b+c,0) & \text{if } b+c \le B \\ (a,B,b+c-B) & \text{if } b+c \ge B \end{Bmatrix}$ — pouring from the third jar into the second

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

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

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

Rubric (for graph reduction problems): 10 points:

- 2 for correct vertices
- 2 for correct edges
  - −½ for forgetting "directed"
- 2 for stating the correct problem (shortest paths)
  - "Breadth-first search" is not a problem; it's an algorithm.
- 2 points for correctly applying the correct algorithm (breadth-first search)
  - −1 for using Dijkstra instead of BFS
- 2 points for time analysis in terms of the input parameters.
- Max 8 points for $O(ABC)$ time; scale partial credit

CS/ECE 374 ☠ Fall 2016

Homework 8

Due Tuesday, November 1, 2016 at 8pm

This is the last homework before Midterm 2.

---

1. After a grueling algorithms midterm, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Champaign-Urbana. Champaign-Urbana is currently suffering from a plague of zombies, so even though the bus stops have fences that *supposedly* keep the zombies out, you'd still like to spend as little time waiting at bus stops as possible. Unfortunately, there isn't a single bus that visits both your exam building and your home; you must transfer between buses at least once.

   Describe and analyze an algorithm to determine a sequence of bus rides from Siebel to your home, that minimizes the total time you spend waiting at bus stops. You can assume that there are b different bus lines, and each bus stops n times per day. Assume that the buses run exactly on schedule, that you have an accurate watch, and that walking between bus stops is too dangerous to even contemplate.

2. Kris is a professional rock climber (friends with Alex and the rest of the climbing crew from HW6) who is competing in the U.S. climbing nationals. The competition requires Kris to use as many holds on the climbing wall as possible, using only transitions that have been explicitly allowed by the route-setter.

   The climbing wall has n holds. Kris is given a list of m pairs (x, y) of holds, each indicating that moving directly from hold x to hold y is allowed; however, moving directly from y to x is not allowed unless the list also includes the pair (y, x). Kris needs to figure out a sequence of allowed transitions that uses as many holds as possible, since each new hold increases his score by one point. The rules allow Kris to choose the first and last hold in his climbing route. The rules also allow him to use each hold as many times as he likes; however, only the first use of each hold increases Kris's score.

   (a) Define the natural graph representing the input. Describe and analyze an algorithm to solve Kris's climbing problem if you are guaranteed that the input graph is a dag.

   (b) Describe and analyze an algorithm to solve Kris's climbing problem with no restrictions on the input graph.

   Both of your algorithms should output the maximum possible score that Kris can earn.

3. Many years later, in a land far, far away, after winning all the U.S. national competitions for 10 years in a row, Kris retired from competitive climbing and became a route setter for competitions. However, as the years passed, the rules changed. Climbers are now required to climb along the *shortest* sequence of legal moves from one specific node to another, where the distance between two holds is specified by the route setter. In addition to the usual set of n holds and m valid moves between them (as in the previous problem), climbers are now told their start hold s, their finish hold t, and the distance from x to y for every allowed move (x, y).

   Rather than make up this year's new route completely from scratch, Kris decides to make one small change to last year's input. The previous route setter suggested a list of k new allowed moves and their distances. Kris needs to to choose the single edge from this list of suggestions that decreases the distance from s to t as much as possible.

   Describe and analyze an algorithm to solve Kris's problem. Your input consists of the following information:

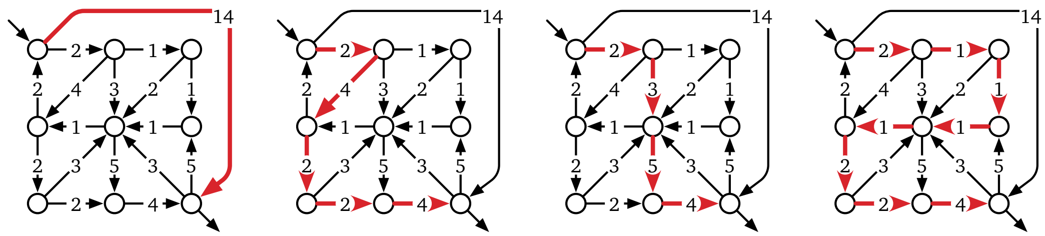
   - A directed graph $G = (V, E)$.
   - Two vertices $s, t \in V$.
   - A set of k new edges E', such that $E \cap E' = \varnothing$
   - A length $\ell(e) \geq 0$ for every edge $e \in E \cup E'$.

   Your algorithm should return the edge $e \in E'$ whose addition to the graph yields the smallest shortestopath distance from s to t.

   For full credit, your algorithm should run in $O(m \log n + k)$ time, but as always, a slower correct algorithm is worth more than a faster incorrect algorithm.

Solved Problem

4. Although we typically speak of "the" shortest path between two nodes, 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 determine the *number* of shortest paths from a source vertex s to a target vertex t in an arbitrary directed graph G with weighted edges. You may assume that all edge weights are positive and that all necessary arithmetic operations can be performed in $O(1)$ time.

[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 $PathsToT(v)$ denote the number of paths in H from v to t; we need to compute $PathsToT(s)$. This function satisfies the following simple recurrence:

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

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 $PathsToT(v) = \sum\varnothing = 0$.

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

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

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

CS/ECE 374 ✧ Fall 2016

Homework 9

Due Tuesday, November 15, 2016 at 8pm

---

1. Consider the following problem, called BoxDepth: Given a set of n axis-aligned rectangles in the plane, how big is the largest subset of these rectangles that contain a common point?

   (a) Describe a polynomial-time reduction from BoxDepth to MaxClique, and prove that your reduction is correct.

   (b) Describe and analyze a polynomial-time algorithm for BoxDepth. *[Hint: Don't try to optimize the running time; $O(n^3)$ is good enough.]*

   (c) Why don't these two results imply that P=NP?

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

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

   (a) Describe a polynomial-time reduction from SubsetSum to Partition.

   (b) Describe a polynomial-time reduction from Partition to SubsetSum.

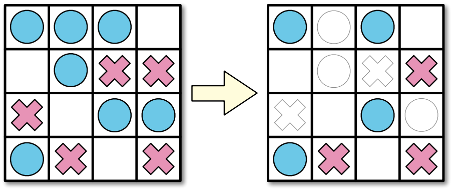
   Don't forget to to prove that your reductions are correct.

3. Suppose you are given a graph $G = (V, E)$ where V represents a collection of people and an edge between two people indicates that they are friends. You wish to partition V into at most k non-overlapping groups $V_1, V_2, \ldots, V_k$ such that each group is very cohesive. One way to model cohesiveness is to insist that each pair of people in the same group should be friends; in other words, they should form a clique.

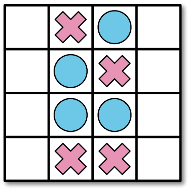
   Prove that the following problem is NP-hard: Given an undirected graph G and an integer k, decide whether the vertices of G can be partitioned into k cliques.

Solved Problem

4. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

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

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

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

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

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

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

- If $x_j =$ True, remove all red stones from column j.
- If $x_j =$ False, remove all blue stones from column j.

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

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

- If column j contains blue stones, set $x_j =$ True.
- If column j contains red stones, set $x_j =$ 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 (for all polynomial-time reductions): 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).
- + 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 ✧ Fall 2016

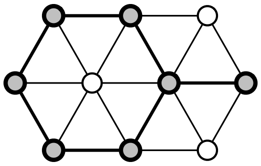
Homework 10

Due Tuesday, November 29, 2016 at 8pm

---

1. A subset S of vertices in an undirected graph G is called ***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.

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



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

3. Charon needs to ferry n recently deceased people across the river Acheron into Hades. Certain pairs of these people are sworn enemies, who cannot be together on either side of the river unless Charon is also present. (If two enemies are left alone, one will steal the obol from the other's mouth, leaving them to wander the banks of the Acheron as a ghost for all eternity. Let's just say this is a Very Bad Thing.) The ferry can hold at most k passengers at a time, including Charon, and only Charon can pilot the ferry.

   Prove that it is NP-hard to decide whether Charon can ferry all n people across the Acheron unharmed.[□] The input for Charon's problem consists of the integers k and n and an n-vertex graph G describing the pairs of enemies. The output is either True or False.

[□] Aside from being, you know, dead.

Problem 3 is a generalization of the following extremely well-known puzzle, whose first known appearance is in the treatise *Propositiones ad Acuendos Juvenes* [*Problems to Sharpen the Young*] by the 8th-century English scholar Alcuin of York.□

> XVIII. Propositio De Homine et Capra et Lvpo.
>
> Homo quidam debebat ultra fluuium transferre lupum, capram, et fasciculum cauli. Et non potuit aliam nauem inuenire, nisi quae duos tantum ex ipsis ferre ualebat. Praeceptum itaque ei fuerat, ut omnia haec ultra illaesa omnino transferret. Dicat, qui potest, quomodo eis illaesis transire potuit?
>
> **Solutio.** Simili namque tenore ducerem prius capram et dimitterem foris lupum et caulum. Tum deinde uenirem, lupumque transferrem: lupoque foris misso capram naui receptam ultra reducerem; capramque foris missam caulum transueherem ultra; atque iterum remigassem, capramque assumptam ultra duxissem. Sicque faciendo facta erit remigatio salubris, absque uoragine lacerationis.

In case your classical Latin is rusty, here is an English translation:

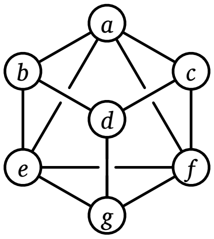
> XVIII. The Problem of the Man, the Goat, and the Wolf.
>
> A man needed to transfer a wolf, a goat, and a bundle of cabbage across a river. However, he found that his boat could only bear the weight of two [objects at a time, including the man]. And he had to get everything across unharmed. Tell me if you can: How they were able to cross unharmed?
>
> **Solution.** In a similar fashion [as an earlier problem], I would first take the goat across and leave the wolf and cabbage on the opposite bank. Then I would take the wolf across; leaving the wolf on shore, I would retrieve the goat and bring it back again. Then I would leave the goat and take the cabbage across. And then I would row across again and get the goat. In this way the crossing would go well, without any threat of slaughter.

Please do not write your solution to problem 3 in classical Latin.

□At least, we *think* that's who wrote it; the evidence for his authorship is rather circumstantial, although we do know from his correspondence with Charlemagne that he sent the emperor some "simple arithmetical problems for fun". Most scholars believe that even if Alcuin is the actual author of the *Propositiones*, he didn't come up with the problems himself, but just collected his problems from other sources. Some things never change.

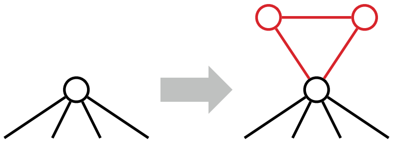
Solved Problem

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



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

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



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

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

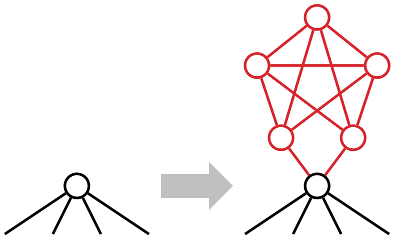
$\Longrightarrow$ Suppose G has a Hamiltonian cycle $v_1 \to v_2 \to \cdots \to v_n \to v_1$. We can construct a double-Hamiltonian tour of H by replacing each vertex v_i with the following walk:

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

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

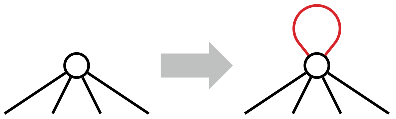
Given any graph G, we can clearly construct the corresponding graph H in polynomial time.

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



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

Common incorrect 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.

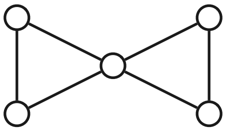


An incorrect vertex gadget.

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

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

On the other hand, 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.

✧

Rubric (for all polynomial-time reductions): 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).
+ 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 ✧ Fall 2016

Homework 11

"Due" Tuesday, December, 2016

This homework is only for practice; it will not be graded. However, **similar questions may appear on the final exam,** so we still strongly recommend treating this as a regular homework. Solutions will be released next Tuesday as usual.

---

1. Recall that w^R denotes the reversal of string w; for example, $\texttt{TURING}^R = \texttt{GNIRUT}$. Prove that the following language is undecidable.

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

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

2. Let M be a Turing machine, let w be an arbitrary input string, and let s be an integer. We say that ***M accepts w in space s*** if, given w as input, M accesses only the first s (or fewer) cells on its tape and eventually accepts.

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

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

   (b) Prove that the following language is undecidable:

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

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

   (a) Prove that $\textsc{SometimesHalt}$ is undecidable.

   (b) Sketch a Turing machine/algorithm that *accepts* $\textsc{SometimesHalt}$.

Solved Problem

4. For each of the following languages, either prove that the language is decidable, or prove that the language is undecidable.

   (a) $L_0 = \big\{\langle M\rangle \bigm| \text{given any input string, } M \text{ eventually leaves its } \mathsf{start} \text{ state}\big\}$

   Solution: We can determine whether a given Turing machine M always leaves its start state by careful analysis of its transition function δ. As a technical point, I will assume that crashing on the first transition does *not* count as leaving the start state.

   - If $\delta(\mathsf{start}, a) = (\cdot, \cdot, -1)$ for any input symbol $a \in \Sigma$, then M crashes on input a without leaving the start state.
   - If $\delta(\mathsf{start}, \square) = (\cdot, \cdot, -1)$, then M crashes on the empty input without leaving the start state.
   - Otherwise, M moves to the right until it leaves the start state. There are two subcases to consider:
     - If $\delta(\mathsf{start}, \square) = (\mathsf{start}, \cdot, +1)$, then M loops forever on the empty input without leaving the start state.
     - Otherwise, for any input string, M must eventually leave the start state, either when reading some input symbol or when reading the first blank.

   It is straightforward (but tedious) to perform this case analysis with a Turing machine that receives the encoding $\langle M\rangle$ as input. We conclude that L_0 is ***decidable***. ■

   (b) $L_1 = \big\{\langle M\rangle \bigm| M \text{ decides } L_0\big\}$

   Solution:

   - By part (a), there is a Turing machine that decides L_0.
   - Let M_{reject} be a Turing machine that immediately rejects its input, by defining $\delta(\mathsf{start}, a) = \mathsf{reject}$ for all $a \in \Sigma \cup \{\square\}$. Then M_{reject} decides the language $\varnothing \neq L_0$. ■

   Thus, Rice's Decision Theorem implies that L_1 is ***undecidable***.

   (c) $L_2 = \big\{\langle M\rangle \bigm| M \text{ decides } L_1\big\}$

   Solution: By part (b), no Turing machine decides L_1, which implies that $L_2 = \varnothing$. Thus, M_{reject} correctly decides L_2. We conclude that L_2 is ***decidable***. ■

   (d) $L_3 = \big\{\langle M\rangle \bigm| M \text{ decides } L_2\big\}$

   Solution: Because $L_2 = \varnothing$, we have

   $$L_3 = \big\{\langle M\rangle \bigm| M \text{ decides } \varnothing\big\} = \big\{\langle M\rangle \bigm| \textsc{Reject}(M) = \Sigma^*\big\}$$

   - We have already seen a Turing machine M_{reject} such that $\textsc{Reject}(M_{\mathsf{reject}}) = \Sigma^*$.
   - Let M_{accept} be a Turing machine that immediately accepts its input, by defining $\delta(\mathsf{start}, a) = \mathsf{accept}$ for all $a \in \Sigma \cup \{\square\}$. Then $\textsc{Reject}(M_{\mathsf{accept}}) = \varnothing \neq \Sigma^*$. ■

   Thus, Rice's Rejection Theorem implies that L_1 is ***undecidable***.

(e) $L_4 = \left\{ \langle M \rangle \;\middle|\; M \text{ decides } L_3 \right\}$

Solution: By part (b), no Turing machine decides L_3, which implies that $L_4 = \varnothing$. Thus, M_{reject} correctly decides L_4. We conclude that L_4 is ***decidable***.

At this point, we have fallen into a loop. For any $k > 4$, define

$$L_k = \left\{ \langle M \rangle \;\middle|\; M \text{ decides } L_{k-1} \right\}.$$

Then L_k is decidable (because $L_k = \varnothing$) if and only if k is even. ■

Rubric: 10 points: 4 for part (a) + 1½ for each other part.

Rubric (for all undecidability proofs, out of 10 points):
Diagonalization:

- + 4 for correct wrapper Turing machine
- + 6 for self-contradiction proof (= 3 for ⇐ + 3 for ⇒)

Reduction:

- + 4 for correct reduction
- + 3 for "if" proof
- + 3 for "only if" proof

Rice's Theorem:

- + 4 for positive Turing machine
- + 4 for negative Turing machine
- + 2 for other details (including using the correct variant of Rice's Theorem)

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: You need #2 to prove #3, but you may find it easier to solve #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 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 substring 000 appears after every 1.
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 $\Sigma = \{0, 1\}$. Describe briefly what each state in your DFAs *means*.

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 to keep the number of states small.

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 no substring `000` appears before a `1`.

5. All strings containing at least three `0`s.

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 `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 $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. 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 be clear. Try to keep the number of states small.

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

2. 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.

Work on these later:

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

*4. All strings w such that $F_{\#(\mathtt{10},w)} \bmod 10 = 4$, where $\#(\mathtt{10}, w)$ 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.

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.

Work on these later:

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

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

Let L be an arbitrary regular language.

1. Prove that the language $\mathit{insert}\mathtt{1}(L) := \{x\mathtt{1}y \mid xy \in L\}$ is regular.

   Intuitively, $\mathit{insert}\mathtt{1}(L)$ is the set of all strings that can be obtained from strings in L by inserting exactly one $\mathtt{1}$. For example, if $L = \{\varepsilon, \mathtt{OOK!}\}$, then $\mathit{insert}\mathtt{1}(L) = \{\mathtt{1}, \mathtt{1OOK!}, \mathtt{O1OK!}, \mathtt{OO1K!}, \mathtt{OOK1!}, \mathtt{OOK!1}\}$.

2. Prove that the language $\mathit{delete}\mathtt{1}(L) := \{xy \mid x\mathtt{1}y \in L\}$ is regular.

   Intuitively, $\mathit{delete}\mathtt{1}(L)$ is the set of all strings that can be obtained from strings in L by deleting exactly one $\mathtt{1}$. For example, if $L = \{\mathtt{101101}, \mathtt{00}, \varepsilon\}$, then $\mathit{delete}\mathtt{1}(L) = \{\mathtt{01101}, \mathtt{10101}, \mathtt{10110}\}$.

Work on these later: (In fact, these might be easier than problems 1 and 2.)

3. Consider the following recursively defined function on strings:

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

   Intuitively, $\mathit{stutter}(w)$ doubles every symbol in w. For example:

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

   Let L be an arbitrary regular language.

   (a) Prove that the language $\mathit{stutter}^{-1}(L) := \{w \mid \mathit{stutter}(w) \in L\}$ is regular.

   (b) Prove that the language $\mathit{stutter}(L) := \{\mathit{stutter}(w) \mid w \in L\}$ is regular.

4. Consider the following recursively defined function on strings:

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

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

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

   Once again, let L be an arbitrary regular language.

   (a) Prove that the language $\mathit{evens}^{-1}(L) := \{w \mid \mathit{evens}(w) \in L\}$ is regular.

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

Alex showed the following context-free grammars in class on Tuesday; 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.

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

  Here is a different grammar for the same language:

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

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

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

Give context-free grammars for each of the following languages. For each grammar, describe *in English* the language for each non-terminal, and in the examples above. As usual, we won't get to all of these in section.

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

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

   [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.]

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

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

Work on these later:

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

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

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

For each of the following languages over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, either prove the language is regular (by giving an equivalent regular expression, DFA, or NFA) or prove that the language is not regular (using a fooling set argument). Exactly half of these languages are regular.

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

2. $\{\mathtt{0}^n\mathtt{1}\mathtt{0}^n w \mid n \geq 0 \text{ and } w \in \Sigma^*\}$

3. $\{w\mathtt{0}^n\mathtt{1}\mathtt{0}^n x \mid w \in \Sigma^* \text{ and } n \geq 0 \text{ and } x \in \Sigma^*\}$

4. Strings in which the number of 0s and the number of 1s differ by at most 2.

5. Strings such that *in every prefix*, the number of 0s and the number of 1s differ by at most 2.

6. Strings such that *in every substring*, the number of 0s and the number of 1s differ by at most 2.

Design Turing machines $M = (Q, \Sigma, \Gamma, \delta, \mathsf{start}, \mathsf{accept}, \mathsf{reject})$ for each of the following tasks, either by listing the states Q, the tape alphabet Γ, and the transition function δ (in a table), or by drawing the corresponding labeled graph.

Each of these machines uses the input alphabet $\Sigma = \{\mathtt{1}, \mathtt{\#}\}$; the tape alphabet Γ can be any superset of $\{\mathtt{1}, \mathtt{\#}, \square, \triangleright\}$ where $\square$ is the blank symbol and $\triangleright$ is a special symbol marking the left end of the tape. Each machine should reject any input not in the form specified below.

---

1. On input $\mathtt{1}^n$, for any non-negative integer n, write $\mathtt{1}^n\mathtt{\#}\mathtt{1}^n$ on the tape and accept.

2. On input $\mathtt{\#}^n\mathtt{1}^m$, for any non-negative integers m and n, write $\mathtt{1}^m$ on the tape and accept. In other words, delete all the #s and shift the 1s to the start of the tape.

3. On input $\mathtt{\#}\mathtt{1}^n$, for any non-negative integer n, write $\mathtt{\#}\mathtt{1}^{2n}$ on the tape and accept. *[Hint: Modify the Turing machine from problem 1.]*

4. On input $\mathtt{1}^n$, for any non-negative integer n, write $\mathtt{1}^{2^n}$ on the tape and accept. *[Hint: Use the three previous Turing machines as subroutines.]*

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?]*

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, Alex described an algorithm of 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 some extensions and applications of this algorithm.

1. Describe an algorithm to compute the product of an n-digit number and an m-digit number, where $m < n$, in $O(m^{\lg 3 - 1} n)$ time.

2. Describe an algorithm to compute the decimal representation of 2^n in $O(n^{\lg 3})$ time. (The standard algorithm that computes one digit at a time requires $\Theta(n^2)$ time.)

3. Describe a divide-and-conquer algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(n^{\lg 3})$ time. *[Hint: Let $x = a \cdot 2^{n/2} + b$. Watch out for an extra log factor in the running time.]*

Think about later:

4. Suppose we can multiply two n-digit numbers in $O(M(n))$ time. Describe an algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(M(n) \log n)$ 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 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

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

   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

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

   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

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

   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).

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

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

   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

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

   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 the string `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 problems. For the first three, use the backtracking algorithms 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 is an automatic zero.

   (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.

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.

Lenny Rutenbar, the founding dean of the new Maximilian Q. Levchin College of Computer Science, has commissioned a series of snow ramps on the south slope of the Orchard Downs sledding hill[*] and challenged Bill Kudeki, head of the Department of Electrical and Computer Engineering, to a sledding contest. Bill and Lenny will both sled down the hill, each trying to maximize their air time. The winner gets to expand their department/college into both Siebel Center and the new ECE Building; the loser has to move their entire department/college under the Boneyard bridge next to Everitt Lab.

Whenever Lenny or Bill 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 Lenny or Bill can spend in the air.

2. Uh-oh. The university lawyers heard about Lenny and Bill's little bet and immediately objected. To protect the university from either lawsuits or 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 Lenny or Bill 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.

3. **To think about later:** When the lawyers realized that imposing their restriction didn't immediately shut down the contest, they added a new restriction: No ramp can be used more than once! Disgusted by the legal interference, Lenny and Bill 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 Lenny and Bill 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.

[*] 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.

2. **To think about later:** 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 $1+3(-2)(-5)+1-6+7=33$.

Recall the class scheduling problem described in lecture on Tuesday. We are given two arrays $S[1..n]$ and $F[1..n]$, where $S[i] < F[i]$ for each i, representing the start and finish times of n classes. Your goal is to find the largest number of classes you can take without ever taking two classes simultaneously. We showed in class that the following greedy algorithm constructs an optimal schedule:

Choose the course that *ends first*, discard all conflicting classes, and recurse.

But this is not the only greedy strategy we could have tried. For each of the following alternative greedy algorithms, either prove that the algorithm always constructs an optimal schedule, or describe a small input example for which the algorithm does not produce an optimal schedule. Assume that all algorithms break ties arbitrarily (that is, in a manner that is completely out of your control). ***Exactly three of these greedy strategies actually work.***

1. Choose the course x that *ends last*, discard classes that conflict with x, and recurse.

2. Choose the course x that *starts first*, discard all classes that conflict with x, and recurse.

3. Choose the course x that *starts last*, discard all classes that conflict with x, and recurse.

4. Choose the course x with *shortest duration*, discard all classes that conflict with x, and recurse.

5. Choose a course x that *conflicts with the fewest other courses*, discard all classes that conflict with x, and recurse.

6. If no classes conflict, choose them all. Otherwise, discard the course with *longest duration* and recurse.

7. If no classes conflict, choose them all. Otherwise, discard a course that *conflicts with the most other courses* and recurse.

8. Let x be the class with the *earliest start time*, and let y be the class with the *second earliest start time*.

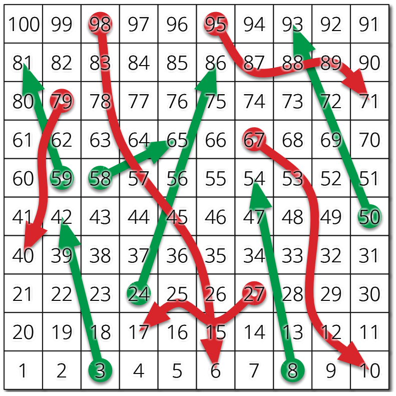
   - If x and y are disjoint, choose x and recurse on everything but x.
   - If x completely contains y, discard x and recurse.
   - Otherwise, discard y and recurse.

9. If any course x completely contains another course, discard x and recurse. Otherwise, choose the course y that *ends last*, discard all classes that conflict with y, and recurse.

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 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*?

1. ***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.

2. Let G be a connected 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 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).

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 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*?

---

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 any game, then A must rank better than B in the overall ranking.

   You are given the list of players and their ranking 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.

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 a new algorithm to compute the maximum number of distinct galaxies Judy can visit. She can 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 $u \to v$ has negative weight. Assume the graph has no negative cycles. *[Hint: Modify the input graph and run Dijkstra's algorithm. Alternatively,* **don't** *modify the input graph, but run Dijkstra's algorithm anyway.]*

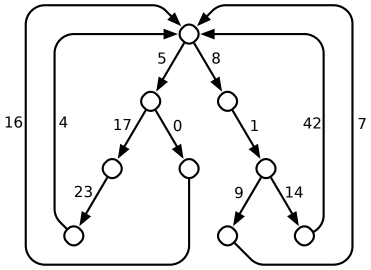
2. You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cites that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving 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 soon as possible, assuming both of you leave home *right now*.

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

   (a) Suppose $w(u\to v) > w'(u\to v)$; that is, the weight you used for $u\to v$ was *larger* than its true weight. Describe an algorithm that repairs the distance array in ***$O(V^2)$ time*** under this assumption. *[Hint: For every pair of vertices x and y, either $u\to v$ 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 ***$O(1)$ time***, again assuming that $w(u\to v) > w'(u\to v)$. *[Hint: Either $u\to v$ is the shortest path from u to v or it isn't.]*

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

   (d) **To think about later:** Argue that when $w(u\to v) < w'(u\to v)$, 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!□ 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.

□No, you can't.

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. 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.

   [Hint: You've seen this problem before.]

   (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.

---

1. Recall 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.

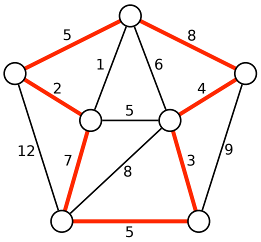
   (b) Prove that kColor problem is NP-hard for any $k \geq 3$.

2. 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.

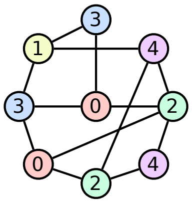
Prove that each of the following problems is NP-hard.

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

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

3. 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.]*



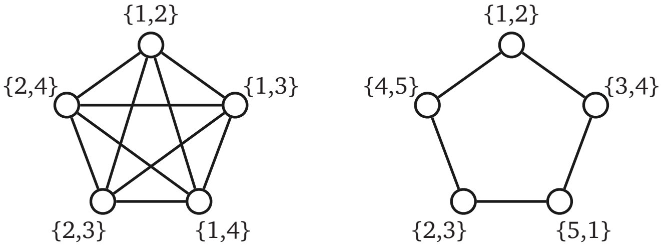
A careful 5-coloring.

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.

Proving that a language L is undecidable by reduction requires several steps. (These are the essentially the same steps you already use to prove that a problem is NP-hard.)

- Choose a language L' that you already know is undecidable (because we told you so in class). The simplest choice is usually the standard halting language

$$\text{Halt} := \{\langle M, w\rangle \mid M \text{ halts on } w\}$$

- Describe an algorithm that decides L', using an algorithm that decides L as a black box. Typically your reduction will have the following form:

  Given an arbitrary string x, construct a special string y,
  such that $y \in L$ if and only if $x \in L'$.

  In particular, if $L = \text{Halt}$, your reduction will have the following form:

  Given the encoding $\langle M, w\rangle$ of a Turing machine M and a string w,
  construct a special string y, such that
  $y \in L$ if and only if M halts on input w.

- Prove that your algorithm is correct. This proof almost always requires two separate steps:
  - Prove that if $x \in L'$ then $y \in L$.
  - Prove that if $x \notin L'$ then $y \notin L$.

Very important: Name every object in your proof, and *always* refer to objects by their names. Never refer to “the Turing machine” or “the algorithm” or “the input string” or (god forbid) “it” or “this”. Even in casual conversation, even if you’re “just” explaining your intuition, even when you’re just *thinking* about the reduction.

Prove that the following languages are undecidable.

1. $\text{AcceptIllini} := \{\langle M\rangle \mid M \text{ accepts the string } \texttt{ILLINI}\}$
2. $\text{AcceptThree} := \{\langle M\rangle \mid M \text{ accepts exactly three strings}\}$
3. $\text{AcceptPalindrome} := \{\langle M\rangle \mid M \text{ accepts at least one palindrome}\}$

Solution (for problem 1): For the sake of argument, suppose there is an algorithm DecideAcceptIllini that correctly decides the language AcceptIllini. Then we can solve the halting problem as follows:

```
DecideHalt(⟨M, w⟩):
    Encode the following Turing machine M′:
        M′(x):
            run M on input w
            return True
    if DecideAcceptIllini(⟨M′⟩)
        return True
    else
        return False
```

We prove this reduction correct as follows:

⟹ Suppose M halts on input w.

Then M' accepts *every* input string x.

In particular, M' accepts the string **ILLINI**.

So DecideAcceptIllini accepts the encoding $\langle M' \rangle$.

So DecideHalt correctly accepts the encoding $\langle M, w \rangle$.

⟸ Suppose M does not halt on input w.

Then M' diverges on *every* input string x.

In particular, M' does not accept the string **ILLINI**.

So DecideAcceptIllini rejects the encoding $\langle M' \rangle$.

So DecideHalt correctly rejects the encoding $\langle M, w \rangle$.

In both cases, DecideHalt is correct. But that's impossible, because Halt is undecidable. We conclude that the algorithm DecideAcceptIllini does not exist. ■

As usual for undecidablility proofs, this proof invokes *four* distinct Turing machines:

- The hypothetical algorithm DecideAcceptIllini.
- The new algorithm DecideHalt that we construct in the solution.
- The arbitrary machine M whose encoding is part of the input to DecideHalt.
- The special machine M' whose encoding DecideHalt constructs (from the encoding of M and w) and then passes to DecideAcceptIllini.

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

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

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

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

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

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, check "True" if the statement is ***always*** true and "False" otherwise. Each correct answer is worth +1 point; each incorrect answer is worth −½ point; checking "I don't know" is worth +¼ point; and flipping a coin is (on average) worth +¼ point. You do ***not*** need to prove your answer is correct.

   Read each statement *very* carefully. Some of these are deliberately subtle.

   (a) If the moon is made of cheese, then Jeff is the Queen of England.

   (b) The language $\{\mathtt{0}^m\mathtt{1}^n \mid m, n \geq 0\}$ is not regular.

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

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

   (e) For all languages L and L', if $L \cap L' = \varnothing$ and L' is not regular, then L is regular.

   (f) For all languages L, if L is not regular, then L does not have a finite fooling set.

   (g) Let $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q, s, Q \setminus A, \delta)$ be arbitrary **DFA**s with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then $L(M) \cap L(M') = \varnothing$.

   (h) Let $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q, s, Q \setminus A, \delta)$ be arbitrary **NFA**s with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then $L(M) \cap L(M') = \varnothing$.

   (i) For all context-free languages L and L', the language $L \bullet L'$ is also context-free.

   (j) Every non-context-free language is non-regular.

2. 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.***

   (a) $\left\{\mathtt{0}^n w \mathtt{0}^n \;\middle|\; w \in \Sigma^+ \text{ and } n > 0\right\}$

   (b) $\left\{w \mathtt{0}^n w \;\middle|\; w \in \Sigma^+ \text{ and } n > 0\right\}$

   For example, both of these languages contain the string `00110100000110100`.

3. Let $L = \{\mathtt{0}^{2i}\mathtt{1}^{i+2j}\mathtt{0}^{j} \mid i, j \geq 0\}$ and let G be the following context free-grammar:

$$S \rightarrow AB$$
$$A \rightarrow \varepsilon \mid \mathtt{00}A\mathtt{1}$$
$$B \rightarrow \varepsilon \mid \mathtt{11}B\mathtt{0}$$

   (a) ***Prove*** that $L(G) \subseteq L$.

   (b) ***Prove*** that $L \subseteq L(G)$.

   [Hint: What are $L(A)$ and $L(B)$? Prove it!]

4. For any language L, let $\textsc{Suffixes}(L) := \left\{x \;\middle|\; yx \in L \text{ for some } y \in \Sigma^*\right\}$ be the language containing all suffixes of all strings in L. For example, if $L = \{\mathtt{010}, \mathtt{101}, \mathtt{110}\}$, then $\textsc{Suffixes}(L) = \{\varepsilon, \mathtt{0}, \mathtt{1}, \mathtt{01}, \mathtt{10}, \mathtt{010}, \mathtt{101}, \mathtt{110}\}$.

   Prove that for any regular language L, the language $\textsc{Suffixes}(L)$ is also regular.

5. For each of the following languages L, give a regular expression that represents L ***and*** describe a DFA that recognizes L.

   (a) The set of all strings in $\{\mathtt{0}, \mathtt{1}\}^*$ that do not contain the substring $\mathtt{0110}$.

   (b) The set of all strings in $\{\mathtt{0}, \mathtt{1}\}^*$ that contain exactly one of the substrings $\mathtt{01}$ or $\mathtt{10}$.

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

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, check "True" if the statement is ***always*** true and "False" otherwise. Each correct answer is worth $+1$ point; each incorrect answer is worth $-\frac{1}{2}$ point; checking "I don't know" is worth $+\frac{1}{4}$ point; and flipping a coin is (on average) worth $+\frac{1}{4}$ point. You do ***not*** need to prove your answer is correct.

   Read each statement *very* carefully. Some of these are deliberately subtle.

   (a) If $2+2=5$, then Jeff is the Queen of England.

   (b) The language $\{0^m\#0^n \mid m,n \geq 0\}$ is regular.

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

   (d) For all languages $L \subset \Sigma^*$, if L cannot be recognized by a DFA, then $\Sigma^* \setminus L$ cannot be represented by a regular expression.

   (e) For all languages L and L', if $L \cap L' = \varnothing$ and L' is regular, then L is regular.

   (f) For all languages L, if L has a finite fooling set, then L is regular.

   (g) 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 $L(M) \cup L(M') = \Sigma^*$.

   (h) 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 $L(M) \cap L(M') = \varnothing$.

   (i) For all context-free languages L and L', the language $L \bullet L'$ is also context-free.

   (j) Every non-regular language is context-free.

2. For each of the following languages over the alphabet $\Sigma = \{0, 1\}$, either ***prove*** that the language is regular or ***prove*** that the language is not regular. ***Exactly one of these two languages is regular.***

   (a) $\left\{w0^nw \;\middle|\; w \in \Sigma^+ \text{ and } n > 0\right\}$

   (b) $\left\{0^nw0^n \;\middle|\; w \in \Sigma^+ \text{ and } n > 0\right\}$

   For example, both of these languages contain the string 00110100000110100.

3. Let $L = \{1^m0^n \mid m \leq n \leq 2m\}$ and let G be the following context free-grammar:

$$S \rightarrow 1S0 \mid 1S00 \mid \varepsilon$$

(a) ***Prove*** that $L(G) \subseteq L$.

(b) ***Prove*** that $L \subseteq L(G)$.

4. For any language L, let $\textsc{Prefixes}(L) := \left\{x \;\middle|\; xy \in L \text{ for some } y \in \Sigma^*\right\}$ be the language containing all prefixes of all strings in L. For example, if $L = \{000, 100, 110, 111\}$, then $\textsc{Prefixes}(L) = \{\varepsilon, 0, 1, 00, 10, 11, 000, 100, 110, 111\}$.

Prove that for any regular language L, the language $\textsc{Prefixes}(L)$ is also regular.

5. For each of the following languages L, give a regular expression that represents L ***and*** describe a DFA that recognizes L.

(a) The set of all strings in $\{0, 1\}^*$ that contain either both or neither of the substrings 01 and 10.

(b) The set of all strings in $\{0, 1\}^*$ that do not contain the substring 1001.

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

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, check "True" if the statement is ***always*** true and "False" otherwise. Each correct answer is worth $+1$ point; each incorrect answer is worth $-½$ point; checking "I don't know" is worth $+¼$ point; and flipping a coin is (on average) worth $+¼$ point. You do ***not*** need to prove your answer is correct.

   Read each statement *very* carefully. Some of these are deliberately subtle.

   (a) If 100 is a prime number, then Jeff is the Queen of England.

   (b) The language $\left\{\mathtt{0}^m\mathtt{0}^{n+m}\mathtt{0}^n \,\middle|\, m,n \geq 0\right\}$ is regular.

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

   (d) For all languages $L \subset \Sigma^*$, if L can be recognized by a DFA, then $\Sigma^* \setminus L$ cannot be represented by a regular expression.

   (e) For all languages L and L', if $L \subseteq L'$ and L' is regular, then L is regular.

   (f) For all languages L, if L has a finite fooling set, then L is not regular.

   (g) Let $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q, s, Q \setminus A, \delta)$ be arbitrary **NFA**s with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then $L(M) \cup L(M') = \Sigma^*$.

   (h) Let $M = (\Sigma, Q, s, A, \delta)$ and $M' = (\Sigma, Q, s, Q \setminus A, \delta)$ be arbitrary **NFA**s with identical alphabets, states, starting states, and transition functions, but with complementary accepting states. Then $L(M) \cap L(M') = \varnothing$.

   (i) For all context-free languages L and L', the language $L \bullet L'$ is also context-free.

   (j) Every regular language is context-free.

2. 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.***

   (a) $\left\{w\mathtt{0}^n w \,\middle|\, w \in \Sigma^+ \text{ and } n > 0\right\}$

   (b) $\left\{\mathtt{0}^n w \mathtt{0}^n \,\middle|\, w \in \Sigma^+ \text{ and } n > 0\right\}$

   For example, both of these languages contain the string $\mathtt{00110100000110100}$.

3. Let $L = \{1^m 0^n \mid n \leq m \leq 2n\}$ and let G be the following context free-grammar:

$$S \rightarrow 1S0 \mid 11S0 \mid \varepsilon$$

   (a) ***Prove*** that $L(G) \subseteq L$.

   (b) ***Prove*** that $L \subseteq L(G)$.

4. For any language L, let $\textsc{Prefixes}(L) := \left\{x \;\middle|\; xy \in L \text{ for some } y \in \Sigma^*\right\}$ be the language containing all prefixes of all strings in L. For example, if $L = \{000, 100, 110, 111\}$, then $\textsc{Prefixes}(L) = \{\varepsilon, 0, 1, 00, 10, 11, 000, 100, 110, 111\}$.

   Prove that for any regular language L, the language $\textsc{Prefixes}(L)$ is also regular.

5. For each of the following languages L, give a regular expression that represents L ***and*** describe a DFA that recognizes L.

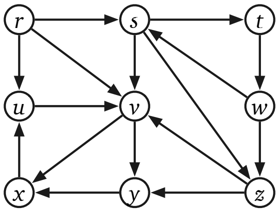
   (a) The set of all strings in $\{0, 1\}^*$ that contain either both or neither of the substrings 01 and 10.

   (b) The set of all strings in $\{0, 1\}^*$ that do not contain the substring 1010.

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

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. ***Clearly*** indicate the following structures in the directed graph below, or write NONE if the indicated structure does not exist. (There are several copies of the graph in the answer booklet.)



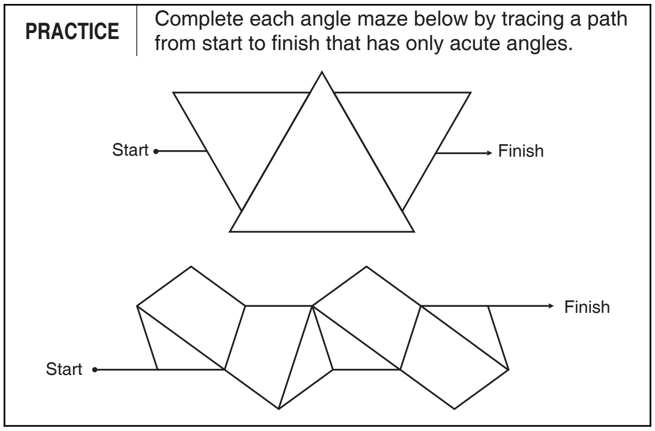
   (a) A depth-first spanning tree rooted at r.

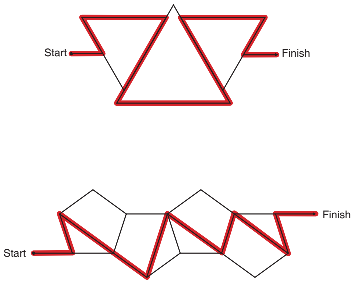
   (b) A breadth-first spanning tree rooted at r.

   (c) A topological order.

   (d) The strongly connected components.

2. The following puzzles appear in my younger daughter's math workbook.[□] (I've put the solutions on the right so you don't waste time solving them during the exam.)


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



   Describe and analyze an algorithm to solve arbitrary acute-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 first maze above has 13 vertices and 15 edges. You are also given two vertices Start and Finish.

   Your algorithm should return TRUE if G contains a walk from Start to Finish that has only acute angles, and FALSE otherwise. Formally, a walk through G is valid if, for any two consecutive edges $u \to v \to w$ in the walk, either $\angle uvw = 180°$ or $0 < \angle uvw < 90°$. Assume you have a subroutine that can compute the angle between any two segments in $O(1)$ time. Do ***not*** assume that angles are multiples of $1°$.

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

3. 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.

4. 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 it is possible to retrieve all the keys without smashing any box except the one your brother has chosen.

(b) Describe and analyze an algorithm to compute the minimum number of boxes that must be smashed to retrieve all the keys.

5. 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.

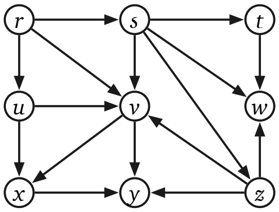
You know all the songs, all the judges, and your own dancing ability extremely well. For each integer k, you know that if you dance to the kth song on the schedule, you will be awarded exactly $\mathit{Score}[k]$ points, but then you will be physically unable to dance for the next $\mathit{Wait}[k]$ 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 $\mathit{Wait}[1\,..\,n]$.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. ***Clearly*** indicate the following structures in the directed graph below, or write NONE if the indicated structure does not exist. (There are several copies of the graph in the answer booklet.)



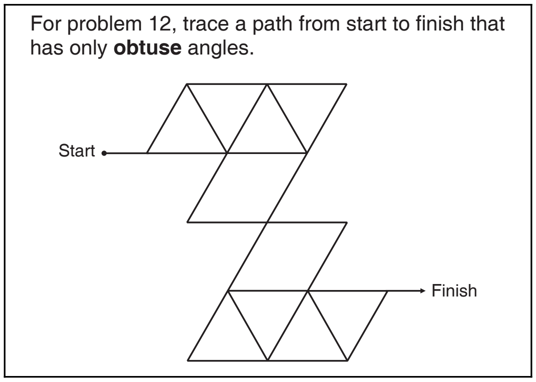
(a) A depth-first spanning tree rooted at r.

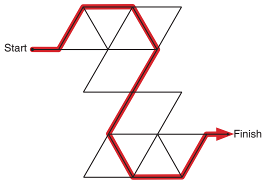
(b) A breadth-first spanning tree rooted at r.

(c) A topological order.

(d) The strongly connected components.

2. The following puzzle appears in my younger daughter's math workbook.□ (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 $90^\circ < \angle uvw \le 180^\circ$ for every pair of consecutive edges $u \to v \to w$ in the walk. Assume you have a subroutine that can compute the angle between any two segments in $O(1)$ time. Do ***not*** assume that angles are multiples of 1°.

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

3. 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?]*

4. 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 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 it is possible to retrieve all the keys without smashing any box except the one your brother has chosen.

   (b) Describe and analyze an algorithm to compute the minimum number of boxes that must be smashed to retrieve all the keys.

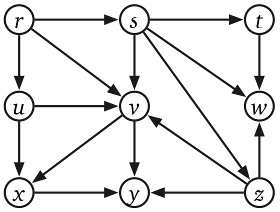
5. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways.

   $\text{BANANA}_{\text{ANANAS}}$ $\quad$ $\text{BAN}_{\text{ANA}}\text{ANA}_{\text{NAS}}$ $\quad$ $\text{B}_{\text{AN}}\text{AN}_{\text{A}}\text{A}_{\text{NA}}\text{NA}_{\text{S}}$

   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.

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. ***Clearly*** indicate the following structures in the directed graph below, or write NONE if the indicated structure does not exist. (There are several copies of the graph in the answer booklet.)



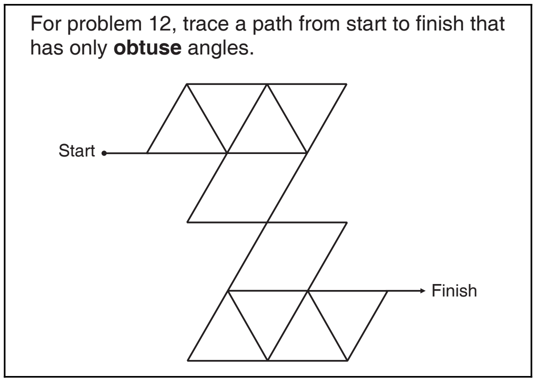
(a) A depth-first spanning tree rooted at r.

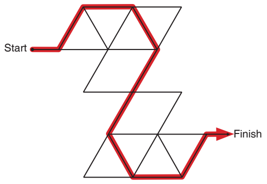
(b) A breadth-first spanning tree rooted at r.

(c) A topological order.

(d) The strongly connected components.

2. The following puzzle appears in my younger daughter's math workbook.□ (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 $90° < \angle uvw \leq 180°$ for every pair of consecutive edges $u \to v \to w$ in the walk. Assume you have a subroutine that can compute the angle between any two segments in $O(1)$ time. Do ***not*** assume that angles are multiples of $1°$.

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

3. 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?]*

4. 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 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 it is possible to retrieve all the keys without smashing any box except the one your brother has chosen.

   (b) Describe and analyze an algorithm to compute the minimum number of boxes that must be smashed to retrieve all the keys.

5. A palindrome is any string that is exactly the same as its reversal, like I, or DEED, or RACECAR, or AMANAPLANACATACANALPANAMA.

   Describe and analyze an algorithm to find the length of the *longest subsequence* of a given string that is also a palindrome. For example, the longest palindrome subsequence of MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM is MHYMRORMYHM, so given that string as input, your algorithm should output the number 11.

CS/ECE 374: Algorithms and Models of Computation, Fall 2016

Final Exam (Version X) — December 15, 2016

Name:											
NetID:											
Section:	A	B	C	D	E	F	G	H	I	J	K
	9–10	10–11	11–12	12–1	1–2	1–2	2–3	2–3	don't	3–4	3–4
	Spencer	Yipu	Charlie	Chao	Alex	Tana	Mark	Konstantinos	know	Mark	Konstantinos

#	1	2	3	4	5	6	Total
Score							
Max	20	10	10	10	10	10	70
Grader							

- ***Don't panic!***
- Please print your name and your NetID and circle your discussion section in the boxes above.
- This is a closed-book, closed-notes, closed-electronics exam. If you brought anything except your writing implements and your two double-sided 8½" × 11" cheat sheets, please put it away for the duration of the exam. In particular, you may not use *any* electronic devices.
- **Please read the entire exam before writing anything.** Please ask for clarification if any question is unclear.
- **The exam lasts 180 minutes.**
- If you run out of space for an answer, continue on the back of the page, or on the blank pages at the end of this booklet, **but please tell us where to look.** Alternatively, feel free to tear out the blank pages and use them as scratch paper.
- As usual, answering any (sub)problem with "I don't know" (and nothing else) is worth 25% partial credit. **Yes, even for problem 1.** Correct, complete, but suboptimal solutions are *always* worth more than 25%. A blank answer is not the same as "I don't know".
- **Beware the Three Deadly Sins.** Give complete solutions, not examples. Don't use weak induction. Declare all your variables.
- If you use a greedy algorithm, you ***must*** prove that it is correct to receive any credit. Otherwise, proofs are required only when we explicitly ask for them.
- **Please return your cheat sheets and all scratch paper with your answer booklet.**
- ***Good luck!*** And have a great winter break!

1. For each of the following questions, indicate ***every*** correct answer by marking the “Yes” box, and indicate ***every*** incorrect answer by marking the “No” box. **Assume P $\neq$ NP.** If there is any other ambiguity or uncertainty, mark the “No” box. For example:

   ☒ Yes ☐ No $2+2=4$

   ☐ Yes ☒ No $x+y=5$

   ☐ Yes ☒ No 3Sat can be solved in polynomial time.

   ☒ Yes ☐ No Jeff is not the Queen of England.

   There are 40 yes/no choices altogether.

   - Each correct choice is worth +½ point.
   - Each incorrect choice is worth −¼ point.
   - To indicate “I don’t know”, write **IDK** next to the boxes; each **IDK** is worth +⅛ point.

(a) Which of the following statements is true for ***every*** language $L \subseteq \{0,1\}^*$?

☐ Yes ☐ No L contains the empty string ε.

☐ Yes ☐ No L^* is infinite.

☐ Yes ☐ No L^* is regular.

☐ Yes ☐ No L is infinite or L is decidable (or both).

☐ Yes ☐ No If L is the union of two regular languages, then L is regular.

☐ Yes ☐ No If L is the union of two decidable languages, then L is decidable.

☐ Yes ☐ No If L is the union of two undecidable languages, then L is undecidable.

☐ Yes ☐ No L is accepted by some NFA with 374 states if and only if L is accepted by some DFA with 374 states.

☐ Yes ☐ No If $L \notin \text{P}$, then L is not regular.

(b) Which of the following languages over the alphabet $\{\mathtt{0}, \mathtt{1}\}$ are ***regular***?

Yes	No	$\{\mathtt{0}^m\mathtt{1}^n \mid m \geq 0 \text{ and } n \geq 0\}$
Yes	No	All strings with the same number of $\mathtt{0}$s and $\mathtt{1}$s
Yes	No	Binary representations of all prime numbers less than 10^{100}
Yes	No	$\{ww \mid w \text{ is a palindrome}\}$
Yes	No	$\{wxw \mid w \text{ is a palindrome and } x \in \{\mathtt{0}, \mathtt{1}\}^*\}$
Yes	No	$\{\langle M \rangle \mid M \text{ accepts a finite number of non-palindromes}\}$

(c) Which of the following languages over the alphabet $\{\mathtt{0}, \mathtt{1}\}$ are ***decidable***?

Yes	No	$\varnothing$
Yes	No	$\{\mathtt{0}^n\mathtt{1}^{2n}\mathtt{0}^n\mathtt{1}^{2n} \mid n \geq 0\}$
Yes	No	$\{ww \mid w \text{ is a palindrome}\}$
Yes	No	$\{\langle M \rangle \mid M \text{ accepts a finite number of non-palindromes}\}$
Yes	No	$\{\langle M, w \rangle \mid M \text{ accepts } ww\}$
Yes	No	$\{\langle M, w \rangle \mid M \text{ accepts } ww \text{ after at most } \lvert w \rvert^2 \text{ transitions}\}$

(d) Which of the following languages can be proved undecidable ***using Rice's Theorem***?

Yes	No	$\{\langle M \rangle \mid M \text{ accepts an infinite number of strings}\}$
Yes	No	$\{\langle M \rangle \mid M \text{ accepts either } \langle M \rangle \text{ or } \langle M \rangle^R\}$
Yes	No	$\{\langle M \rangle \mid M \text{ does not accept exactly 374 palindromes}\}$
Yes	No	$\{\langle M \rangle \mid M \text{ accepts some string } w \text{ after at most } \lvert w \rvert^2 \text{ transitions}\}$

(e) Which of the following is a good English specification of a recursive function that can be used to compute the edit distance between two strings $A[1..n]$ and $B[1..n]$?

Yes	No	$Edit(i,j)$ is the answer for i and j.
Yes	No	$Edit(i,j)$ is the edit distance between $A[i]$ and $B[j]$.
Yes	No	$Edit[1..n,1..n]$ stores the edit distances for all prefixes.
Yes	No	$Edit(i,j)$ is the edit distance between $A[i..n]$ and $B[j..n]$.
Yes	No	$Edit[i,j]$ is the value stored at row i and column j of the table.

(f) Suppose we want to prove that the following language is undecidable.

$$\textsc{Muggle} := \big\{\langle M\rangle \bigm| M \text{ accepts } \texttt{SCIENCE} \text{ but rejects } \texttt{MAGIC}\big\}$$

Professor Potter, your instructor in Defense Against Models of Computation and Other Dark Arts, suggests a reduction from the standard halting language

$$\textsc{Halt} := \big\{\langle M, w\rangle \bigm| M \text{ halts on inputs } w\big\}.$$

Specifically, suppose there is a Turing machine DetectoMuggletum that decides Muggle. Professor Potter claims that the following algorithm decides Halt.

```
DecideHalt($\langle M, w\rangle$):
    Encode the following Turing machine:
        RubberDuck($x$):
            run $M$ on input $w$
            if $x$ = MAGIC
                return False
            else
                return True
    return DetectoMuggletum($\langle$RubberDuck$\rangle$)
```

Which of the following statements is true for all inputs $\langle M, w\rangle$?

Yes	No	If M rejects w, then RubberDuck rejects MAGIC.
Yes	No	If M accepts w, then DetectoMuggletum accepts $\langle$RubberDuck$\rangle$.
Yes	No	If M rejects w, then DecideHalt rejects $\langle M, w\rangle$.
Yes	No	DecideHalt decides the language Halt. (That is, Professor Potter's reduction is actually correct.)
Yes	No	DecideHalt actually runs (or simulates) RubberDuck.

(g) Consider the following pair of languages:

- HamPath $:= \{G \mid G \text{ is a directed graph with a Hamiltonian path}\}$
- 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 ***must*** be true, assuming P≠NP?

Yes	No	Acyclic ∈ NP
Yes	No	Acyclic ∩ HamPath ∈ P
Yes	No	HamPath is decidable.
Yes	No	There is no polynomial-time reduction from HamPath to Acyclic.
Yes	No	There is no polynomial-time reduction from Acyclic to HamPath.

2. 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.

3. Recall that a ***run*** in a string is a maximal non-empty substring in which all symbols are equal. For example, the string `0001111000` consists of three runs: a run of three `0`s, a run of four `1`s, and another run of three `0`s.

   (a) Let L be the set of all strings in $\{\mathtt{0}, \mathtt{1}\}^*$ in which every run of `0`s has odd length and every run of `1`s has even length. For example, L contains ε and `00000` and `0001111000`, but L does not contain `1` or `0000` or `110111000`.

   Describe both a regular expression for L and a DFA that accepts L.

   (b) Let L' be the set of all non-empty strings in $\{\mathtt{0}, \mathtt{1}\}^*$ in which the *number* of runs is equal to the *length* of the first run. For example, L' contains `0` and `1100` and `0000101`, but L' does not contain `0000` or `110111000` or ε.

   Prove that L' is not a regular language.

4. Your cousin 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.

5. ***Prove*** that each of the following languages is undecidable:

   (a) $\{\langle M\rangle \mid M \text{ accepts } \texttt{RICESTHEOREM}\}$

   (b) $\{\langle M\rangle \mid M \text{ rejects } \texttt{RICESTHEOREM}\}$ *[Hint: Use part (a),* ***not*** *Rice's theorem]*

6. 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.]*

(scratch paper)

(scratch paper)

You may assume the following problems are 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 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?

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 an undirected graph G, is there a path in G that visits every vertex exactly once?

HamiltonianCycle: Given an undirected graph G, is there a cycle in G that visits every vertex exactly once?

DirectedHamiltonianCycle: Given an directed graph G, is there a directed 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?

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?

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

You may assume the following languages are undecidable:

$$\begin{aligned}
\textsc{SelfReject} &:= \left\{\langle M\rangle \;\middle|\; M \text{ rejects } \langle M\rangle\right\}\\
\textsc{SelfAccept} &:= \left\{\langle M\rangle \;\middle|\; M \text{ accepts } \langle M\rangle\right\}\\
\textsc{SelfHalt} &:= \left\{\langle M\rangle \;\middle|\; M \text{ halts on } \langle M\rangle\right\}\\
\textsc{SelfDiverge} &:= \left\{\langle M\rangle \;\middle|\; M \text{ does not halt on } \langle M\rangle\right\}\\
\textsc{Reject} &:= \left\{\langle M,w\rangle \;\middle|\; M \text{ rejects } w\right\}\\
\textsc{Accept} &:= \left\{\langle M,w\rangle \;\middle|\; M \text{ accepts } w\right\}\\
\textsc{Halt} &:= \left\{\langle M,w\rangle \;\middle|\; M \text{ halts on } w\right\}\\
\textsc{Diverge} &:= \left\{\langle M,w\rangle \;\middle|\; M \text{ does not halt on } w\right\}\\
\textsc{NeverReject} &:= \left\{\langle M\rangle \;\middle|\; \textsc{Reject}(M) = \varnothing\right\}\\
\textsc{NeverAccept} &:= \left\{\langle M\rangle \;\middle|\; \textsc{Accept}(M) = \varnothing\right\}\\
\textsc{NeverHalt} &:= \left\{\langle M\rangle \;\middle|\; \textsc{Halt}(M) = \varnothing\right\}\\
\textsc{NeverDiverge} &:= \left\{\langle M\rangle \;\middle|\; \textsc{Diverge}(M) = \varnothing\right\}
\end{aligned}$$

CS 473 ✧ Spring 2017

Homework 0

Due Wednesday, January 25, 2017 at 8pm

- **This homework tests your familiarity with prerequisite material:** designing, describing, and analyzing elementary algorithms; fundamental graph problems and algorithms; and especially facility with recursion and induction. Notes on most of this prerequisite material are available on the course web page.
- **Each student must submit individual solutions for this homework.** For all future homeworks, groups of up to three students will be allowed to submit joint solutions.
- **Submit your solutions electronically on Gradescope as PDF files.**
  - Submit a separate file for each numbered problem.
  - You can find a LaTeX solution template on the course web site (soon); please use it if you plan to typeset your homework.
  - If you must submit scanned handwritten solutions, please use dark ink (not pencil) on blank white printer paper (not notebook or graph paper), use a high-quality scanner (not a phone camera), and print the resulting PDF file on a black-and-white printer to verify readability before you submit.

☞ Some important course policies ☜

- **You may use any source at your disposal**—paper, electronic, or human—but you ***must*** cite ***every*** source that you use, and you ***must*** write everything yourself in your own words. See the academic integrity policies on the course web site for more details.
- The answer ***"I don't know"*** (and *nothing* else) is worth 25% partial credit on any problem or subproblem, on any homework or exam, except for extra-credit problems. We will accept synonyms like "No idea" or "WTF" or "¯\(•_•)/¯", but you must write *something*.
- **Avoid the Deadly Sins!** There are a few dangerous writing (and thinking) habits that will trigger an ***automatic zero*** on any homework or exam problem. Yes, really.
  - Always give complete solutions, not just examples.
  - Every algorithm requires an English specification.
  - Greedy algorithms require formal correctness proofs.
  - Never use weak induction.

See the course web site for more information.

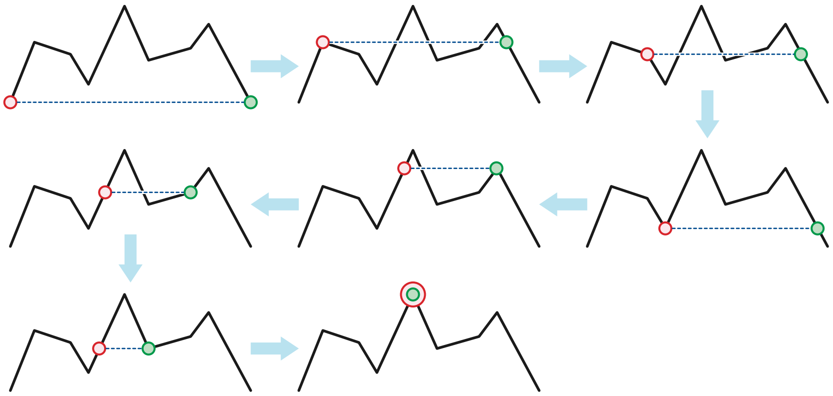
If you have any questions about these policies,
please don't hesitate to ask in class, in office hours, or on Piazza.

---

1. Every cheesy romance movie has a scene where the romantic couple, after a long and frustrating separation, suddenly see each other across a long distance, and then slowly approach one another with unwavering eye contact as the music rolls in and the rain lifts and the sun shines through the clouds and the music swells and everyone starts dancing with rainbows and kittens and chocolate unicorns and. . . .[□]

   Suppose a romantic couple—in grand computer science tradition, named Alice and Bob—enters their favorite park at the east and west entrances and immediately establish eye-contact. They can't just run directly to each other; instead, they must stay on the path that zig-zags through the part between the east and west entrances. To maintain the proper dramatic tension, Alice and Bob must traverse the path so that they always lie on a direct east-west line.

   We can describe the zigzag path as two arrays $X[0..n]$ and $Y[0..n]$, containing the x- and y-coordinates of the corners of the path, in order from the southwest endpoint to the southeast endpoint. The X array is sorted in increasing order, and $Y[0] = Y[n]$. The path is a sequence of straight line segments connecting these corners.



Alice and Bob meet. Alice walks backward in step 2, and Bob walks backward in steps 5 and 6.

   (a) Suppose $Y[0] = Y[n] = 0$ and $Y[i] > 0$ for every other index i; that is, the endpoints of the path are strictly below every other point on the path. Prove that under these conditions, Alice and Bob can meet.

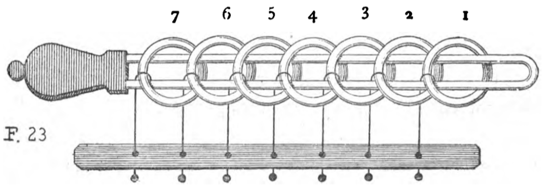
   [Hint: Describe a graph that models all possible locations and transitions of the couple along the path. What are the vertices of this graph? What are the edges? What can you say about the degrees of the vertices?]

   (b) If the endpoints of the path are *not* below every other vertex, Alice and Bob might still be able to meet, or they might not. Describe an algorithm to decide whether Alice and Bob can meet, without either breaking east-west eye contact or stepping off the path, given the arrays $X[0..n]$ and $Y[0..n]$ as input.

   *[Hint: Build the graph from part (a). (How?) What problem do you need to solve on this graph? Call a textbook algorithm to solve that problem. (Do **not** regurgitate the textbook algorithm.) What is your overall running time as a function of n?]*

[□] Fun fact: Damien Chazelle, the director of *Whiplash* and *La La Land*, is the son of Princeton computer science professor Bernard Chazelle.

2. The Tower of Hanoi is a relatively recent descendant of a much older mechanical puzzle known as the Chinese rings, Baguenaudier (a French word meaning “to wander about aimlessly”), Meleda, Patience, Tiring Irons, Prisoner’s Lock, Spin-Out, and many other names. This puzzle was already well known in both China and Europe by the 16th century. The Italian mathematician Luca Pacioli described the 7-ring puzzle and its solution in his unpublished treatise *De Viribus Quantitatis*, written between 1498 and 1506;□ only a few years later, the Ming-dynasty poet Yang Shen described the 9-ring puzzle as “a toy for women and children”.



A drawing of a 7-ring Baguenaudier, from *Récréations Mathématiques* by Édouard Lucas (1891)

The Baguenaudier puzzle has many physical forms, but it typically consists of a long metal loop and several rings, which are connected to a solid base by movable rods. The loop is initially threaded through the rings as shown in the figure above; the goal of the puzzle is to remove the loop.

More abstractly, we can model the puzzle as a sequence of bits, one for each ring, where the ith bit is **1** if the loop passes through the ith ring and **0** otherwise. (Here we index the rings from right to left, as shown in the figure.) The puzzle allows two legal moves:

- You can always flip the 1st (= rightmost) bit.
- If the bit string ends with exactly i **0**s, you can flip the $(i+2)$th bit.

The goal of the puzzle is to transform a string of n **1**s into a string of n **0**s. For example, the following sequence of 21 moves solve the 5-ring puzzle:

$$\texttt{11111} \xrightarrow{1} \texttt{11110} \xrightarrow{3} \texttt{11010} \xrightarrow{1} \texttt{11011} \xrightarrow{2} \texttt{11001} \xrightarrow{1} \texttt{11000} \xrightarrow{5} \texttt{01000}$$
$$\xrightarrow{1} \texttt{01001} \xrightarrow{2} \texttt{01011} \xrightarrow{1} \texttt{01010} \xrightarrow{3} \texttt{01110} \xrightarrow{1} \texttt{01111} \xrightarrow{2} \texttt{01101} \xrightarrow{1} \texttt{01100} \xrightarrow{4} \texttt{00100}$$
$$\xrightarrow{1} \texttt{00101} \xrightarrow{2} \texttt{00111} \xrightarrow{1} \texttt{00110} \xrightarrow{3} \texttt{00010} \xrightarrow{1} \texttt{00011} \xrightarrow{2} \texttt{00001} \xrightarrow{1} \texttt{00000}$$

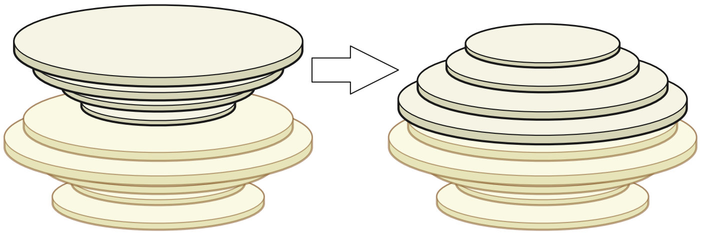
(a) Describe an algorithm to solve the Baguenaudier puzzle. Your input is the number of rings n; your algorithm should print a sequence of moves that solves the n-ring puzzle. For example, given the integer 5 as input, your algorithm should print the sequence $1,3,1,2,1,5,1,2,1,3,1,2,1,4,1,2,1,3,1,2,1$.

(b) *Exactly* how many moves does your algorithm perform, as a function of n? Prove your answer is correct.

(c) ***[Extra credit]*** Call a sequence of moves *reduced* if no move is the inverse of the previous move. Prove that for any non-negative integer n, there is *exactly one* reduced sequence of moves that solves the n-ring Baguenaudier puzzle. *[Hint: See problem 1!]*

□*De Viribus Quantitatis* [*On the Powers of Numbers*] is an important early work on recreational mathematics and perhaps the oldest surviving treatise on magic. Pacioli is better known for *Summa de Aritmetica*, a near-complete encyclopedia of late 15th-century mathematics, which included the first description of double-entry bookkeeping.

3. 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.



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!]*

CS 473 ✧ Spring 2017

Homework 1

Due Wednesday, February 1, 2017 at 8pm

Starting with this homework, groups of up to three people can submit joint solutions. Each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the Gradescope names and email addresses of each group member. In addition, whoever submits the homework must tell Gradescope who their other group members are.

---

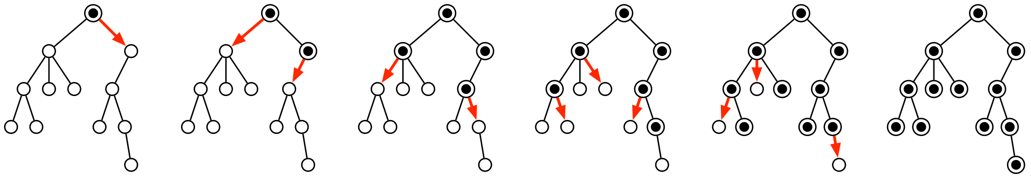
1. A palindrome is any string that is exactly the same as its reversal, like `I`, or `DEED`, or `RACECAR`, or `AMANAPLANACATACANALPANAMA`.

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

   `BUB` • `BASEESAB` • `ANANA`
   `B` • `U` • `BB` • `A` • `SEES` • `ABA` • `NAN` • `A`
   `B` • `U` • `BB` • `A` • `SEES` • `A` • `B` • `ANANA`
   `B` • `U` • `B` • `B` • `A` • `S` • `E` • `E` • `S` • `A` • `B` • `A` • `N` • `ANA`

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

2. Suppose we need to distribute a message to all the nodes in a rooted tree. Initially, only the root node knows the message. In a single round, any node that knows the message can forward it to at most one of its children.



A message being distributed through a tree in five rounds.

   (a) Describe an algorithm to compute the minimum number of rounds required for the message to be delivered to all nodes in a ***binary*** tree.

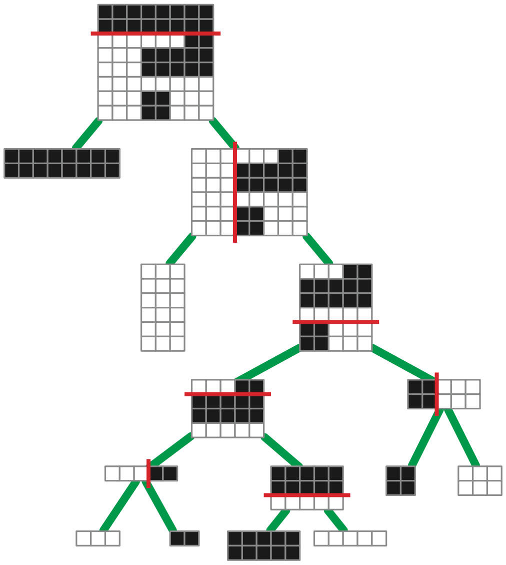
   (b) Describe an algorithm to compute the minimum number of rounds required for the message to be delivered to all nodes in an ***arbitrary rooted*** tree.

   [Hint: Don't forget to justify your algorithm's correctness; you may find the lecture notes on greedy algorithms helpful. Any algorithm for this part also solves part (a).]

3. Suppose you are given an $m \times n$ bitmap, represented by an array $M[1\,..\,m, 1\,..\,n]$ of 0s and 1s. A *solid block* in M is a contiguous subarray $M[i\,..\,i', j\,..\,i']$ in which all bits are equal.

   A *guillotine subdivision* is a compact data structure to represent bitmaps as a recursive decomposition into solid blocks. If the entire bitmap M is a solid block, there is nothing to do. Otherwise, we cut M into two smaller bitmaps along a horizontal or vertical line, and then decompose the two smaller bitmaps recursively.□

   Any guillotine subdivision can be represented as a binary tree, where each internal node stores the position and orientation of a cut, and each leaf stores a single bit 0 or 1 indicting the contents of the corresponding block. The *size* of a guillotine subdivision is the number of leaves in the corresponding binary tree (that is, the final number of solid blocks), and the *depth* of a guillotine subdivision is the depth of the corresponding binary tree.



A guillotine subdivision with size 8 and depth 5.

   (a) Describe and analyze an algorithm to compute a guillotine subdivision of M of minimum *size*.

   (b) Describe and analyze an algorithm to compute a guillotine subdivision of M of minimum *depth*.

□Guillotine subdivisions are similar to kd-trees, except that the cuts in a guillotine subdivision are *not* required to alternate between horizontal and vertical.

Standard dynamic programming rubric. For problems worth 10 poins:

- 3 points for a clear **English** specification of the recursive function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.)
  - + 2 points for describing the function itself. (For example: "$OPT(i, j)$ is the edit distance between $A[1..i]$ and $B[1..i]$.")
  - + 1 point for stating how to call your recursive function to get the final answer. (For example: "We need to compute $OPT(m, n)$.")
  - + An English description of the ***algorithm*** is not sufficient. We want an English description of the underlying recursive ***problem***. In particular, the description should specify precisely the role of each input parameter.
  - + No credit for the rest of the problem if the English description is is missing. (This is a Deadly Sin.)
- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 point for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 points for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 3 points for details of the iterative dynamic programming algorithm
  - + 1 point for describing the memoization data structure
  - + 1 point for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested loops, be sure to specify the nesting order.
  - + 1 point for time analysis
- It is *not* necessary to state a space bound.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem says otherwise.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative psuedocode 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 still need to specify the underlying recursive function in English.)
- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we have included them, significantly more students turned in algorithms that meet the target time bound but didn't work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).

CS 473 ✧ Spring 2017

Homework 2

Due Wednesday, February 8, 2017 at 8pm

There are only two problems, but the first one counts double.

---

1. Suppose you are given a two-dimensional array $M[1..n, 1..n]$ of numbers, which could be positive, negative, or zero, and which are *not* necessarily integers. The ***maximum subarray problem*** asks to find the largest sub of elements in any contiguous subarray of the form $M[i..i', j..j']$. In this problem we'll develop an algorithm for the maximum subarray problem that runs in $O(n^3)$ time.

   The algorithm is a combination of divide and conquer and dynamic programming. Let L be a horizontal line through M that splits the rows (roughly) in half. After some preprocessing, the algorithm finds the maximum-sum subarray that crosses L, the maximum-sum subarray above L, and the maximum-sum subarray below L. The first subarray is found by dynamic programming; the last two subarrays are found recursively.

   (a) For any indices i and j, let $Sum(i, j)$ denote the sum of all elements in the subarray $M[1..i, 1..j]$. Describe an algorithm to compute $Sum(i, j)$ for all indices i and j in $O(n^2)$ time.

   (b) Describe a simple(!!) algorithm to solve the maximum subarray problem in $O(n^4)$ time, using the output of your algorithm for part (a).

   (c) Describe an algorithm to find the maximum-sum subarray ***that crosses L*** in $O(n^3)$ time, using the output of your algorithm for part (a). *[Hint: Consider the top half and the bottom half of M separately.]*

   (d) Describe a divide-and-conquer algorithm to find the maximum-sum subarray in M in $O(n^3)$ time, using your algorithm for part (c) as a subroutine. *[Hint: Why is the running time $O(n^3)$ and not $O(n^3 \log n)$?]*

   In fact, the subproblem in part (c) — and thus the entire maximum subarray problem — can be solved in $n^3/2^{\Omega(\sqrt{\log n})}$ time using a recent algorithm of Ryan Williams. Williams' algorithm can also be used to compute all-pairs shortest paths in the same slightly subcubic running time. The divide-ando-conquer strategy itself is due to Tadao Takaoka.

   There is a simpler $O(n^3)$-time algorithm for the maximum subarray problem, based on Kadane's $O(n)$-time algorithm for the one-dimensional problem. (For every pair of indices i and i', find the best subarray of the form $M[i..i', j..j']$ in $O(n)$ time.) It's unclear whether this approach can be sped up using Williams' algorithm (or its predecessors) without the divide-and-conquer layer.

   An algorithm for the maximum subarray problem (or all-pairs shortest paths) that runs in $O(n^{2.99999999})$ time would be a major breakthrough.

2. The Doctor and River Song decide to play a game on a directed acyclic graph G, which has one source s and one sink t.[□]

   Each player has a token on one of the vertices of G. At the start of the game, The Doctor's token is on the source vertex s, and River's token is on the sink vertex t. The players alternate turns, with The Doctor moving first. On each of his turns, the Doctor moves his token forward along a directed edge; on each of her turns, River moves her token *backward* along a directed edge.

   If the two tokens ever meet on the same vertex, River wins the game. ("Hello, Sweetie!") If the Doctor's token reaches t or River's token reaches s before the two tokens meet, then the Doctor wins the game.

   Describe and analyze an algorithm to determine who wins this game, assuming both players play perfectly. That is, if the Doctor can win *no matter how River moves*, then your algorithm should output "Doctor", and if River can win *no matter how the Doctor moves*, your algorithm should output "River". (Why are these the only two possibilities?) The input to your algorithm is the graph G.

[□] The labels s and t may be abbreviations for the Untempered **S**chism and the **T**ime Vortex, or the Shining World of the Seven Systems (otherwise known as Gallifrey) and Trenzalore, or Skaro and Telos, or Something else Timey-wimey.

CS 473 ✧ Spring 2017

Homework 5

Due Wednesday, February 15, 2017 at 8pm

---

0. **[Warmup only; do not submit solutions]**

   After sending his loyal friends Rosencrantz and Guildenstern off to Norway, Hamlet decides to amuse himself by repeatedly flipping a fair coin until the sequence of flips satisfies some condition. For each of the following conditions, compute the *exact* expected number of flips until that condition is met.

   (a) Hamlet flips heads.

   (b) Hamlet flips both heads and tails (in different flips, of course).

   (c) Hamlet flips heads twice.

   (d) Hamlet flips heads twice in a row.

   (e) Hamlet flips heads followed immediately by tails.

   (f) Hamlet flips more heads than tails.

   (g) Hamlet flips the same positive number of heads and tails.

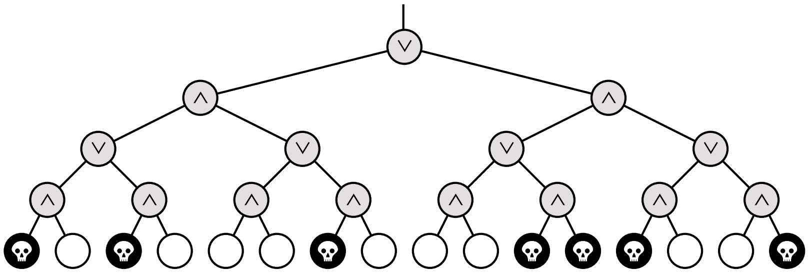
   [Hint: Be careful! If you're relying on intuition instead of a proof, you're probably wrong.]

1. Consider the following non-standard algorithm for shuffling a deck of n cards, initially numbered in order from 1 on the top to n on the bottom. At each step, we remove the top card from the deck and ***insert*** it randomly back into in the deck, choosing one of the n possible positions uniformly at random. The algorithm ends immediately after we pick up card $n-1$ and insert it randomly into the deck.

   (a) Prove that this algorithm uniformly shuffles the deck, meaning each permutation of the deck has equal probability. *[Hint: Prove that at all times, the cards below card $n-1$ are uniformly shuffled.]*

   (b) What is the *exact* expected number of steps executed by the algorithm? *[Hint: Split the algorithm into phases that end when card $n-1$ changes position.]*

2. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.



You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

(a) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy!]*

(b) Unfortunately, Death won't give you enough time to look at every node in the tree. Describe a *randomized* algorithm that determines whether you can win in $O(3^n)$ expected time. *[Hint: Consider the case $n = 1$.]*

*(c) **[Extra credit]** Describe and analyze a randomized algorithm that determines whether you can win in $O(c^n)$ expected time, for some constant $c < 3$. *[Hint: You may not need to change your algorithm from part (b) at all!]*

3. The following randomized variant of "one-armed quicksort" selects the kth smallest element in an unsorted array $A[1 \,..\, n]$. As usual, Partition($A[1 \,..\, n], p$) partitions the array A into three parts by comparing the pivot element $A[p]$ to every other element, using $n-1$ comparisons, and returns the new index of the pivot element.

```
QuickSelect($A[1 .. n], k$):
    $r \leftarrow$ Partition($A[1 .. n]$, Random($n$))
    if $k < r$
        return QuickSelect($A[1 .. r-1], k$)
    else if $k > r$
        return QuickSelect($A[r+1 .. n], k-r$)
    else
        return $A[k]$
```

(a) State a recurrence for the expected running time of QuickSelect, as a function of n and k.

(b) What is the *exact* probability that QuickSelect compares the ith smallest and jth smallest elements in the input array? The correct answer is a simple function of i, j, and k. *[Hint: Check your answer by trying a few small examples.]*

(c) What is the *exact* probability that in one of the recursive calls to QuickSelect, the first argument is the subarray $A[i \,..\, j]$? The correct answer is a simple function of i, j, and k. *[Hint: Check your answer by trying a few small examples.]*

(d) Show that for any n and k, the expected running time of QuickSelect is $\Theta(n)$. You can use either the recurrence from part (a) or the probabilities from part (b) or (c).

CS 473 ✧ Spring 2017

Homework 4

Due Wednesday, March 1, 2017 at 8pm

1. Recall that a *priority search tree* is a binary tree in which every node has both a *search key* and a *priority*, arranged so that the tree is simultaneously a binary search tree for the keys and a min-heap for the priorities. A ***heater*** is a priority search tree in which the *priorities* are given by the user, and the *search keys* are distributed uniformly and independently at random in the real interval $[0, 1]$. Intuitively, a heater is a sort of anti-treap.

   The following problems consider an n-node heater T whose priorities are the integers from 1 to n. We identify nodes in T by their *priorities*; thus, “node 5” means the node in T with *priority* 5. For example, the min-heap property implies that node 1 is the root of T. Finally, let i and j be integers with $1 \leq i < j \leq n$.

   (a) What is the *exact* expected depth of node j in an n-node heater? Answering the following subproblems will help you:

      i. Prove that in a random permutation of the $(i+1)$-element set $\{1, 2, \ldots, i, j\}$, elements i and j are adjacent with probability $2/(i+1)$.

      ii. Prove that node i is an ancestor of node j with probability $2/(i+1)$. *[Hint: Use the previous question!]*

      iii. What is the probability that node i is a *descendant* of node j? *[Hint: Do **not** use the previous question!]*

   (b) Describe and analyze an algorithm to insert a new item into a heater. Analyze the expected running time as a function of the number of nodes.

   (c) Describe an algorithm to delete the minimum-priority item (the root) from an n-node heater. What is the expected running time of your algorithm?

2. Suppose we are given a coin that may or may not be biased, and we would like to compute an accurate *estimate* of the probability of heads. Specifically, if the actual unknown probability of heads is p, we would like to compute an estimate $\tilde{p}$ such that

$$\Pr[|\tilde{p} - p| > \varepsilon] < \delta$$

   where ε is a given ***accuracy*** or ***error*** parameter, and δ is a given ***confidence*** parameter.

   The following algorithm is a natural first attempt; here Flip( ) returns the result of an independent flip of the unknown coin.

```
MeanEstimate($\varepsilon$):
  $count \leftarrow 0$
  for $i \leftarrow 1$ to $N$
      if Flip( ) = Heads
          $count \leftarrow count + 1$
  return $count/N$
```

   (a) Let $\tilde{p}$ denote the estimate returned by MeanEstimate(ε). Prove that $\mathrm{E}[\tilde{p}] = p$.

(b) Prove that if we set $N = \lceil \alpha/\varepsilon^2 \rceil$ for some appropriate constant α, then we have $\Pr[|\tilde{p} - p| > \varepsilon] < 1/4$. *[Hint: Use Chebyshev's inequality.]*

(c) We can increase the previous estimator's confidence by running it multiple times, independently, and returning the *median* of the resulting estimates.

```
MedianOfMeansEstimate($\delta, \varepsilon$):
  for $j \leftarrow 1$ to $K$
      $estimate[j] \leftarrow$ MeanEstimate($\varepsilon$)
  return Median($estimate[1..K]$)
```

Let p^* denote the estimate returned by MedianOfMeansEstimate(δ, ε). Prove that if we set $N = \lceil \alpha/\varepsilon^2 \rceil$ (inside MeanEstimate) and $K = \lceil \beta \ln(1/\delta) \rceil$, for some appropriate constants α and β, then $\Pr[|p^* - p| > \varepsilon] < \delta$. *[Hint: Use Chernoff bounds.]*

CS 473 ✧ Spring 2017

Homework 5

Due Wednesday, March 8, 2017 at 8pm

1. ***Reservoir sampling*** is a method for choosing an item uniformly at random from an arbitrarily long stream of data.

```
GetOneSample(stream $S$):
    $\ell \leftarrow 0$
    while $S$ is not done
        $x \leftarrow$ next item in $S$
        $\ell \leftarrow \ell + 1$
        if Random($\ell$) $= 1$
            sample $\leftarrow x$        $(\star)$
    return sample
```

At the end of the algorithm, the variable ℓ stores the length of the input stream S; this number is *not* known to the algorithm in advance. If S is empty, the output of the algorithm is (correctly!) undefined.

In the following, consider an arbitrary non-empty input stream S, and let n denote the (unknown) length of S.

(a) Prove that the item returned by GetOneSample(S) is chosen uniformly at random from S.

(b) What is the *exact* expected number of times that GetOneSample(S) executes line $(\star)$?

(c) What is the *exact* expected value of ℓ when GetOneSample(S) executes line $(\star)$ for the *last* time?

(d) What is the *exact* expected value of ℓ when either GetOneSample(S) executes line $(\star)$ for the *second* time (or the algorithm ends, whichever happens first)?

(e) Describe and analyze an algorithm that returns a subset of k distinct items chosen uniformly at random from a data stream of length at least k. The integer k is given as part of the input to your algorithm. Prove that your algorithm is correct.

For example, if $k = 2$ and the stream contains the sequence ⟨♠, ♥, ♦, ♣⟩, the algorithm should return the subset {♦, ♠} with probability $1/6$.

2. ***Tabulated hashing*** uses tables of random numbers to compute hash values. Suppose $|\mathcal{U}| = 2^w \times 2^w$ and $m = 2^\ell$, so the items being hashed are pairs of w-bit strings (or $2w$-bit strings broken in half) and hash values are ℓ-bit strings.

Let $A[0\,..\,2^w - 1]$ and $B[0\,..\,2^w - 1]$ be arrays of independent random ℓ-bit strings, and define the hash function $h_{A,B} \colon \mathcal{U} \to [m]$ by setting

$$h_{A,B}(x, y) := A[x] \oplus B[y]$$

where $\oplus$ denotes bit-wise exclusive-or. Let $\mathcal{H}$ denote the set of all possible functions $h_{A,B}$. Filling the arrays A and B with independent random bits is equivalent to choosing a hash function $h_{A,B} \in \mathcal{H}$ uniformly at random.

(a) Prove that $\mathscr{H}$ is 2-uniform.

(b) Prove that $\mathscr{H}$ is 3-uniform. *[Hint: Solve part (a) first.]*

(c) Prove that $\mathscr{H}$ is ***not*** 4-uniform.

Yes, "see part (b)" is worth full credit for part (a), but only if your solution to part (b) is correct.

CS 473 ✧ Spring 2017

Homework 6

Due Wednesday, March 16, 2017 at 8pm

1. Suppose you are given a directed graph $G = (V, E)$, two vertices s and t, a capacity function $c\colon E \to \mathbb{R}^+$, and a second function $f : E \to \mathbb{R}$. Describe and analyze an algorithm to determine whether f is a maximum (s,t)-flow in G. *[Hint: Don't make any "obvious" assumptions!]*

2. Suppose you are given a flow network G with ***integer*** edge capacities and an ***integer*** maximum flow f^* in G. Describe algorithms for the following operations:

   (a) Increment(e): Increase the capacity of edge e by 1 and update the maximum flow.

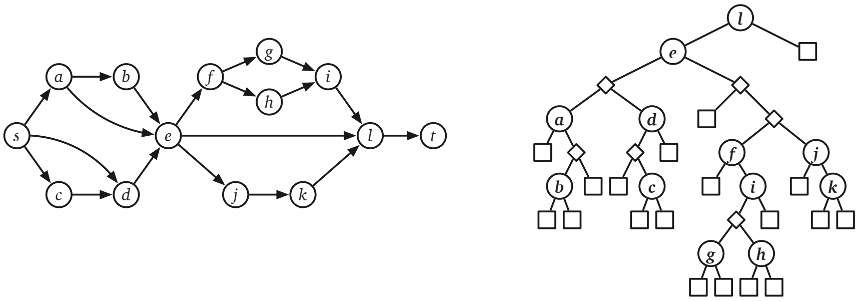
   (b) Decrement(e): Decrease the capacity of edge e by 1 and update the maximum flow.

   Both algorithms should modify f^* so that it is still a maximum flow, but more quickly than recomputing a maximum flow from scratch.

3. An ***(s, t)-series-parallel*** graph is a directed acyclic graph with two designated vertices s (the *source*) and t (the *target* or *sink*) and with one of the following structures:

   - **Base case:** A single directed edge from s to t.
   - **Series:** The union of an (s,u)-series-parallel graph and a (u,t)-series-parallel graph that share a common vertex u but no other vertices or edges.
   - **Parallel:** The union of two smaller (s,t)-series-parallel graphs with the same source s and target t, but with no other vertices or edges in common.

   Every (s,t)-series-parallel graph G can be represented by a ***decomposition tree***, which is a binary tree with three types of nodes: leaves corresponding to single edges in G, series nodes (each labeled by some vertex), and parallel nodes (unlabeled).



An series-parallel graph and its decomposition tree.

   (a) Suppose you are given a directed graph G with two special vertices s and t. Describe and analyze an algorithm that either builds a decomposition tree for G or correctly reports that G is not (s,t)-series-parallel. *[Hint: Build the tree from the bottom up.]*

   (b) Describe and analyze an algorithm to compute a maximum (s,t)-flow in a given (s,t)-series-parallel flow network with arbitrary edge capacities. *[Hint: In light of part (a), you can assume that you are actually given the decomposition tree.]*

CS 473 ✧ Spring 2017

Homework 7

Due Wednesday, March 29, 2017 at 8pm

1. Suppose we are given an array $A[1\,..\,m][1\,..\,n]$ of non-negative real numbers. We want to ***round*** A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longmapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

   (a) Describe and analyze an efficient algorithm that either rounds A in this fashion, or correctly reports that no such rounding is possible.

   (b) Prove that a legal rounding is possible *if and only if* the sum of entries in each row is an integer, and the sum of entries in each column is an integer. In other words, prove that either your algorithm from part (a) returns a legal rounding, or a legal rounding is *obviously* impossible.

2. Quentin, Alice, and the other Brakebills Physical Kids are planning an excursion through the Neitherlands to Fillory. The Neitherlands is a vast, deserted city composed of several plazas, each containing a single fountain that can magically transport people to a different world. Adjacent plazas are connected by gates, which have been cursed by the Beast. The gates between plazas are open only for five minutes every hour, all simultaneously—from 12:00 to 12:05, then from 1:00 to 1:05, and so on—and are otherwise locked. During those five minutes, if more than one person passes through any single gate, the Beast will detect their presence.□ Moreover, anyone attempting to open a locked gate, or attempting to pass through more than one gate within the same five-minute period will turn into a niffin.□ However, any number of people can safely pass through *different* gates at the same time and/or pass through the same gate at *different* times.

   You are given a map of the Neitherlands, which is a graph G with a vertex for each fountain and an edge for each gate, with the fountains to Earth and Fillory clearly marked. Suppose you are also given a positive integer h. Describe and analyze an algorithm to compute the maximum number of people that can walk from the Earth fountain to the Fillory fountain in at most h hours—that is, after the gates have opened at most h times—without anyone alerting the Beast or turning into a niffin. *[Hint: Build a different graph.]*

□This is very bad.

□This is very very bad.

Rubric (graph reductions): For a problem worth 10 points, solved by reduction to maximum flow:

- 2 points for a complete description of the relevant flow network, specifying the set of vertices, the set of edges (being careful about direction), the source and target vertices s and t, and the capacity of every edge. (If the flow network is part of the original input, just say that.)
- 1 point for a description of the algorithm to construct this flow network from the stated input. This could be as simple as "We can construct the flow network in $O(n^3)$ time by brute force."
- 1 point for precisely specifying the problem to be solved on the flow network (for example: "maximum flow from x to y") and the algorithm (For example: "Ford-Fulkerson" or "Orlin") to solve that problem. Do *not* regurgitate the details of the maximum-flow algorithm itself.
- 1 point for a description of the algorithm to extract the answer to the stated problem from the maximum flow. This could be as simple as "Return TRUE if the maximum flow value is at least 42 and FALSE otherwise."
- **4 points for a proof that your reduction is correct.** This proof will almost always have two components (worth 2 points each). For example, if your algorithm returns a boolean, you should prove that its True answers are correct and that its False answers are correct. If your algorithm returns a number, you should prove that number is neither too large nor too small.
- 1 point for the running time of the overall algorithm, expressed as a function of the original input parameters, *not* just the number of vertices and edges in your flow network. You may assume that maximum flows can be computed in $O(VE)$ time.

Reductions to other flow-based problems described in class or in the notes (for example: edge-disjoint paths, maximum bipartite matching, minimum-cost circulation) or to other standard graph problems (for example: reachability, topological sort, minimum spanning tree, all-pairs shortest paths) have similar requirements.

CS 473 ✧ Spring 2017

Homework 8

Due Wednesday, April 12, 2017 at 8pm

1. Recall that a ***path cover*** of a directed acyclic graph is a collection of directed paths, such that every vertex in G appears in at least one path. We previously saw how to compute *disjoint* path covers (where each vertex lies on *exactly* one path) by reduction to maximum bipartite matching. Your task in this problem is to compute path covers *without* the disjointness constraint.

   (a) Suppose you are given a dag G with a unique source s and a unique sink t. Describe an algorithm to find the smallest path cover of G in which every path starts at s and ends at t.

   (b) Describe an algorithm to find the smallest path cover of an arbitrary dag G, with no additional restrictions on the paths. *[Hint: Use part (a).]*

2. Recall that an ***(s,t)-series-parallel*** graph is an directed acyclic graph with two designated vertices s (the *source*) and t (the *target* or *sink*) and with one of the following structures:

   - **Base case:** A single directed edge from s to t.
   - **Series:** The union of an (s,u)-series-parallel graph and a (u,t)-series-parallel graph that share a common vertex u but no other vertices or edges.
   - **Parallel:** The union of two smaller (s,t)-series-parallel graphs with the same source s and target t, but with no other vertices or edges in common.

   Any series-parallel graph can be represented by a binary *decomposition tree*, whose interior nodes correspond to series compositions and parallel compositions, and whose leaves correspond to individual edges. In a previous homework, we saw how to construct the decomposition tree for any series-parallel graph in $O(V+E)$ time, and then how to compute a maximum (s,t)-flow in $O(V+E)$ time.

   Describe an efficient algorithm to compute a *minimum-cost* maximum flow from s to t in an (s,t)-series-parallel graph G in which every edge has capacity 1 and arbitrary cost. *[Hint: First consider the special case where G has only two vertices but lots of edges.]*

3. Every year, Professor Dumbledore assigns the instructors at Hogwarts to various faculty committees. There are n faculty members and c committees. Each committee member has submitted a list of their *prices* for serving on each committee; each price could be positive, negative, zero, or even infinite. For example, Professor Snape might declare that he would serve on the Student Recruiting Committee for 1000 Galleons, that he would *pay* 10000 Galleons to serve on the Defense Against the Dark Arts Course Revision Committee, and that he would not serve on the Muggle Relations committee for any price.

   Conversely, Dumbledore knows how many instructors are needed for each committee, as well as a list of instructors who would be suitable members for each committee. (For example: "Dark Arts Revision: 5 members, anyone but Snape.") If Dumbledore assigns an instructor to a committee, he must pay that instructor's price from the Hogwarts treasury.

Dumbledore needs to assign instructors to committees so that (1) each committee is full, (3) no instructor is assigned to more than three committees, (2) only suitable and willing instructors are assigned to each committee, and (4) the total cost of the assignment is as small as possible. Describe and analyze an efficient algorithm that either solves Dumbledore's problem, or correctly reports that there is no valid assignment whose total cost is finite.

CS 473 ✧ Spring 2017

Homework 9

Due Wednesday, April 19, 2017 at 8pm

1. Suppose you are given an arbitrary directed graph $G = (V, E)$ with arbitrary edge weights $\ell \colon E \to \mathbb{R}$. Each edge in G is colored either red, white, or blue to indicate how you are permitted to modify its weight:

   - You may increase, but not decrease, the length of any red edge.
   - You may decrease, but not increase, the length of any blue edge.
   - You may not change the length of any black edge.

   The ***cycle nullification*** problem asks whether it is possible to modify the edge weights—subject to these color constraints—so that *every cycle in G has length* 0. Both the given weights and the new weights of the individual edges can be positive, negative, or zero. To keep the following problems simple, assume that G is strongly connected.

   (a) Describe a linear program that is feasible if and only if it is possible to make every cycle in G have length 0. *[Hint: Pick an arbitrary vertex s, and let* $\operatorname{dist}(v)$ *denote the length of every walk from s to v.]*

   (b) Construct the dual of the linear program from part (a). *[Hint: Choose a convenient objective function for your primal LP.]*

   (c) Give a self-contained description of the combinatorial problem encoded by the dual linear program from part (b), and prove *directly* that it is equivalent to the original cycle nullification problem. Do not use the words "linear", "program", or "dual". Yes, you have seen this problem before.

   (d) Describe and analyze an algorithm to determine *in $O(EV)$ time* whether it is possible to make every cycle in G have length 0, using your dual formulation from part (c). Do not use the words "linear", "program", or "dual".

2. *There is no problem 2.*

CS 473 ✧ Spring 2017

Homework 10

Due Wednesday, April 26, 2017 at 8pm

1. An ***integer linear program*** is a linear program with the additional explicit constraint that the variables must take *only* integer values. The ILP-Feasibility problem asks whether there is an integer vector that satisfies a given system of linear inequalities—or more concisely, whether a given integer linear program is feasible.

   Describe a polynomial-time reduction from 3Sat to ILP-Feasibility. Your reduction implies that ILP-Feasibility is NP-hard.

2. There are two different versions of the Hamiltonian cycle problem, one for directed graphs and one for undirected graphs. We saw a proof in class (and there are two proofs in the notes) that the *directed* Hamiltonian cycle problem is NP-hard.

   (a) Describe a polynomial-time reduction from the *undirected* Hamiltonian cycle problem to the *directed* Hamiltonian cycle problem. Prove your reduction is correct.

   (b) Describe a polynomial-time reduction from the *directed* Hamiltonian cycle problem to the *undirected* Hamiltonian cycle problem. Prove your reduction is correct.

   (c) Which of these two reductions implies that the *undirected* Hamiltonian cycle problem is NP-hard?

3. Recall that a 3CNF formula is a conjunction (And) of several distinct clauses, each of which is a disjunction (Or) of exactly three distinct literals, where each literal is either a variable or its negation.

   Suppose you are given a magic black box that can determine **in polynomial time**, whether an arbitrary given 3CNF formula is satisfiable. Describe and analyze a **polynomial-time** algorithm that either computes a satisfying assignment for a given 3CNF formula or correctly reports that no such assignment exists, using the magic black box as a subroutine. *[Hint: Call the magic black box more than once. First imagine an even more magical black box that can decide Sat for arbitrary boolean formulas, not just 3CNF formulas.]*

Rubric (for all polynomial-time reductions): 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).
+ 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 473 ✧ Spring 2017

Homework 11

"Due" Wednesday, May 3, 2017 at 8pm

This homework will not be graded.
However, material covered by this homework *may* appear on the final exam.

---

1. Let Φ be a boolean formula in conjunctive normal form, with exactly three literals in each clause. Recall that an assignment of boolean values to the variables in Φ ***satisfies*** a clause if at least one of its literals is TRUE. The ***maximum satisfiability problem*** for 3CNF formulas, usually called MAX3SAT, asks for the maximum number of clauses that can be simultaneously satisfied by a single assignment.

   Solving MAX3SAT exactly is clearly also NP-hard; this question asks about approximation algorithms. Let $Max3Sat(\Phi)$ denote the maximum number of clauses in Φ that can be simultaneously satisfied by one variable assignment.

   (a) Suppose we assign variables in Φ to be TRUE or FALSE using independent fair coin flips. Prove that the expected number of satisfied clauses is at least $\frac{7}{8}Max3Sat(\Phi)$.

   (b) Let k^+ denote the number of clauses satisfied by setting every variable in Φ to TRUE, and let k^- denote the number of clauses satisfied by setting every variable in Φ to FALSE. Prove that $\max\{k^+, k^-\} \geq Max3Sat(\Phi)/2$.

   (c) Let $Min3Unsat(\Phi)$ denote the *minimum* number of clauses that can be simultaneously left *unsatisfied* by a single assignment. Prove that it is NP-hard to approximate $Min3Unsat(\Phi)$ within a factor of $10^{10^{100}}$.

2. Consider the following algorithm for approximating the minimum vertex cover of a connected graph G: ***Return the set of all non-leaf nodes of an arbitrary depth-first spanning tree***. (Recall that a depth-first spanning tree is a rooted tree; the root is not considered a leaf, even if it has only one neighbor in the tree.)

   (a) Prove that this algorithm returns a vertex cover of G.

   (b) Prove that this algorithm returns a 2-approximation to the smallest vertex cover of G.

   (c) Describe an infinite family of connected graphs for which this algorithm returns a vertex cover of size *exactly* $2 \cdot \text{OPT}$. This family implies that the analysis in part (b) is tight. *[Hint: First find just* ***one*** *such graph, with few vertices.]*

3. Consider the following modification of the “dumb” 2-approximation algorithm for minimum vertex cover that we saw in class. The only change is that we return a set of edges instead of a set of vertices.

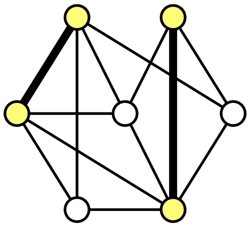
```
ApproxMinMaxMatching($G$):
  $M \leftarrow \varnothing$
  while G has at least one edge
      $uv \leftarrow$ any edge in $G$
      $G \leftarrow G \setminus \{u, v\}$
      $M \leftarrow M \cup \{uv\}$
  return $M$
```

(a) Prove that the output subgraph M is a *matching*—no pair of edges in M share a common vertex.

(b) Prove that M is a *maximal* matching—M is not a proper subgraph of another matching in G.

(c) Prove that M contains at most twice as many edges as the *smallest* maximal matching in G.

(d) Describe an infinite family of graphs G such that the matching returned by ApproxMinMaxMatching(G) contains exactly twice as many edges as the smallest maximum matching in G. This family implies that the analysis in part (c) is tight. *[Hint: First find just **one** such graph, with few vertices.]*



The smallest maximal matching in a graph.

CS 473 ✧ Spring 2017

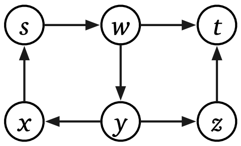
Midterm 1

February 21, 2016

1. Recall that a ***walk*** in a directed graph G is an arbitrary sequence of vertices $v_0 \to v_1 \to \cdots \to v_k$, such that $v_{i-1} \to v_i$ is an edge in G for every index i. A ***path*** is a walk in which no vertex appears more than once.

   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 whose length is a multiple of 3.

   For example, given the graph shown 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.]

2. 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. A ***smooth*** shuffle of X and Y is a shuffle of X and Y that never uses more than two consecutive symbols of either string. For example,

   - `prDoYgNAraMmmIiCng` is a smooth shuffle of `DYNAMIC` and `programming`.
   - `DYprNogrAaMmmICing` is a shuffle of `DYNAMIC` and `programming`, but it is not a smooth shuffle (because of the substrings `ogr` and `ing`).

   Describe and analyze an algorithm to decide, given three strings X, Y, and Z, whether Z is a smooth shuffle of X and Y.

3. (a) Describe an algorithm that simulates a fair coin, using independent rolls of a fair three-sided die as your only source of randomness. Your algorithm should return either HEADS or TAILS, each with probability $1/2$.

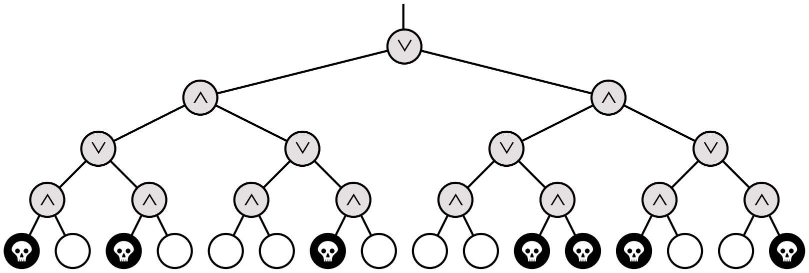
   (b) What is the expected number of die rolls performed by your algorithm in part (a)?

   (c) Describe an algorithm that simulates a fair three-sided die, using independent fair coin flips as your only source of randomness. Your algorithm should return either 1, 2, or 3, each with probability $1/3$.

   (d) What is the expected number of coin flips performed by your algorithm in part (c)?

4. Death knocks on Dirk Gently's door one cold blustery morning and challenges him to a game. Emboldened by his experience with algorithms students, Death presents Dirk with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, Dirk and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, Dirk dies; if it's white, Dirk lives forever. Dirk moves first, so Death gets the last turn.

   (Yes, this is precisely the same game from Homework 3.)



   Unfortunately, Dirk slept through Death's explanation of the rules, so he decides to just play randomly. Whenever it's Dirk's turn, he flips a fair coin and moves left on heads, or right on tails, confident that the Fundamental Interconnectedness of All Things will keep him alive, unless it doesn't. Death plays much more purposefully, of course, always choosing the move that maximizes the probability that Dirk loses the game.

   (a) Describe an algorithm that computes *the probability* that Dirk wins the game against Death.

   (b) Realizing that Dirk is not taking the game seriously, Death gives up in desperation and decides to also play randomly! Describe an algorithm that computes *the probability* that Dirk wins the game again Death, assuming *both* players flip fair coins to decide their moves.

   For both algorithms, the input consists of the integer n (specifying the depth of the tree) and an array of 4^n bits specifying the colors of leaves in left-to-right order.

CS 473 ✧ Spring 2017

Midterm 2

April 4, 2016

---

1. Let $G = (V, E)$ be an arbitrary undirected graph. Suppose we color each vertex of G uniformly and independently at random from a set of three colors: red, green, or blue. An edge of G is *monochromatic* if both of its endpoints have the same color.

   (a) What is the *exact* expected number of monochromatic edges? (Your answer should be a simple function of V and E.)

   (b) For each edge $e \in E$, define an indicator variable X_e that equals 1 if e is monochromatic and 0 otherwise. ***Prove*** that
   $$\Pr[(X_a = 1) \land (X_b = 1)] \;=\; \Pr[X_a = 1] \cdot \Pr[X_b = 1]$$
   for every pair of edges $a \neq b$. This claim implies that the random variables X_e are pairwise independent.

   (c) ***Prove*** that there is a graph G such that
   $$\Pr[(X_a = 1) \land (X_b = 1) \land (X_c = 1)] \;\neq\; \Pr[X_a = 1] \cdot \Pr[X_b = 1] \cdot \Pr[X_c = 1]$$
   for some triple of distinct edges a, b, c in G. This claim implies that the random variables X_e are *not necessarily* 3-wise independent.

2. The White Rabbit has a very poor memory, and so he is constantly forgetting his regularly scheduled appointments with the Queen of Hearts. In an effort to avoid further beheadings of court officials, The King of Hearts has installed an app on Rabbit's pocket watch to automatically remind Rabbit of any upcoming appointments. For each reminder Rabbit receives, Rabbit has a 50% chance of actually remembering his appointment (decided by an independent fair coin flip).

   First, suppose the King of Hearts sends Rabbit k separate reminders for a *single* appointment.

   (a) What is the *exact* probability that Rabbit will remember his appointment? Your answer should be a simple function of k.

   (b) What value of k should the King choose so that the probability that Rabbit will remember this appointment is at least $1 - 1/n^\alpha$? Your answer should be a simple function of n and α.

   Now suppose the King of Hearts sends Rabbit k separate reminders for each of n different appointments. (That's nk reminders altogether.)

   (c) What is the *exact* expected number of appointments that Rabbit will remember? Your answer should be a simple function of n and k.

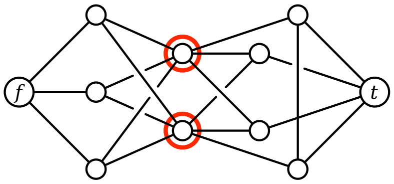
   (d) What value of k should the King choose so that the probability that Rabbit remembers *every* appointment is at least $1 - 1/n^\alpha$? Again, your answer should be a simple function of n and α.

   [Hint: There is a simple solution that does not use tail inequalities.]

3. The Island of Sodor is home to an extensive rail network. Recently, several cases of a deadly contagious disease (either swine flu or zombies; reports are unclear) have been reported in the village of Ffarquhar. The controller of the Sodor railway plans to close certain railway stations to prevent the disease from spreading to Tidmouth, his home town. No trains can pass through a closed station. To minimize expense (and public notice), he wants to close as few stations as possible. However, he cannot close the Ffarquhar station, because that would expose him to the disease, and he cannot close the Tidmouth station, because then he couldn't visit his favorite pub.

   Describe and analyze an algorithm to find the minimum number of stations that must be closed to block all rail travel from Ffarquhar to Tidmouth. The Sodor rail network is represented by an undirected graph, with a vertex for each station and an edge for each rail connection between two stations. Two special vertices f and t represent the stations in Ffarquhar and Tidmouth.

   For example, given the following input graph, your algorithm should return the integer 2.



4. The Department of Commuter Silence at Shampoo-Banana University has a flexible curriculum with a complex set of graduation requirements. The department offers n different courses, and there are m different requirements. Each requirement specifies a subset of the n courses and the number of courses that must be taken from that subset. The subsets for different requirements may overlap, but each course can only be used to satisfy *at most one* requirement.

   For example, suppose there are $n = 5$ courses A, B, C, D, E and $m = 2$ graduation requirements:

   - You must take at least 2 courses from the subset $\{A, B, C\}$.
   - You must take at least 2 courses from the subset $\{C, D, E\}$.

   Then a student who has only taken courses B, C, D cannot graduate, but a student who has taken either A, B, C, D or B, C, D, E can graduate.

   Describe and analyze an algorithm to determine whether a given student can graduate. The input to your algorithm is the list of m requirements (each specifying a subset of the n courses and the number of courses that must be taken from that subset) and the list of courses the student has taken.

CS 473 ✧ Spring 2017

Final Exam

May 10, 2017

1. A ***three-dimensional matching*** in an undirected graph G is a collection of vertex-disjoint triangles. A three-dimensional matching is *maximal* if it is not a proper subgraph of a larger three-dimensional matching in the same graph.

   (a) Let M and M' be two arbitrary maximal three-dimensional matchings in the same underlying graph G. ***Prove*** that $|M| \leq 3 \cdot |M'|$.

   (b) Finding the *largest* three-dimensional matching in a given graph is NP-hard. Describe and analyze a fast 3-approximation algorithm for this problem.

   (c) Finding the *smallest maximal* three-dimensional matching in a given graph is NP-hard. Describe and analyze a fast 3-approximation algorithm for this problem.

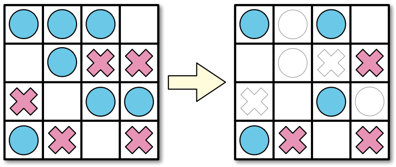
2. Let $G = (V, E)$ be an arbitrary dag with a unique source s and a unique sink t. Suppose we compute a random walk from s to t, where at each node v (except t), we choose an outgoing edge $v \to w$ uniformly at random to determine the successor of v.

   (a) Describe and analyze an algorithm to compute, for every vertex v, the probability that the random walk visits v.

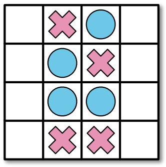
   (b) Describe and analyze an algorithm to compute the expected number of edges in the random walk.

   Assume all relevant arithmetic operations can be performed exactly in $O(1)$ time.

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, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

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

4. Suppose you are given a bipartite graph $G = (L \sqcup R, E)$ and a maximum matching M in G. Describe and analyze fast algorithms for the following problems:

   (a) Insert(e): Insert a new edge e into G and update the maximum matching. (You can assume that e is not already an edge in G, and that $G + e$ is still bipartite.)

   (b) Delete(e): Delete the existing edge e from G and update the maximum matching. (You can assume that e is in fact an edge in G.)

   Your algorithms should modify M so that it is still a maximum matching, faster than recomputing a maximum matching from scratch.

5. You are applying to participate in this year's Trial of the Pyx, the annual ceremony where samples of all British coinage are tested, to ensure that they conform as strictly as possible to legal standards. As a test of your qualifications, your interviewer at the Worshipful Company of Goldsmiths has given you a bag of n commemorative Alan Turing half-guinea coins, exactly two of which are counterfeit. One counterfeit coin is very slightly lighter than a genuine Turing; the other is very slightly heavier. Together, the two counterfeit coins have *exactly* the same weight as two genuine coins. Your task is to identify the two counterfeit coins.

   The weight difference between the real and fake coins is too small to be detected by anything other than the Royal Pyx Coin Balance. You can place any two disjoint sets of coins in each of the Balance's two pans; the Balance will then indicate which of the two subsets has larger total weight, or that the two subsets have the same total weight. Unfortunately, each use of the Balance requires completing a complicated authorization form (in triplicate), submitting a blood sample, and scheduling the Royal Bugle Corps, so you *really* want to use the Balance as few times as possible.

   (a) Suppose you *randomly* choose $n/2$ of your n coins to put on one pan of the Balance, and put the remaining $n/2$ coins on the other pan. What is the probability that the two subsets have equal weight?

   (b) Describe and analyze a randomized algorithm to identify the two fake coins. What is the expected number of times your algorithm uses the Balance? To simplify the algorithm, you may assume that n is a power of 2.

6. Suppose you are given a set L of n line segments in the plane, where each segment has one endpoint on the vertical line $x = 0$ and one endpoint on the vertical line $x = 1$, 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.

Some Useful Inequalities

Let $X = \sum_{i=1}^{n} X_i$, where each X_i is a $0/1$ random variable, and let $\mu = \mathrm{E}[X]$.

- **Markov's Inequality:** $\Pr[X \ge x] \le \mu/x$ for all $x > 0$.
- **Chebyshev's Inequality:** If $X_1, X_2, \ldots, X_n$ are pairwise independent, then for all $\delta > 0$:

$$\Pr[X \ge (1+\delta)\mu] < \frac{1}{\delta^2\mu} \quad \text{and} \quad \Pr[X \le (1-\delta)\mu] < \frac{1}{\delta^2\mu}$$

- **Chernoff Bounds:** If $X_1, X_2, \ldots, X_n$ are fully independent, then for all $0 < \delta \le 1$:

$$\Pr[X \ge (1+\delta)\mu] \le \exp\left(-\delta^2\mu/3\right) \quad \text{and} \quad \Pr[X \le (1-\delta)\mu] \le \exp\left(-\delta^2\mu/2\right)$$

Some Useful Algorithms

- **Random(k):** Returns an element of $\{1, 2, \ldots, k\}$, chosen independently and uniformly at random, in $O(1)$ time. For example, Random(2) can be used for a fair coin flip.
- **Ford and Fulkerson's maximum flow algorithm:** Returns a maximum (s, t)-flow f^* in a given flow network in ***$O(E \cdot |f^*|)$ time***. If all input capacities are integers, then all output flow values are also integers.
- **Orlin's maximum flow algorithm:** Returns a maximum (s, t)-flow in a given flow network in ***$O(VE)$ time***. If all input capacities are integers, then all output flow values are also integers.
- **Orlin's minimum-cost flow algorithm:** Returns a minimum-cost flow in a given flow network in ***$O(E^2 \log^2 V)$ time***. If all input capacities, costs, and balances are integers, then all output flow values are also integers.

Some Useful NP-hard Problems:

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?

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

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.

HydraulicPress: And here ve go!

Common Grading Rubrics

(For problems out of 10 points)

General Principles:

- 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. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.
- A clear, correct, and correctly analyzed algorithm, no matter how slow, is always worth more than “I don’t know”. An incorrect algorithm, no matter how fast, may be worth nothing.
- Proofs of correctness are required on exams if and only if we explicitly ask for them.

Dynamic Programming:

- 3 points for a **clear English specification** of the underlying recursive function = 2 for describing the function itself + 1 for describing how to call the function to get your final answer. We want an English description of the underlying recursive *problem*, not just the algorithm/recurrence. In particular, your description should specify precisely the role of each input parameter. **No credit for the rest of the problem if the English description is is missing; this is a Deadly Sin.**
- 4 points for correct recurrence = 1 for base case(s) + 3 for recursive case(s). **No credit for iterative details if the recursive case(s) are incorrect.**
- 3 points for iterative details = 1 for memoization structure + 1 for evaluation order + 1 for time analysis. Complete iterative pseudocode is *not* required for full credit.

Graph Reductions:

- 4 points for a complete description of the relevant graph, including vertices, edges (including whether directed or undirected), numerical data (weights, lengths, capacities, costs, balances, and the like), source and target vertices, and so on. If the graph is part of the original input, just say that.
- 4 points for other details of the reduction, including how to build the graph from the original input, the precise problem to be solved on the graph, the precise algorithm used to solve that problem, and how to extract your final answer from the output of that algorithm.
- 2 points for running time of the overall algorithm, expressed as a function of the original input parameters, *not* just the number of vertices and edges in the graph.

NP-hardness Proofs:

- 3 points for a complete description of the reduction, including an appropriate NP-hard problem to reduce from, how to transform the input, and how to transform the output.
- 6 points for the proof of correctness = 3 for the “if” part + 3 for the “only if” part.
- 1 points for “polynomial time”.

CS/ECE 374 A ✧ Spring 2018

Homework 0

Due Tuesday, January 23, 2018 at 8pm

- **Each student must submit individual solutions for this homework.** For all future homeworks, groups of up to three students can submit joint solutions.
- **Submit your solutions electronically to Gradescope as PDF files.** Submit a separate PDF file for each numbered problem. If you plan to typeset your solutions, please use the LaTeX solution template on the course web site. If you must submit scanned handwritten solutions, please use a black pen on blank white paper and a high-quality scanner app (or an actual scanner).
- You are ***not*** required to sign up on Gradescope or Piazza with your real name and your illinois.edu email address; you may use any email address and alias of your choice. However, to give you credit for the homework, we need to know who Gradescope thinks you are. **Please fill out the web form linked from the course web page.**

☞ Some important course policies ☜

- **You may use any source at your disposal**—paper, electronic, or human—but you ***must*** cite ***every*** source that you use, and you ***must*** write everything yourself in your own words. See the academic integrity policies on the course web site for more details.
- The answer ***"I don't know"*** (and *nothing* else) is worth 25% partial credit on any required problem or subproblem on any homework or exam. We will accept synonyms like "No idea" or "WTF" or "\(ó_ò)/", but you must write *something*.

  On the other hand, only the homework problems you submit actually contribute to your overall course grade, so submitting "I don't know" for an entire numbered homework problem will almost certainly hurt your grade more than submitting nothing at all.
- **Avoid the Three Deadly Sins!** Any homework or exam solution that breaks any of the following rules will be given an ***automatic zero***, unless the solution is otherwise perfect. Yes, we really mean it. We're not trying to be scary or petty (Honest!), but we do want to break a few common bad habits that seriously impede mastery of the course material.
  - Always give complete solutions, not just examples.
  - Always declare all your variables, in English. In particular, always describe the specific problem your algorithm is supposed to solve.
  - Never use weak induction.

See the course web site for more information.

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

1. The famous Basque computational arborist Gorka Oihanean has a favorite 26-node binary tree, in which each node is labeled with a letter of the alphabet. Intorder and postorder traversals of his tree visits the nodes in the following orders:

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

   (a) List the nodes in Professor Oihanean's tree according to a preorder traversal.

   (b) Draw Professor Oihanean's tree.

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

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

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

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

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

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

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

   You may assume without proof that $|x \bullet y| = |x| + |y|$, or any other 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 length $|w|$, concatenation $\bullet$, and the *swap* function. Do not appeal to intuition!

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

   - The empty string ε is in L.
   - For any string x in L, the string $0x$ is also in L.
   - For any strings x and y in L, the string $1x1y$ is also in L.
   - These are the only strings in L.

   (a) Prove that the string 101110101101011 is in L.

   (b) Prove that every string $w \in L$ contains an even number of 1s.

   (c) Prove that every string $w \in \{0,1\}^*$ with an even number of 1s is a member of L.

   Let $\#(a, w)$ denote the number of times symbol a appears in string w; for example,

   $$\#(0, 101110101101011) = 5 \quad \text{and} \quad \#(1, 101110101101011) = 10.$$

   You may assume without proof that $\#(a, uv) = \#(a, u) + \#(a, v)$ for any symbol a and any strings u and v, or any other result proved in class, in lab, or in the lecture notes. Otherwise, your proofs must be formal and self-contained.

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

Solved Problems

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

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

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

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

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

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

You may assume without proof the following statements for all strings x, y, and z:

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

Solution:

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

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

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

(b) Let w be an arbitrary palindrome.
Assume that $x = x^R$ for every palindrome x such that $|x| < |w|$.
There are three cases to consider (mirroring the definition of "palindrome"):

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

$x \in P$. In this case, we have

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

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

Rubric: 4 points: standard induction rubric (scaled)

(c) Let w be an arbitrary string such that $w = w^R$.

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

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

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

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

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

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

  We conclude that w is a palindrome by definition.

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

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

Standard induction rubric. For problems worth 10 points:

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

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

CS/ECE 374 A ✧ Spring 2018

Homework 1

Due Tuesday, January 30, 2018 at 8pm

Starting with this homework, groups of up to three people can submit joint solutions. Each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the Gradescope names and email addresses of each group member. In addition, whoever submits the homework must tell Gradescope who their other group members are.

1. For each of the following languages over the alphabet $\{0,1\}$, give a regular expression that describes that language, and *briefly* argue why your expression is correct.

   (a) All strings except 001.

   (b) All strings that end with the suffix 001001.

   (c) All strings that contain the substring 001.

   (d) All strings that contain the subsequence 001.

   (e) All strings that do not contain the substring 001.

   (f) All strings that do not contain the subsequence 001.

2. Let L denote the set of all strings in $\{0,1\}^*$ that contain all four strings 00, 01, 10, and 11 as substrings. For example, the strings 110011 and 01001011101001 are in L, but the strings 00111 and 1010101 are not.

   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 accept states A, and the transition function δ. Do not attempt to *draw* your DFA; the smallest DFA for this language has 20 states, which is too many for a drawing to be understandable.

   Argue that your machine accepts every string in L and nothing else, by explaining what each state in your DFA *means*. Formal descriptions without English explanations will receive no credit, even if they are correct. (See the standard DFA rubric for more details.)

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

3. Let L be the set of all strings in $\{0,1\}^*$ that contain *exactly one* occurrence of the substring `010`.

   (a) Give a regular expression for L, and briefly argue why your expression is correct. *[Hint: You may find the shorthand notation $A^+ = AA^*$ useful.]*

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

   Argue that your machine accepts every string in L and nothing else, by explaining what each state in your DFA *means*. 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. Drawings or formal descriptions without English explanations will receive no credit, even if they are correct.

Solved problem

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

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

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

   /⋆⋆⋆/ //⋄//⋄↲ /⋆///⋄⋆⋄↲⋆⋆/ /⋆⋄//⋄↲⋄⋆/

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

   /⋆/ //⋄//⋄↲⋄↲ /⋆⋄/⋆⋄⋆/⋄⋆/

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

   Solution:

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

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

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

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

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

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

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

Rubric: Standard regular expression rubric

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

Solution:

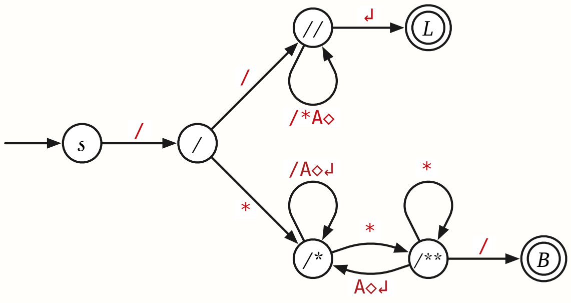
$$\Big(\diamond + \text{↲} + \texttt{//}(\texttt{/} + \star + \texttt{A} + \diamond)^*\text{↲} + \texttt{/}\star(\texttt{/} + \texttt{A} + \diamond + \text{↲} + \star\star^*(\texttt{A} + \diamond + \text{↲}))^*\star\star^*\texttt{/}\Big)^*$$

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

(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:

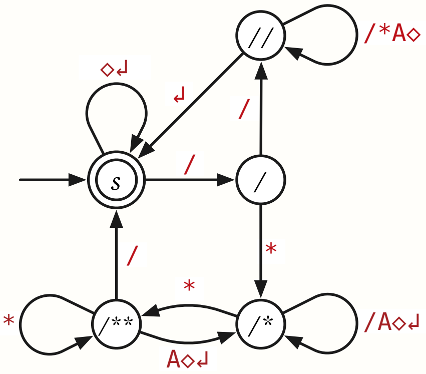
- s — We have not read anything.
- / — We just read the initial /.
- // — We are reading a line comment.
- L — We have just read a complete line comment.
- /* — We are reading a block comment, and we did not just read a ⋆ after the opening /⋆.
- /** — We are reading a block comment, and we just read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

■

Rubric: Standard DFA design rubric

(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

Standard regular expression rubric. For problems worth 10 points:

- 2 points for a syntactically correct regular expression.
- **Homework only:** 4 points for a *brief* English explanation of your regular expression. This is how you argue that your regular expression is correct.
  - **Deadly Sin ("Declare your variables."): No credit for the problem if the English explanation is missing, *even if the regular expression is correct*.**
  - For longer expressions, you should explain each of the major components of your expression, and separately explain how those components fit together.
  - We do not want a *transciption*; 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 longer than necessary may be penalized. Regular expressions that are *significantly* longer than necessary may get no credit at all.

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.
  - **For drawings:** Use an arrow from nowhere to indicate s, and doubled circles to indicate accepting states A. If $A = \varnothing$, say so explicitly. If your drawing omits a reject state, say so explicitly. **Draw neatly!** If we can't read your solution, we can't give you credit for it,.
  - **For text descriptions:** You can describe the transition function either using a 2d array, using mathematical notation, or using an algorithm.
  - **For product constructions:** You must give a complete description of the states and transition functions of the DFAs you are combining (as either drawings or text), together with the accepting states of the product DFA.
- **Homework only:** 4 points for *briefly* and correctly explaining the purpose of each state *in English*. This is how you justify that your DFA is correct.
  - **Deadly Sin ("Declare your variables."): No credit for the problem if the English description is missing, *even if the DFA is correct*.**
  - For product constructions, explaining the states in the factor DFAs is sufficient.
- 4 points for correctness. (8 points on exams, with all penalties doubled)
  - −1 for a single mistake: a single misdirected transition, a single missing or extra accept 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.
- DFA drawings with too many states may be penalized. DFA drawings with *significantly* too many states may get no credit at all.
- Half credit for describing an NFA when the problem asks for a DFA.

CS/ECE 374 A ✧ Spring 2018

Homework 2

Due Tuesday, February 6, 2018 at 8pm

---

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

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

   (b) $\left\{w \in (\mathtt{0}+\mathtt{1})^* \;\middle|\; \#(\mathtt{0},w) \le 2\cdot\#(\mathtt{1},w)\right\}$

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

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

2. For each of the following 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) $(\mathtt{0}+\mathtt{1})^*\cdot\mathtt{0001}\cdot(\mathtt{0}+\mathtt{1})^*$

   (b) $(\mathtt{1}+\mathtt{01}+\mathtt{001})^*\mathtt{0}^*$

3. 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). Recall that Σ^+ denotes the set of all *nonempty* strings over Σ.

   (a) Strings in which the substrings $\mathtt{00}$ and $\mathtt{11}$ do not appear the same number of times. For example, $\mathtt{1100011} \notin L$ because both substrings appear twice, but $\mathtt{01000011} \in L$.

   (b) Strings in which the substrings $\mathtt{01}$ and $\mathtt{10}$ do not appear the same number of times. For example, $\mathtt{1100011} \notin L$ because both substrings appear twice, but $\mathtt{01000011} \in L$.

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

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

   [Hint: Exactly two of these languages are regular. Strings can be empty.]

Solved problem

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

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

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

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

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

- If x contains a $\mathtt{0}$, then we can write $w = \mathtt{01}^n\mathtt{0}y$ for some integer n and some string y; but this is impossible, because the prefix $\mathtt{01}^n\mathtt{0}$ is a palindrome of length at least 2.
- Otherwise, $x = \mathtt{1}^n$ for some integer n. Every prefix of w has the form $\mathtt{01}^m$ for some integer $m \le n$. Any palindrome that starts with $\mathtt{0}$ must end with $\mathtt{0}$, so the only palindrome prefixes of w are ε and $\mathtt{0}$, both of which have length less than 2.

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

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

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

Solution: Not regular. Call this language L_b.

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

Let x and y be arbitrary distinct strings in F.

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

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

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

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

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

Because F is infinite, L_b cannot be regular. ■

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

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

Solution: Not regular. Call this language L_c.

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

Let x and y be arbitrary distinct strings in F.

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

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

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

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

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

Because F is infinite, L_c cannot be regular. ■

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

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

Solution: Regular. Call this language L_d.

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

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

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

Solution: Regular. Call this language L_d.

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

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

■

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

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

Standard fooling set rubric. For problems worth 5 points:

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

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

CS/ECE 374 A ✧ Spring 2018

Homework 3

Due Tuesday, February 13, 2018 at 8pm

---

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

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

   (b) $\left\{w \in (\mathtt{0}+\mathtt{1})^* \;\middle|\; \#(\mathtt{0},w) \le 2\cdot\#(\mathtt{1},w)\right\}$

   (c) Strings in which the substrings `00` and `11` appear the same number of times. For example, $\mathtt{1100011} \in L$ because both substrings appear twice, but $\mathtt{01000011} \notin L$. *[Hint: This is the complement of the language you considered in HW2.]*

2. Let $inc\colon \{\mathtt{0},\mathtt{1}\}^* \to \{\mathtt{0},\mathtt{1}\}^*$ denote the *increment* function, which transforms the binary representation of an arbitrary integer n into the binary representation of $n+1$, truncated to the same number of bits. For example:

   $$inc(\mathtt{0010}) = \mathtt{0011} \qquad inc(\mathtt{0111}) = \mathtt{1000} \qquad inc(\mathtt{1111}) = \mathtt{0000} \qquad inc(\varepsilon) = \varepsilon$$

   Let $L \subseteq \{\mathtt{0},\mathtt{1}\}^*$ be an arbitrary regular language. Prove that $inc(L) = \{inc(w) \mid w \in L\}$ is also regular.

3. A ***shuffle*** of two strings x and y is any string obtained by interleaving the symbols in x and y, but keeping them in the same order. For example, the following strings are shuffles of `HOGWARTS` and `BRAKEBILLS`:

   `HOGWARTSBRAKEBILLS`   `HOGBRAKEWARTSBILLS`   `BHROAGKWEABRITLSLS`

   More formally, a string z is a shuffle of strings x and y if and only if (at least) one of the following conditions holds:

   - $x = \varepsilon$ and $z = y$
   - $y = \varepsilon$ and $z = x$
   - $x = ax'$ and $z = az'$ where z' is a shuffle of x' and y
   - $y = ay'$ and $z = az'$ where z' is a shuffle of x and y'

   For any two languages L and L' over the alphabet $\{\mathtt{0},\mathtt{1}\}$, define

   $$shuffles(L,L') = \left\{z \in \{\mathtt{0},\mathtt{1}\}^* \;\middle|\; z \text{ is a shuffle of some } x \in L \text{ and } y \in L'\right\}$$

   Prove that if L and L' are regular languages, then $shuffles(L,L')$ is also a regular language.

Solved problem

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

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

$$Q' = (Q \times Q \times Q) \cup \{s'\}$$

$$s' \text{ is an explicit state in } Q'$$

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

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

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

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

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

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 ✧ Spring 2018

Homework 4

Due Tuesday, February 27, 2018 at 8pm

---

1. At the end of the second act of the action blockbuster *Fast and Impossible XIII¾: Guardians of Expendable Justice Reloaded*, the villainous Dr. Metaphor hypnotizes the entire Hero League/Force/Squad, arranges them in a long line at the edge of a cliff, and instructs each hero to shoot the closest taller heroes to their left and right, at a prearranged signal.

   Suppose we are given the heights of all n heroes, in clockwise order around the circle, in an array $Ht[1..n]$. (To avoid salary arguments, the producers insisted that no two heroes have the same height.) Then we can compute the Left and Right targets of each hero in $O(n^2)$ time using the following algorithm.

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

   (a) Describe a divide-and-conquer algorithm that computes the output of WhoTargetsWhom in $O(n \log n)$ time.

   (b) Prove that at least $\lfloor n/2 \rfloor$ of the n heroes are targets. That is, prove that the output arrays $R[0..n-1]$ and $L[0..n-1]$ contain at least $\lfloor n/2 \rfloor$ distinct values (other than None).

   (c) Alas, Dr. Metaphor's diabolical plan is successful. At the prearranged signal, all the heroes simultaneously shoot their targets, and all targets fall over the cliff, apparently dead. Metaphor repeats his dastardly experiment over and over; after each massacre, he forces the remaining heroes to choose new targets, following the same algorithm, and then shoot their targets at the next signal. Eventually, only the shortest member of the Hero Crew/Alliance/Posse is left alive.[1]

   Describe an algorithm that computes the number of rounds before Dr. Metaphor's deadly process finally ends. For full credit, your algorithm should run in $O(n)$ time.

[1] In the thrilling final act, Retcon the Squirrel, the last surviving member of the Hero Team/Group/Society, saves everyone by traveling back in time and retroactively replacing the other $n-1$ heroes with lifelike balloon sculptures.

2. Describe and analyze a recursive algorithm to reconstruct an arbitrary binary tree, given its preorder and inorder node sequences as input.

   The input to your algorithm is a pair of arrays $Pre[1..n]$ and $In[1..n]$, each containing a permutation of the same set of n distinct symbols. Your algorithm should return an n-node binary tree whose nodes are labeled with those n symbols (or an error code if no binary tree is consistent with the input arrays). You solved an instance of this problem in Homework 0.

3. Suppose we are given a set S of n items, each with a *value* and a *weight*. For any element $x \in S$, we define two subsets:

   - $S_{<x}$ is the set of all elements of S whose value is smaller than the value of x.
   - $S_{>x}$ is the set of all elements of S whose value is larger than the value of x.

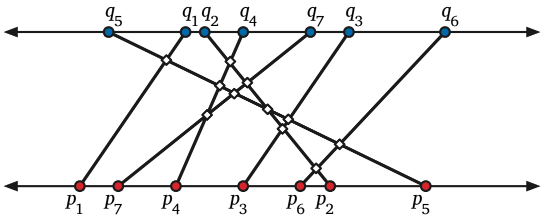
   For any subset $R \subseteq S$, let $w(R)$ denote the sum of the weights of elements in R. The ***weighted median*** of R is any element x such that $w(S_{<x}) \leq w(S)/2$ and $w(S_{>x}) \leq w(S)/2$.

   Describe and analyze an algorithm to compute the weighted median of a given weighted set in $O(n)$ time. Your input consists of two unsorted arrays $S[1..n]$ and $W[1..n]$, where for each index i, the ith element has value $S[i]$ and weight $W[i]$. You may assume that all values are distinct and all weights are positive.

   [Hint: Use or modify the linear-time selection algorithm described in class on Thursday.]

Solved problem

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



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

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

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

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

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

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

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

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

Rubric: This is enough for full credit.

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

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

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

```
MergeAndCount2($A[1\,..\,n], m$):
  $i \gets 1$;  $j \gets m+1$;  $total \gets 0$
  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 + 3 for divide (split and recurse) + 3 for conquer (merge and count) + 2 for time analysis. Max 3 points for a correct $O(n^2)$-time algorithm. This is neither the only way to correctly describe this algorithm nor the only correct $O(n \log n)$-time algorithm. No proof of correctness is required.

Notice that each boxed algorithm is preceded by an English description of the task that algorithm performs. **Omitting these descriptions is a Deadly Sin.**

CS/ECE 374 A ✧ Spring 2018

Homework 5

Due Tuesday, March 6, 2018 at 8pm

1. 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 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 $Score[k]$ points, but then you will be physically unable to dance for the next $Wait[k]$ songs (that is, you cannot dance to songs $k+1$ through $k+Wait[k]$). 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 $Score[1..n]$ and $Wait[1..n]$.

2. Suppose you are given a NFA $M = (\{0,1\}, Q, s, A, \delta)$ 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 indicated by a boolean array $A[1..k]$, where $A[q] = \text{True}$ 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 $inDelta[p, a, q] = \text{True}$ if and only if $q \in \delta(p, a)$.

   Finally, the input string is given as an array $w[1..n]$. 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!!]*

3. Recall that a ***palindrome*** is any string that is exactly the same as its reversal, like the empty srting, or `I`, or `DEED`, or `RACECAR`, or `AMANAPLANACATACANALPANAMA`.

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

   `BUB` • `BASEESAB` • `ANANA`
   `B` • `U` • `BB` • `ASEESA` • `B` • `ANANA`
   `BUB` • `B` • `A` • `SEES` • `ABA` • `N` • `ANA`
   `B` • `U` • `BB` • `A` • `S` • `EE` • `S` • `A` • `B` • `A` • `NAN` • `A`
   `B` • `U` • `B` • `B` • `A` • `S` • `E` • `E` • `S` • `A` • `B` • `A` • `N` • `A` • `N` • `A`

   (a) Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example:

   - Given the string `PALINDROME`, your algorithm should return the integer 10.
   - Given the string `BUBBASEESABANANA`, your algorithm should return the integer 3.
   - Given the string `RACECAR`, your algorithm should return the integer 1.

   (b) 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

   `BUB` • `B` • `ALA` • `SEES` • `ABA` • `N` • `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.

Solved Problem

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

$$\texttt{BANANA}{}_{\texttt{ANANAS}} \qquad \texttt{BAN}{}_{\texttt{ANA}}\texttt{ANA}{}_{\texttt{NAS}} \qquad \texttt{B}{}_{\texttt{AN}}\texttt{AN}{}_{\texttt{A}}\texttt{A}{}_{\texttt{NA}}\texttt{NA}{}_{\texttt{S}}$$

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

$${}_{\texttt{PRO}}\texttt{D}{}_{\texttt{G}}\texttt{Y}{}_{\texttt{R}}\texttt{NAM}{}_{\texttt{AMMI}}\texttt{I}{}_{\texttt{N}}\texttt{C}{}_{\texttt{G}} \qquad \texttt{DY}{}_{\texttt{PRO}}\texttt{N}{}_{\texttt{G}}\texttt{A}{}_{\texttt{R}}\texttt{M}{}_{\texttt{AMM}}\texttt{IC}{}_{\texttt{ING}}$$

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

Solution: We define a boolean function $\mathit{Shuf}(i, j)$, which is True if and only if the prefix $C[1..i+j]$ is a shuffle of the prefixes $A[1..i]$ and $B[1..j]$. This function satisfies the following recurrence:

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

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

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

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

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

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

Standard dynamic programming rubric. For problems worth 10 poins:

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 point for a clear English description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) **Deadly Sin: Automatic zero if the English description is missing.**
  - + 1 point for stating how to call your function to get the final answer.
  - + 1 point for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 points for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 4 points for details of the dynamic programming algorithm
  - + 1 point for describing the memoization data structure
  - + 2 points for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested loops, be sure to specify the nesting order.
  - + 1 point for time analysis
- It is *not* necessary to state a space bound.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem 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 still need to describe the underlying recursive function in English.)
- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we have included them, significantly more students turned in algorithms that meet the target time bound but didn't work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).

CS/ECE 374 A ✧ Spring 2018

Homework 6

Due Tuesday, March 13, 2018 at 8pm

1. Suppose you are given an array $A[1..n]$ of positive integers, each of which is colored either red or blue. An *increasing back-and-forth subsequence* is an sequence of indices $I[1..\ell]$ with the following properties:

   - $1 \le I[j] \le n$ for all j.
   - $A[I[j]] < A[I[j+1]]$ for all $j < \ell$.
   - If $A[I[j]]$ is red, then $I[j+1] > I[j]$.
   - If $A[I[j]]$ is blue, then $I[j+1] < I[j]$.

   Less formally, suppose we start with a token on some integer $A[j]$, and then repeatedly move the token Left (if it's on a bLue square) or Right (if it's on a Red square), always moving from a smaller number to a larger number. Then the sequence of token positions is an increasing back-and-forth subsequence.

   Describe and analyze an efficient algorithm to compute the length of the longest increasing back-and-forth subsequence of a given array of n red and blue integers. For example, given the input array

1	1	0	2	5	9	6	6	4	5	8	9	7	7	3	2	3	8	4	0
🟥	🟦	🟥	🟦	🟦	🟥	🟦	🟥	🟦	🟥	🟥	🟦	🟦	🟥	🟥	🟥	🟦	🟦	🟥	🟦
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

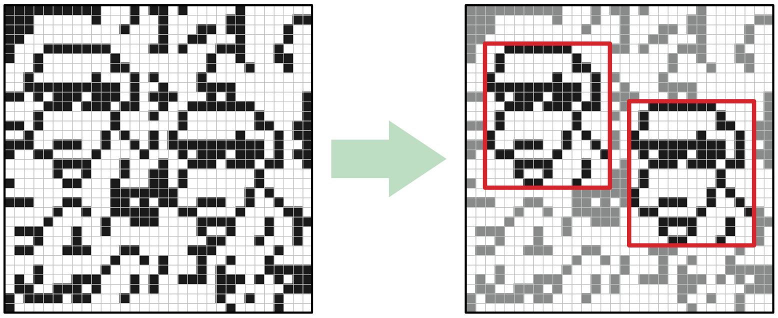
   your algorithm should return the integer 9, which is the length of the following increasing back-and-forth subsequence:

0	1	2	3	4	6	7	8	9
🟦	🟥	🟥	🟦	🟦	🟥	🟦	🟥	🟦
20	1	16	17	9	8	13	11	12

   (The small numbers are indices into the input array.)

2. Describe and analyze an algorithm that finds the largest rectangular pattern that appears more than once in a given bitmap. Your input is a two-dimensional array $M[1..n, 1..n]$ of bits; your output is the area of the repeated pattern. (The two copies of the pattern might overlap, but must not actually coincide.)

   For example, given the bitmap shown on the left in the figure below, your algorithm should return $15 \times 13 = 195$, because the same 15×13 doggo appears twice, as shown on the right, and this is the largest such pattern.



3. ***AVL trees*** were the earliest self-balancing balanced binary search trees, first described in 1962 by Georgy Adelson-Velsky and Evgenii Landis. An AVL tree is a binary search tree where for every node v, the height of the left subtree of v and the height of the right subtree of v differ by at most 1.

   Describe and analyze an efficient algorithm to construct an optimal AVL tree for a given set of keys and frequencies. Your input consists of a sorted array $A[1..n]$ of search keys and an array $f[1..n]$ of frequency counts, where $f[i]$ is the number of searches for $A[i]$. Your task is to construct an AVL tree for the given keys such that the total cost of all searches is as small as possible. This is exactly the same cost function that we considered in Thursday's class; the only difference is that the output tree must satisfy the AVL balance constraint.

   [Hint: You do ***not*** *need to know or use the insertion and deletion algorithms that keep the AVL tree balanced.]*

Solved Problems

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

$$S \rightarrow \varepsilon \mid \mathbf{(}S\mathbf{)} \mid \mathbf{[}S\mathbf{]} \mid SS$$

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

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

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 $A[i] \in \{\mathbf{(}, \mathbf{)}, \mathbf{[}, \mathbf{]}\}$ for every index i.

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

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

Here $\boldsymbol{A[i] \sim A[k]}$ indicates that $A[i]$ and $A[k]$ are matching delimiters: Either $A[i] = \mathbf{(}$ and $A[k] = \mathbf{)}$ or $A[i] = \mathbf{[}$ and $A[k] = \mathbf{]}$.

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

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

■

Rubric: 10 points, standard dynamic programming rubric

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

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

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

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

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

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

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

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

```
ComputeMaxFun(v):
    $v.yes \leftarrow v.fun$
    $v.no \leftarrow 0$
    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 $\mathit{MaxFun}(v)$ 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:

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

(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.

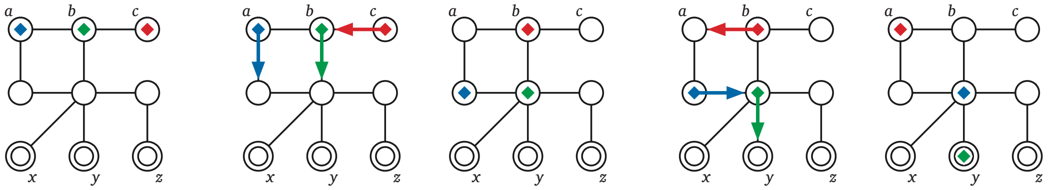
CS/ECE 374 A ✧ Spring 2018

Homework 7

Due Tuesday, March 27, 2018 at 8pm
(*after* Spring Break)

---

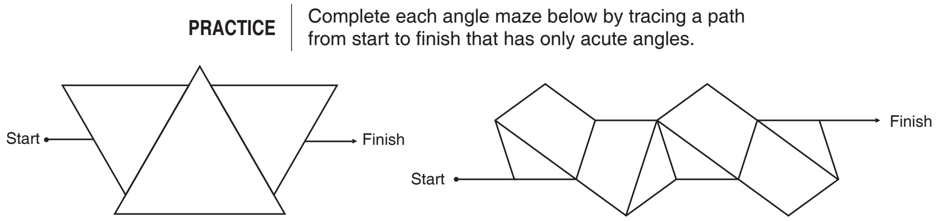
1. Consider the following solitaire game, played on a connected undirected graph G. Initially, tokens are placed on three start vertices a, b, c. In each turn, you *must* move *all three* tokens, by moving each token along an edge from its current vertex to an adjacent vertex. At the end of each turn, the three tokens *must* lie on three different vertices. Your goal is to move the tokens onto three goal vertices x, y, z; it does not matter which token ends up on which goal vertex.



The initial configuration of the puzzle and the first two turns of a solution.

Describe and analyze an algorithm to determine whether this puzzle is solvable. Your input consists of the graph G, the start vertices a, b, c, and the goal vertices x, y, z. Your output is a single bit: True or False. *[Hint: You've seen this sort of thing before.]*

2. The following puzzles appear in my daughter's elementary-school math workbook.[1]


PRACTICE | Complete each angle maze below by tracing a path from start to finish that has only acute angles.

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

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 first maze above has 13 vertices and 15 edges. You are also given two vertices Start and Finish.

Your algorithm should return True if G contains a walk from Start to Finish that has only acute angles, and False otherwise. Formally, a walk through G is valid if, for any two consecutive edges $u \to v \to w$ in the walk, either $\angle uvw = \pi$ or $0 < \angle uvw < \pi/2$. Assume you have a subroutine that can determine in $O(1)$ time whether two segments with a common vertex define a straight, obtuse, right, or acute angle.

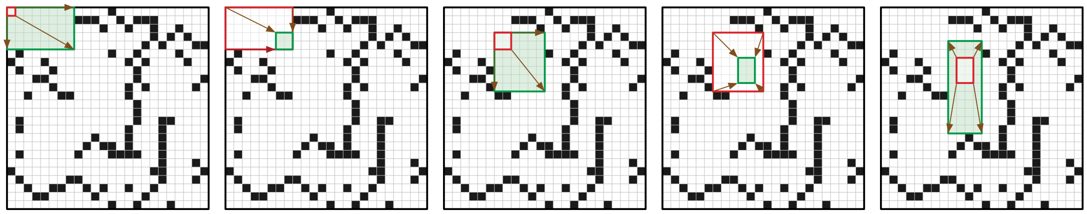
[1] Jason Batterson and Shannon Rogers, *Beast Academy Math: Practice 3A*, 2012. See https://www.beastacademy.com/resources/printables.php for more examples.

3. ***Rectangle Walk*** is a new abstract puzzle game, available for only 99¢ on Steam, iOS, Android, Xbox One, Playstation 5, Nintendo Wii U, Atari 2600, Palm Pilot, Commodore 64, TRS-80, Sinclair ZX-1, DEC PDP-8, ILLIAC V, Zuse Z3, Duramesc, Odhner Arithmometer, Analytical Engine, Jacquard Loom, Horologium mirabile Lundense, Leibniz Stepped Reckoner, Antikythera Mechanism, and Pile of Sticks.

   The game is played on an $n \times n$ grid of black and white squares. The player moves a rectangle through this grid, subject to the following conditions:

   - The rectangle must be aligned with the grid; that is, the top, bottom, left, and right coordinates must be integers.
   - The rectangle must fit within the $n \times n$ grid, and it must contain at least one grid cell.
   - The rectangle must not contain a black square.
   - In a single move, the player can replace the current rectangle r with any rectangle r' that either contains r or is contained in r.

   Initially, the player's rectangle is a 1×1 square in the upper right corner. The player's goal is to reach a 1×1 square in the bottom left corner using as few moves as possible.



The first five steps in a Rectangle Walk.

   Describe and analyze an algorithm to compute the length of the shortest Rectangle Walk in a given bitmap. Your input is an array $M[1..n, 1..n]$, where $M[i, j] = 1$ indicates a black square and $M[i, j] = 0$ indicates a white square. You can assume that a valid rectangle walk exists; in particular, $M[1, 1] = 0$ and $M[n, n] = 0$. For example, given the bitmap shown above, (I think) your algorithm should return the integer 18.

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 (for the sequence of operations listed above).

Solution: Let A, B, C denote the capacities of the three jars. We reduce the problem to breadth-first search in the following directed graph:

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

- $\left\{\begin{matrix}(a+b,0,c) & \text{if } a+b\le A\\ (A,a+b-A,c) & \text{if } a+b\ge A\end{matrix}\right\}$ — pouring from jar 2 into jar 1
- $\left\{\begin{matrix}(a,0,b+c) & \text{if } b+c\le C\\ (a,b+c-C,C) & \text{if } b+c\ge C\end{matrix}\right\}$ — pouring from jar 2 into jar 3
- $\left\{\begin{matrix}(a+c,b,0) & \text{if } a+c\le A\\ (A,b,a+c-A) & \text{if } a+c\ge A\end{matrix}\right\}$ — pouring from jar 3 into Jar 1
- $\left\{\begin{matrix}(a,b+c,0) & \text{if } b+c\le B\\ (a,B,b+c-B) & \text{if } b+c\ge B\end{matrix}\right\}$ — pouring from jar 3 into jar 2

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

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

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

Rubric: 10 points: standard graph reduction rubric (see next page)

- Brute force construction is fine.

−1 for calling Dijkstra instead of BFS

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

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

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

CS/ECE 374 A ✧ Spring 2018

Homework 8

Due Tuesday, April 3, 2018 at 8pm

This is the last homework before Midterm 2.

---

1. After moving to a new city, you decide to choose a walking route from your home to your new office. To get a good daily workout, your route must consist of an uphill path (for exercise) followed by a downhill path (to cool down), or just an uphill path, or just a downhill path.[1] (You'll walk the same path home, so you'll get exercise one way or the other.) But you also want the *shortest* path that satisfies these conditions, so that you actually get to work on time.

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

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

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

2. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero.

   (a) How could we delete an arbitrary vertex v from this graph, without changing the shortest-path distance between any other pair of vertices? Describe an algorithm that constructs a directed graph $G' = (V', E')$ with weighted edges, where $V' = V \setminus \{v\}$, and the shortest-path distance between any two nodes in G' is equal to the shortest-path distance between the same two nodes in G, in $O(V^2)$ time.

   (b) Now suppose we have already computed all shortest-path distances in G'. Describe an algorithm to compute the shortest-path distances in the original graph G from v to every other vertex, and from every other vertex to v, all in $O(V^2)$ time.

   (c) Combine parts (a) and (b) into another all-pairs shortest path algorithm that runs in $O(V^3)$ time. (The resulting algorithm is *almost* the same as Floyd-Warshall!)

[1] A *hill* is an area of land that extends above the surrounding terrain, usually at a fairly gentle gradient. Like a building, but smoother and made of dirt and rock and trees instead of steel and concrete. It's hard to explain.

3. The first morning after returning from a glorious spring break, Alice wakes to discover that her car won't start, so she has to get to her classes at Sham-Poobanana University by public transit. She has a complete transit schedule for Poobanana County. The bus routes are represented in the schedule by a directed graph G, whose vertices represent bus stops and whose edges represent bus routes between those stops. For each edge $u{\to}v$, the schedule records three positive real numbers:

   - $\ell(u{\to}v)$ is the length of the bus ride from stop u to stop v (in minutes)
   - $f(u{\to}v)$ is the first time (in minutes past 12am) that a bus leaves stop u for stop v.
   - $\Delta(u{\to}v)$ is the time between successive departures from stop u to stop v (in minutes).

   Thus, the first bus for this route leaves u at time $f(u{\to}v)$ and arrives at v at time $f(u{\to}v)+\ell(u{\to}v)$, the second bus leaves u at time $f(u{\to}v)+\Delta(u{\to}v)$ and arrives at v at time $f(u{\to}v)+\Delta(u{\to}v)+\ell(u{\to}v)$, the third bus leaves u at time $f(u{\to}v)+2\cdot\Delta(u{\to}v)$ and arrives at v at time $f(u{\to}v)+2\cdot\Delta(u{\to}v)+\ell(u{\to}v)$, and so on.

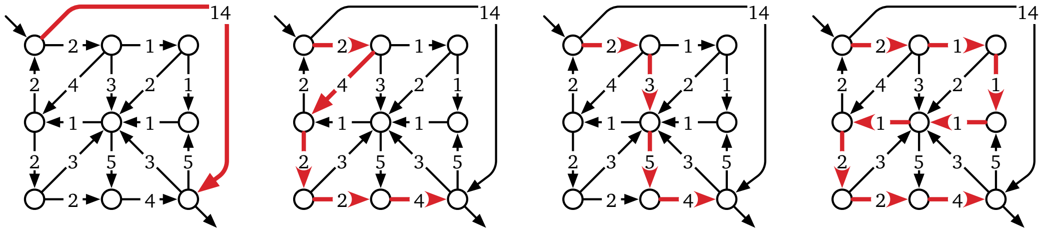
   Alice wants to leaves from stop s (her home) at a certain time and arrive at stop t (The See-Bull Center for Fake News Detection) as quickly as possible. If Alice arrives at a stop on one bus at the exact time that another bus is scheduled to leave, she can catch the second bus. Because she's a student at SPU, Alice can ride the bus for free, so she doesn't care how many times she has to change buses.

   Describe and analyze an algorithm to find the earliest time Alice can reach her destination. Your input consists of the directed graph $G=(V,E)$, the vertices s and t, the values $\ell(e), f(e), \Delta(e)$ for each edge $e\in E$, and Alice's starting time (in minutes past 12am).

   [Hint: In this rare instance, modifying the algorithm may be more efficient than modifying the input graph. Don't describe the algorithm from scratch; just describe your changes.]

Solved Problem

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



Four (of many) equal-length shortest paths.

Describe and analyze an algorithm to determine the *number* of shortest paths from a source vertex s to a target vertex t in an arbitrary directed graph G with weighted edges. You may assume that all edge weights are positive and that the 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 ✧ Spring 2018

Homework 9

Due Tuesday, April 17, 2018 at 8pm

---

1. For any integer k, the problem kSat is defined as follows:

   - Input: A boolean formula Φ in conjunctive normal form, with exactly k distinct literals in each clause.
   - Output: True if Φ has a satisfying assignment, and False otherwise.

   (a) Describe a polynomial-time reduction from **2**Sat to **3**Sat, and prove that your reduction is correct.

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

   (c) Why don't these results imply a polynomial-time algorithm for 3Sat?

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

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

   (a) Describe a polynomial-time reduction from SubsetSum to Partition.

   (b) Describe a polynomial-time reduction from Partition to SubsetSum.

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

3. *Pebbling* is a solitaire game played on an undirected graph G, where each vertex has zero or more *pebbles*. A single *pebbling move* removes two pebbles from some vertex v and adds one pebble to an arbitrary neighbor of v. (Obviously, v must have at least two pebbles before the move.) The PebbleClearing problem asks, given a graph $G = (V, E)$ and a pebble count $p(v)$ for each vertex v, whether is there a sequence of pebbling moves that removes all but one pebble. Prove that PebbleClearing is NP-hard.

Solved Problem

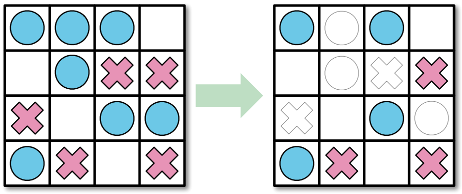
4. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions:

   (1) Every row contains at least one stone.

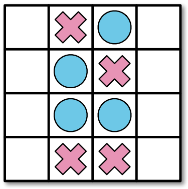
   (2) No column contains stones of both colors.

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

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



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

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

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

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

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

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

- If $x_j =$ True, remove all red stones from column j.
- If $x_j =$ False, remove all blue stones from column j.

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

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

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

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

This reduction clearly requires only polynomial time. ■

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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?

SteamedHams: Aurora borealis? At this time of year, at this time of day, in this part of the country, localized entirely within your kitchen? May I see it?

CS/ECE 374 A ✧ Spring 2018

Homework 10

Due Tuesday, April 24, 2018 at 8pm

This is the last graded homework before the final exam.

---

1. (a) 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.

   (b) 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.

2. Fix an alphabet $\Sigma = \{0, 1\}$. Prove that the following problems are NP-hard.[1]

   (a) Given a regular expression R over the alphabet Σ, is $L(R) \neq \Sigma^*$?

   (b) Given an NFA M over the alphabet Σ, is $L(M) \neq \Sigma^*$?

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

3. 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 $x \bullet y$ denotes the concatenation of strings x and y. Prove that the following language is undecidable.

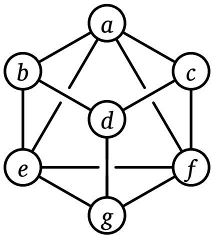
   $$\textsc{SelfSelfAccept} := \{\langle M \rangle \mid M \text{ accepts the string } \langle M \rangle \bullet \langle M \rangle\}$$

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

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

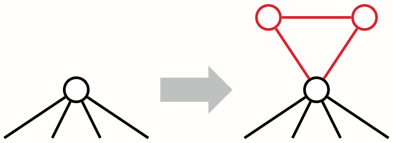
Solved Problem

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



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

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



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

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

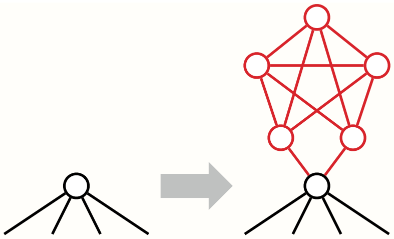
$\Longrightarrow$ Suppose G has a Hamiltonian cycle $v_1 \to v_2 \to \cdots \to v_n \to v_1$. We can construct a double-Hamiltonian tour of H by replacing each vertex v_i with the following walk:

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

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

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

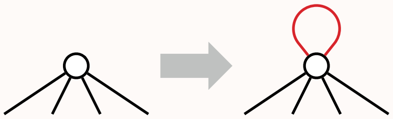
With more effort, we can construct a graph H that contains a double-Hamiltonian tour ***that traverses each edge of H at most once*** if and only if G contains a Hamiltonian cycle. For each vertex v in G we attach a more complex gadget containing five vertices and eleven edges, as shown on the next page.



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

■

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.

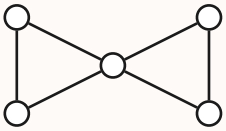


An incorrect vertex gadget.

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:

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

On the other hand, 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.

♣

Rubric: 10 points, standard polynomial-time reduction rubric

CS/ECE 374 A ✧ Spring 2018

“Homework” 11

“Due” Tuesday, May 1, 2018

This homework is optional. However, **similar undecidability questions may appear on the final exam**, so we still strongly recommend treating at least those questions as regular homework. Solutions will be released next Tuesday as usual.

---

1. Let M be a Turing machine, let w be an arbitrary input string, and let s be an integer. We say that ***M accepts w in space s*** if, given w as input, M accesses only the first s (or fewer) cells on its tape and eventually accepts.

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

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

   (b) Prove that the following language is undecidable:

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

2. Consider the following language:

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

   (a) Prove that $\textsc{Picky}$ is undecidable.

   (b) Sketch a Turing machine/algorithm that *accepts* $\textsc{Picky}$.

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 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, STRESSED^R = DESSERTS and WTF374^R = 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: You need #2 to prove #3, but you may find it easier to solve #3 first.]

To think about later: Let $\#(a, w)$ denote the number of times symbol a appears in string w. For example, $\#($X$,$WTF374$) = 0$ and $\#($0$,$000010101010010100$) = 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 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 all the 1s appear before any 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 $\Sigma = \{0, 1\}$. Describe briefly what each state in your DFAs *means*.

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 to keep the number of states small.

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 all the 1s appear before any 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 $\Sigma = \{\texttt{0}, \texttt{1}\}$. 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 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.

Work on these later:

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

*4. 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 $\mathtt{0}$s is exactly twice the number of $\mathtt{1}$s.

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

1. (a) Convert the regular expression $(\mathtt{0}^*\mathtt{1} + \mathtt{01}^*)^*$ into an NFA using Thompson's algorithm.

   (b) Convert the NFA you just constructed into a DFA using the incremental subset construction. Draw the resulting DFA. Your DFA should have four states, all reachable from the start state. (Some of these states are obviously equivalent, but keep them separate.)

   (c) **Think about later:** Convert the DFA you just constructed into a regular expression using Han and Wood's algorithm. You should *not* get the same regular expression you started with.

   (d) What is this language?

2. (a) Convert the regular expression $(\varepsilon + (\mathtt{0} + \mathtt{11})^*\mathtt{0})\mathtt{1}(\mathtt{11})^*$ into an NFA using Thompson's algorithm.

   (b) Convert the NFA you just constructed into a DFA using the incremental subset construction. Draw the resulting DFA. Your DFA should have six states, all reachable from the start state. (Some of these states are obviously equivalent, but keep them separate.)

   (c) **Think about later:** Convert the DFA you just constructed into a regular expression using Han and Wood's algorithm. You should *not* get the same regular expression you started with.

   (d) What is this language?

Let L be an arbitrary regular language.

1. Prove that the language $\mathit{insert}\mathtt{1}(L) := \{x\mathtt{1}y \mid xy \in L\}$ is regular.

   Intuitively, $\mathit{insert}\mathtt{1}(L)$ is the set of all strings that can be obtained from strings in L by inserting exactly one $\mathtt{1}$. For example, if $L = \{\varepsilon, \mathtt{OOK!}\}$, then $\mathit{insert}\mathtt{1}(L) = \{\mathtt{1}, \mathtt{1OOK!}, \mathtt{O1OK!}, \mathtt{OO1K!}, \mathtt{OOK1!}, \mathtt{OOK!1}\}$.

2. Prove that the language $\mathit{delete}\mathtt{1}(L) := \{xy \mid x\mathtt{1}y \in L\}$ is regular.

   Intuitively, $\mathit{delete}\mathtt{1}(L)$ is the set of all strings that can be obtained from strings in L by deleting exactly one $\mathtt{1}$. For example, if $L = \{\mathtt{101101}, \mathtt{00}, \varepsilon\}$, then $\mathit{delete}\mathtt{1}(L) = \{\mathtt{01101}, \mathtt{10101}, \mathtt{10110}\}$.

Work on these later: (In fact, these might be easier than problems 1 and 2.)

3. Consider the following recursively defined function on strings:

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

   Intuitively, $\mathit{stutter}(w)$ doubles every symbol in w. For example:

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

   Let L be an arbitrary regular language.

   (a) Prove that the language $\mathit{stutter}^{-1}(L) := \{w \mid \mathit{stutter}(w) \in L\}$ is regular.

   (b) Prove that the language $\mathit{stutter}(L) := \{\mathit{stutter}(w) \mid w \in L\}$ is regular.

4. Consider the following recursively defined function on strings:

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

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

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

   Once again, let L be an arbitrary regular language.

   (a) Prove that the language $\mathit{evens}^{-1}(L) := \{w \mid \mathit{evens}(w) \in L\}$ is regular.

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

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

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

$$\mathit{flipOdds}(\mathtt{0000111101010101}) = \mathtt{1010010111111111}$$

Solution: Let $M = (Q, s, A, \delta)$ be a 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 *flipOdd1* inverts every other $\mathtt{1}$ bit of its input string, starting with the first $\mathtt{1}$. For example:

$$\mathit{flipOdd1s}(\mathtt{0000\underline{1}1\underline{1}10\underline{1}010\underline{1}01}) = \mathtt{0000\underline{0}1\underline{0}100\underline{0}100\underline{0}1}$$

Solution: Let $M = (Q, s, A, \delta)$ be a 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 $\mathtt{1}$ bit, and 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 $\mathtt{1}$ bit of 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$(L) := \{flipOdd1s(w) \mid w \in L\}$, where the function *flipOdd1* is defined as in the previous problem.

Solution: Let $M = (Q, s, A, \delta)$ be a 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 rejects.

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

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

■

4. Faro$(L) := \left\{faro(w, x) \;\middle|\; w, x \in L \text{ and } |w| = |x|\right\}$, where the function *faro* is defined recursively as follows:

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

For example, $faro(0001101, 1111001) = 01010111100011$. (A "faro shuffle" splits a deck of cards into two equal piles and then perfectly interleaves them.)

Solution: Let $M = (Q, s, A, \delta)$ be a DFA that accepts L. We construct a DFA $M' = (Q', s', A', \delta')$ that accepts Faro(L) as follows.

Intuitively, M' reads the string $faro(w, x)$ as input, splits the string into the subsequences w and x, and passes each of those strings to an independent copy of M.

Each state $(q_1, q_2, next)$ indicates that the copy of M that gets w is in state q_1, the copy of M that gets x is in state q_2, and *next* indicates which copy gets the next input bit.

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

■

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?]*

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, Jeff described an algorithm of 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 some extensions and applications of this algorithm.

1. Describe an algorithm to compute the product of an n-digit number and an m-digit number, where $m < n$, in $O(m^{\lg 3 - 1} n)$ time.

2. Describe an algorithm to compute the decimal representation of 2^n in $O(n^{\lg 3})$ time.

   [Hint: Repeated squaring. The standard algorithm that computes one decimal digit at a time requires $\Theta(n^2)$ time.]

3. Describe a divide-and-conquer algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(n^{\lg 3})$ time.

   [Hint: Let $x = a \cdot 2^{n/2} + b$. Watch out for an extra log factor in the running time.]

Think about later:

4. Suppose we can multiply two n-digit numbers in $O(M(n))$ time. Describe an algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(M(n) \log n)$ 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 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

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

   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

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

   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

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

   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).

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 the string 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 problems. For the first three, use the backtracking algorithms 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 is an automatic zero.

   (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.

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.

Lenny Adve, the founding dean of the new Maximilian Q. Levchin College of Computer Science, has commissioned a series of snow ramps on the south slope of the Orchard Downs sledding hill[1] and challenged Bill Kudeki, head of the Department of Electrical and Computer Engineering, to a sledding contest. Bill and Lenny will both sled down the hill, each trying to maximize their air time. The winner gets to expand their department/college into Siebel Center, the new ECE Building, *and* the English Building; the loser has to move their entire department/college under the Boneyard bridge next to Everitt Lab (along with the English department).

Whenever Lenny or Bill 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 Lenny and Bill can travel through the air. *[Hint: Do whatever you **feel** like you wanna do. Gosh!]*

2. Uh-oh. The university lawyers heard about Lenny and Bill'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 Lenny or Bill 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.

3. **To think about later:** When the lawyers realized that imposing their restriction didn't immediately shut down the contest, they added a new restriction: No ramp can be used more than once! Disgusted by the legal interference, Lenny and Bill 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 Lenny and Bill 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.

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:

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

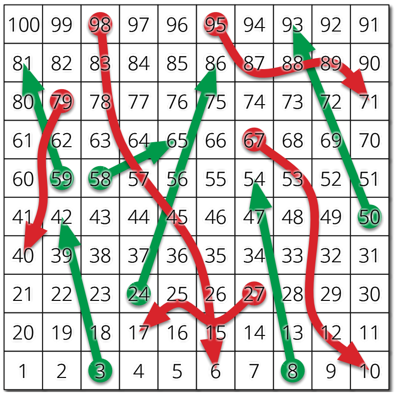
Describe and analyze an algorithm to compute, given a list of integers separated by + and − signs, the maximum possible value the expression can take by adding parentheses. Parentheses must be used only to group additions and subtractions; in particular, do not use them to create implicit multiplication 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*?

---

1. ***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). 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).

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 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 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*?

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 any game, then A must rank better than B in the overall ranking.

   You are given the list of players and their ranking 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 $1 \leq s \leq n$ identifying Judy's home galaxy, and an array $D[1..n]$ containing the distances of each galaxy from s.

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 $u \to v$ has negative weight. Assume the graph has no negative cycles. *[Hint: Modify the input graph and run Dijkstra's algorithm. Alternatively,* ***don't*** *modify the input graph, but run Dijkstra's algorithm anyway.]*

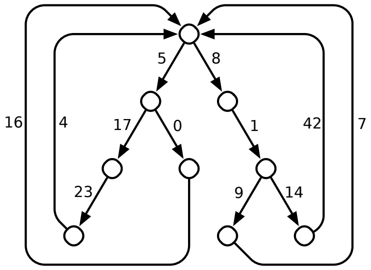
2. You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cites that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving 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 soon as possible, assuming both of you leave home *right now*.

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 $u{\to}v$, 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 $w(u{\to}v)$ denote the weight that you used in your distance computation, and let $w'(u{\to}v)$ denote the correct weight of $u{\to}v$.

   (a) Suppose $w(u{\to}v) > w'(u{\to}v)$; that is, the weight you used for $u{\to}v$ was *larger* than its true weight. Describe an algorithm that repairs the distance array in ***$O(V^2)$ time*** under this assumption. *[Hint: For every pair of vertices x and y, either $u{\to}v$ 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 ***$O(1)$ time***, again assuming that $w(u{\to}v) > w'(u{\to}v)$. *[Hint: Either $u{\to}v$ is the shortest path from u to v or it isn't.]*

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

   (d) **To think about later:** Argue that when $w(u{\to}v) < w'(u{\to}v)$, 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. ***Flappy Bird*** is a popular mobile game written by Nguyễn Hà Đông, originally released in May 2013. The game features a bird named "Faby", who flies to the right at constant speed. Whenever the player taps the screen, Faby is given a fixed upward velocity; between taps, Faby falls due to gravity. Faby flies through a landscape of pipes until it touches either a pipe or the ground, at which point the game is over. Your task, should you choose to accept it, is to develop an algorithm to play Flappy Bird automatically.

   Well, okay, not Flappy Bird exactly, but the following drastically simplified variant, which I will call ***Flappy Pixel***. Instead of a bird, Faby is a single point, specified by three integers: horizontal position x (in pixels), vertical position y (in pixels), and vertical speed y' (in pixels per frame). Faby's environment is described by two arrays $Hi[1..n]$ and $Lo[1..n]$, where for each index i, we have $0 < Lo[i] < Hi[i] < h$ for some fixed height value h. The game is described by the following piece of pseudocode:

```
FLAPPYPIXEL($Hi[1..n], Lo[1..n]$):
  $y \gets \lceil h/2 \rceil$
  $y' \gets 0$
  for $x \gets 1$ to $n$
      if the player taps the screen
          $y' \gets 10$              ⟨⟨flap⟩⟩
      else
          $y' \gets y' - 1$          ⟨⟨fall⟩⟩
      $y \gets y + y'$
      if $y < Lo[x]$ or $y > Hi[x]$
          return FALSE               ⟨⟨player loses⟩⟩
  return TRUE                        ⟨⟨player wins⟩⟩
```

   Notice that in each iteration of the main loop, the player has the option of tapping the screen.

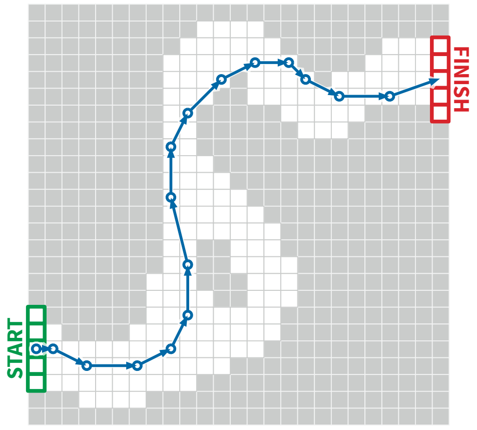
   Describe and analyze an algorithm to determine the minimum number of times that the player must tap the screen to win Flappy Pixel, given the integer h and the arrays $Hi[1..n]$ and $Lo[1..n]$ as input. If the game cannot be won at all, your algorithm should return ∞. Describe the running time of your algorithm as a function of n and h.

[Problem 2 is on the back.]

2. ***Racetrack*** (also known as *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 changes each component of the velocity by at most 1. 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.

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.

   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 line" consists of all `0` bits in column 1, and the "finishing line" consists of all `0` bits in column n.

   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: Your initial analysis can be improved.]

[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 entire line segment between the old position and the new position to lie inside the track. Sometimes Speed Racer has to push the A button.

To think about later:

3. Consider the following variant of Flappy Pixel. The mechanics of the game are unchanged, but now the environment is specified by an array *Points*$[1\,..\,n, 1\,..\,h]$ of integers, which could be positive, negative, or zero. If Faby falls off the top or bottom edge of the environment, the game immediately ends and the player gets nothing. Otherwise, at each frame, the player earns *Points*$[x, y]$ points, where (x, y) is Faby's current position. The game ends when Faby reaches the right end of the environment.

```
FlappyPixel2($Points[1\,..\,n]$):
    $score \leftarrow 0$
    $y \leftarrow \lceil h/2 \rceil$
    $y' \leftarrow 0$
    for $x \leftarrow 1$ to $n$
        if the player taps the screen
            $y' \leftarrow 10$            ⟨⟨flap⟩⟩
        else
            $y' \leftarrow y' - 1$        ⟨⟨fall⟩⟩
        $y \leftarrow y + y'$
        if $y < 1$ or $y > h$
            return $-\infty$          ⟨⟨fail⟩⟩
        $score \leftarrow score + Points[x, y]$
    return $score$
```

Describe and analyze an algorithm to determine the maximum possible score that a player can earn in this game.

4. We can also consider a similar variant of Racetrack. Instead of bits, the "track" is described by an array *Points*$[1\,..\,n, 1\,..\,n]$ of *numbers*, which could be positive, negative, or zero. Whenever the car lands on a grid cell (i, j), the player receives *Points*$[i, j]$ points. Forbidden grid cells are indicated by *Points*$[i, j] = -\infty$.

Describe and analyze an algorithm to find the largest possible score that a player can earn by moving a car from column 1 (the starting line) to column n (the finish line).

[Hint: Wait, what if all the point values are positive?]

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 $x_1, x_2, \ldots, x_n \in \{\text{TRUE}, \text{FALSE}\}$ 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 $x_1, x_2, \ldots, x_n \in \{\text{TRUE}, \text{FALSE}\}$ that make K output TRUE, or NONE if there are no such inputs.

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

2. 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.

   [Hint: You've seen this problem before.]

   (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. Recall 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.

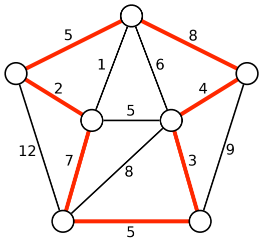
   (b) Prove that kColor problem is NP-hard for any $k \geq 3$.

2. 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.

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

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

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

3. Given an undirected graph G, does G have a spanning tree with at most 374 leaves?

Proving that a language L is undecidable by reduction requires several steps. (These are the essentially the same steps you already use to prove that a problem is NP-hard.)

- Choose a language L' that you already know is undecidable (because we told you so in class). The simplest choice is usually the standard halting language

$$\text{HALT} := \big\{\langle M, w\rangle \;\big|\; M \text{ halts on } w\big\}$$

- Describe an algorithm that decides L', using an algorithm that decides L as a black box. Typically your reduction will have the following form:

  Given an arbitrary string x, construct a special string y,
  such that $y \in L$ if and only if $x \in L'$.

  In particular, if $L = \text{HALT}$, your reduction will have the following form:

  Given the encoding $\langle M, w\rangle$ of a Turing machine M and a string w,
  construct a special string y, such that
  $y \in L$ if and only if M halts on input w.

- Prove that your algorithm is correct. This proof almost always requires two separate steps:
  - Prove that if $x \in L'$ then $y \in L$.
  - Prove that if $x \notin L'$ then $y \notin L$.

Very important: Name every object in your proof, and *always* refer to objects by their names. Never refer to "the Turing machine" or "the algorithm" or "the code" or "the input string" or (gods forbid) "it" or "this", even in casual conversation, even if you're "just" explaining your intuition, even when you're "just" *thinking* about the reduction to yourself.

Prove that the following languages are undecidable.

1. $\text{ACCEPTILLINI} := \big\{\langle M\rangle \;\big|\; M \text{ accepts the string } \texttt{ILLINI}\big\}$
2. $\text{ACCEPTTHREE} := \big\{\langle M\rangle \;\big|\; M \text{ accepts exactly three strings}\big\}$
3. $\text{ACCEPTPALINDROME} := \big\{\langle M\rangle \;\big|\; M \text{ accepts at least one palindrome}\big\}$
4. $\text{ACCEPTONLYPALINDROMES} := \big\{\langle M\rangle \;\big|\; \text{Every string accepted by } M \text{ is a palindrome}\big\}$

A solution for problem 1 appears on the next page; don't look at it until you've thought a bit about the problem first.

Solution (for problem 1): For the sake of argument, suppose there is an algorithm DecideAcceptIllini that correctly decides the language AcceptIllini. Then we can solve the halting problem as follows:

```
DecideHalt(⟨M, w⟩):
  Encode the following Turing machine M′:
        M′(x):
          run M on input w
          return True
  if DecideAcceptIllini(⟨M′⟩)
        return True
  else
        return False
```

We prove this reduction correct as follows:

$\Longrightarrow$ Suppose M halts on input w.

Then M' accepts *every* input string x.

In particular, M' accepts the string `ILLINI`.

So DecideAcceptIllini accepts the encoding $\langle M' \rangle$.

So DecideHalt correctly accepts the encoding $\langle M, w \rangle$.

$\Longleftarrow$ Suppose M does not halt on input w.

Then M' diverges on *every* input string x.

In particular, M' does not accept the string `ILLINI`.

So DecideAcceptIllini rejects the encoding $\langle M' \rangle$.

So DecideHalt correctly rejects the encoding $\langle M, w \rangle$.

In both cases, DecideHalt is correct. But that's impossible, because Halt is undecidable. We conclude that the algorithm DecideAcceptIllini does not exist. ■

As usual for undecidablility proofs, this proof invokes *four* distinct Turing machines:

- The hypothetical algorithm DecideAcceptIllini.
- The new algorithm DecideHalt that we construct in the solution.
- The arbitrary machine M whose encoding is part of the input to DecideHalt.
- The special machine M' whose encoding DecideHalt constructs (from the encoding of M and w) and then passes to DecideAcceptIllini.

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 $\textsc{AcceptIn}(\mathcal{L}) := \left\{\langle M\rangle \;\middle|\; \textsc{Accept}(M) \in \mathcal{L}\right\}$ is undecidable.

You may find the following Turing machines useful:

- $M_{\textsc{Accept}}$ accepts every input.
- $M_{\textsc{Reject}}$ rejects every input.
- $M_{\textsc{Hang}}$ infinite-loops on every input.

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

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

To think about later. Which of the following languages are undecidable? How would you prove that? Remember that we know several ways to prove undecidability:

- Diagonalization: Assume the language is decidable, and derive an algorithm with self-contradictory behavior.
- Reduction: Assume the language is decidable, and derive an algorithm for a known undecidable language, like $\textsc{Halt}$ or $\textsc{SelfReject}$ or $\textsc{NeverAccept}$.
- Rice's Theorem: Find an appropriate family of languages $\mathcal{L}$, a machine Y that accepts a language in $\mathcal{L}$, and a machine N that does not accept a language in $\mathcal{L}$.
- Closure: If two languages L and L' are decidable, then the languages $L \cap L'$ and $L \cup L'$ and $L \setminus L'$ and $L \oplus L'$ and L^* are all decidable, too.

---

6. $\textsc{Accept}\{\{\varepsilon\}\} := \left\{\langle M\rangle \;\middle|\; M \text{ accepts only the string } \varepsilon\text{; that is, } \textsc{Accept}(M) = \{\varepsilon\}\right\}$
7. $\textsc{Accept}\{\varnothing\} := \left\{\langle M\rangle \;\middle|\; M \text{ does not accept any strings; that is, } \textsc{Accept}(M) = \varnothing\right\}$
8. $\textsc{Accept}\varnothing := \left\{\langle M\rangle \;\middle|\; \textsc{Accept}(M) \text{ is not an acceptable language}\right\}$
9. $\textsc{Accept=Reject} := \left\{\langle M\rangle \;\middle|\; \textsc{Accept}(M) = \textsc{Reject}(M)\right\}$
10. $\textsc{Accept}{\neq}\textsc{Reject} := \left\{\langle M\rangle \;\middle|\; \textsc{Accept}(M) \neq \textsc{Reject}(M)\right\}$
11. $\textsc{Accept}{\cup}\textsc{Reject} := \left\{\langle M\rangle \;\middle|\; \textsc{Accept}(M) \cup \textsc{Reject}(M) = \Sigma^*\right\}$

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, check "Yes" if the statement is ***always*** true and "No" otherwise. Each correct answer is worth +1 point; each incorrect answer is worth −½ point; checking "I don't know" is worth +¼ point; and flipping a coin is (on average) worth +¼ point. You do ***not*** need to prove your answer is correct.

   Read each statement *very* carefully. Some of these are deliberately subtle.

   (a) Every infinite language is regular.

   (b) If L is not regular, then for every string $w \in L$, there is a DFA that accepts w.

   (c) If L is context-free and L has a finite fooling set, then L is regular.

   (d) If L is regular and $L' \cap L = \varnothing$, then L' is regular.

   (e) The language $\{\mathtt{0}^i\mathtt{1}^j\mathtt{0}^k \mid i+j+k \geq 374\}$ is not regular.

   (f) The language $\{\mathtt{0}^i\mathtt{1}^j\mathtt{0}^k \mid i+j-k \geq 374\}$ is not regular.

   (g) Let $M = (Q, \{\mathtt{0},\mathtt{1}\}, s, A, \delta)$ be an arbitrary DFA, and let $M' = (Q, \{\mathtt{0},\mathtt{1}\}, s, A, \delta')$ be the DFA obtained from M by changing every 0-transition into a 1-transition and vice versa. More formally, M and M' have the same states, input alphabet, starting state, and accepting states, but $\delta'(q,\mathtt{0}) = \delta(q,\mathtt{1})$ and $\delta'(q,\mathtt{1}) = \delta(q,\mathtt{0})$. Then $L(M) \cap L(M') = \varnothing$.

   (h) Let $M = (Q, \Sigma, s, A, \delta)$ be an arbitrary NFA, and $M' = (Q', \Sigma, s, A', \delta')$ be any NFA obtained from M by deleting some subset of the states. More formally, we have $Q' \subseteq Q$, $A' = A \cap Q'$, and $\delta'(q,a) = \delta(q,a) \cap Q'$ for all $q \in Q'$. Then $L(M') \subseteq L(M)$.

   (i) For every regular language L, the language $\left\{\mathtt{0}^{|w|} \;\middle|\; w \in L\right\}$ is also regular.

   (j) For every context-free language L, the language $\left\{\mathtt{0}^{|w|} \;\middle|\; w \in L\right\}$ is also context-free.

2. For any language L, define

$$\text{StripInit0s}(L) = \left\{w \;\middle|\; \mathtt{0}^j w \in L \text{ for some } j \geq 0\right\}$$

   Less formally, StripInit0s(L) is the set of all strings obtained by stripping any number of initial 0s from strings in L. For example, if L is the one-string language $\{\mathtt{00011010}\}$, then

$$\text{StripInit0s}(L) = \{\mathtt{00011010}, \mathtt{0011010}, \mathtt{011010}, \mathtt{11010}\}.$$

   Prove that if L is a regular language, then StripInit0s(L) is also a regular language.

3. For each of the following languages L over the alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$, give a regular expression that represents L ***and*** describe a DFA that recognizes L.

   (a) $\{\texttt{0}^n w \texttt{1}^n \mid n > 1 \text{ and } w \in \Sigma^*\}$

   (b) All strings in $\texttt{0}^*\texttt{1}\texttt{0}^*$ whose length is a multiple of 3.

4. The *parity* of a bit-string is `0` if the number of `1` bits is even, and `1` if the number of `1` bits is odd. 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 $EvenParity(L) := \{w \in L \mid parity(w) = \texttt{0}\}$ is also regular.

   (c) Let L be an arbitrary regular language. Prove that the language $AddParity(L) := \{w \bullet parity(w) \mid w \in L\}$ is also regular. For example, if L contains the string `11100` and `11000`, then $AddParity(L)$ contains the strings `111001` and `110000`.

5. Let L be the language $\left\{\texttt{0}^i\texttt{1}^j\texttt{0}^k \;\middle|\; i = j \text{ or } j = k\right\}$.

   (a) ***Prove*** that L is not a regular language.

   (b) Describe a context-free grammar for L.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, check "Yes" if the statement is ***always*** true and "No" otherwise. Each correct answer is worth +1 point; each incorrect answer is worth −½ point; checking "I don't know" is worth +¼ point; and flipping a coin is (on average) worth +¼ point. You do ***not*** need to prove your answer is correct.

   Read each statement *very* carefully. Some of these are deliberately subtle.

   (a) No infinite language is regular.

   (b) If L is regular, then for every string $w \in L$, there is a DFA that rejects w.

   (c) If L is context-free and L has a finite fooling set, then L is not regular.

   (d) If L is regular and $L' \cap L = \varnothing$, then L' is not regular.

   (e) The language $\{\mathtt{0}^i \mathtt{1}^j \mathtt{0}^k \mid i + j + k \geq 374\}$ is regular.

   (f) The language $\{\mathtt{0}^i \mathtt{1}^j \mathtt{0}^k \mid i + j - k \geq 374\}$ is regular.

   (g) Let $M = (Q, \{\mathtt{0}, \mathtt{1}\}, s, A, \delta)$ be an arbitrary DFA, and let $M' = (Q, \{\mathtt{0}, \mathtt{1}\}, s, A, \delta')$ be the DFA obtained from M by changing every $\mathtt{0}$-transition into a $\mathtt{1}$-transition and vice versa. More formally, M and M' have the same states, input alphabet, starting state, and accepting states, but $\delta'(q, \mathtt{0}) = \delta(q, \mathtt{1})$ and $\delta'(q, \mathtt{1}) = \delta(q, \mathtt{0})$. Then $L(M) \cup L(M') = \{\mathtt{0}, \mathtt{1}\}^*$.

   (h) Let $M = (Q, \Sigma, s, A, \delta)$ be an arbitrary NFA, and $M' = (Q', \Sigma, s, A', \delta')$ be any NFA obtained from M by deleting some subset of the states. More formally, we have $Q' \subseteq Q$, $A' = A \cap Q'$, and $\delta'(q, a) = \delta(q, a) \cap Q'$ for all $q \in Q'$. Then $L(M') \subseteq L(M)$.

   (i) For every non-regular language L, the language $\left\{\mathtt{0}^{|w|} \;\middle|\; w \in L\right\}$ is also non-regular.

   (j) For every context-free language L, the language $\left\{\mathtt{0}^{|w|} \;\middle|\; w \in L\right\}$ is also context-free.

2. For any language L, define

$$\textsc{StripFinal}\mathtt{0}\textsc{s}(L) = \{w \mid w\mathtt{0}^n \in L \text{ for some } n \geq 0\}$$

   Less formally, $\textsc{StripFinal}\mathtt{0}\textsc{s}(L)$ is the set of all strings obtained by stripping any number of final $\mathtt{0}$s from strings in L. For example, if L is the one-string language $\{\mathtt{01101000}\}$, then

$$\textsc{StripFinal}\mathtt{0}\textsc{s}(L) = \{\mathtt{01101}, \mathtt{011010}, \mathtt{0110100}, \mathtt{01101000}\}.$$

   Prove that if L is a regular language, then $\textsc{StripFinal}\mathtt{0}\textsc{s}(L)$ is also a regular language.

3. For each of the following languages L over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, give a regular expression that represents L ***and*** describe a DFA that recognizes L.

   (a) $\left\{\mathtt{0}^n w \mathtt{1}^n \;\middle|\; n \geq 1 \text{ and } w \in \Sigma^+\right\}$

   (b) All strings in $\mathtt{0}^*\mathtt{1}^*\mathtt{0}^*$ whose length is even.

4. The *parity* of a bit-string is $\mathtt{0}$ if the number of $\mathtt{1}$ bits is even, and $\mathtt{1}$ if the number of $\mathtt{1}$ bits is odd. For example:

$$\mathit{parity}(\varepsilon) = \mathtt{0} \qquad \mathit{parity}(\mathtt{0010100}) = \mathtt{0} \qquad \mathit{parity}(\mathtt{00101110100}) = \mathtt{1}$$

   (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 $\mathit{OddParity}(L) := \{w \in L \mid \mathit{parity}(w) = \mathtt{1}\}$ is also regular.

   (c) Let L be an arbitrary regular language. Prove that the language $\mathit{AddParity}(L) := \{\mathit{parity}(w) \cdot w \mid w \in L\}$ is also regular. For example, if L contains the strings $\mathtt{01110}$ and $\mathtt{01100}$, then $\mathit{AddParity}(L)$ contains the strings $\mathtt{101110}$ and $\mathtt{001100}$.

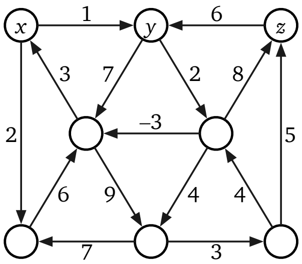
5. Let L be the language $\left\{\mathtt{0}^i \mathtt{1}^j \mathtt{0}^k \;\middle|\; 2i = k \text{ or } i = 2k\right\}$.

   (a) ***Prove*** that L is not a regular language.

   (b) Describe a context-free grammar for L.

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. ***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.

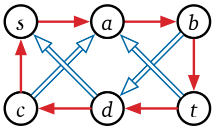
   (b) A breadth-first tree rooted at y.

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

   (d) The shortest directed cycle.

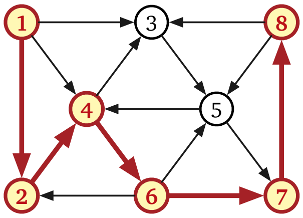
2. Suppose you are given a directed graph G where some edges are red and the remaining edges are blue. Describe an algorithm to find the shortest walk in G from one vertex s to another vertex t in which no three consecutive edges have the same color. That is, if the walk contains two red edges in a row, 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 $s{\to}a{\to}b{\Rightarrow}d{\to}c{\Rightarrow}a{\to}b{\to}c$.



3. 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 $1{\to}2{\to}4{\to}6{\to}7{\to}8$.



4. 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.)

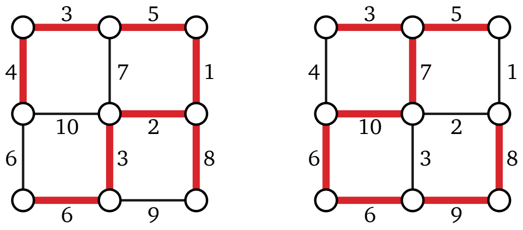
2	3	99	7	11	13	17	19	25	29	31	−5	41	43	47	53	8	61	67	71

   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 $k = 1$.]*

5. Suppose you give one of your interns at Twitbook an undirected graph G with weighted edges, and you ask them to compute a shortest-path tree rooted at a particular vertex. Two weeks later, your intern finally comes back with a spanning tree T of G. Unfortunately, the intern didn't record the shortest-path distances, the direction of the shortest-path edges, or even the source vertex (which you and the intern have both forgotten).

   Describe and analyze an algorithm to determine, given a weighted undirected graph G and a spanning tree T of G, whether T is in fact a *shortest-path* tree in G. Assume all edge weights are non-negative.

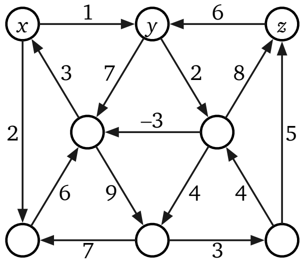
   For example, given the inputs shown below, your algorithm should return True for the example on the left, because T is a shortest-path tree rooted at the upper right vertex of G, but your algorithm should return False for the example on the right.



Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. ***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.

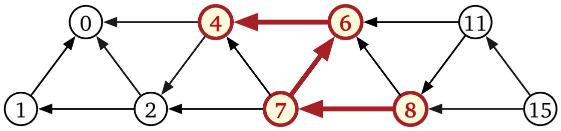
(c) A shortest-path tree rooted at z.

(d) The shortest directed cycle.

2. Let G be a ***directed*** graph, where every vertex v has an associated height $h(v)$, and for every edge $u \rightarrow v$ 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 $8 \rightarrow 7 \rightarrow 6 \rightarrow 4$.



3. 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.)

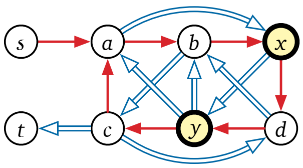
2	3	99	7	11	13	17	19	25	29	31	−5	41	43	47	53	8	61	67	71

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 $k = 1$.]*

4. 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.[1]

   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.



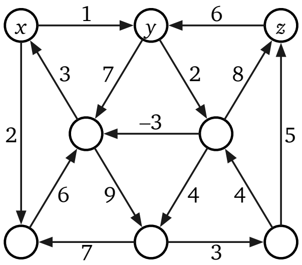
5. Let G be a directed graph with weighted edges, in which every vertex is colored either red, green, or blue. Describe and analyze an algorithm to compute the length of the shortest walk in G that starts at a red vertex, then visits any number of vertices of any color, then visits a green vertex, then 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.

[1] 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.

Write your answers in the separate answer booklet.

Please return this question sheet and your cheat sheet with your answers.

1. ***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.

2. After a few weeks of following your uphill-downhill walking path to work, your boss demands that you start showing up to work on time, so you decide to change your walking strategy. Your new goal is to walk to the highest altitude you can (to maximize exercise), while keeping the total length of your walk from home to work below some threshold (to make sure you get to work on time). Describe and analyze an algorithm to compute your new favorite route.

   Your input consists of an ***undirected*** graph G, where each vertex v has a height $h(v)$ and each edge e has a positive length $\ell(e)$, along with a start vertex s, a target vertex t, and a maximum length L. Your algorithm should return the ***maximum height*** reachable by a walk from s to t in G, whose total length is at most L.

   [Hint: This is the same input as HW8 problem 1, but the problem is completely different. In particular, the number of uphill/downhill switches in your walk is irrelevant.]

3. 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.)

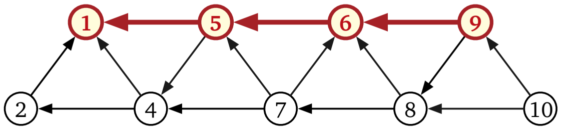
2	3	99	7	11	13	17	19	25	29	31	−5	41	43	47	53	8	61	67	71

   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 $k = 1$.]*

4. 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 ***maximum span*** of a path in G with at most 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.

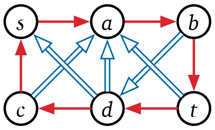
   For example, given the following labeled graph and the integer $k = 3$ as input, your algorithm should return the integer 8, which is the span of the downward path $9{\to}6{\to}5{\to}1$.



   [Hint: This is a very different question from problem 2.]

5. Suppose you are given a directed graph G where some edges are red and the remaining edges are blue, along with two vertices s and t. Describe an algorithm to compute the length of the shortest walk in G from s to t that traverses an even number of red edges and an even number of blue edges. If the walk traverses the same edge multiple times, each traversal counts toward the total for that color.

   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 6, because the shortest legal walk from s to t is $s{\to}a{\to}b{\Rightarrow}d{\Rightarrow}a{\to}b{\to}t$.



CS/ECE 374 A ✧ Spring 2018

Final Exam ♫

May 8, 2018

Real name:

NetID:

Gradescope name:

Gradescope email:

- ***Don't panic!***
- If you brought anything except your writing implements and your two 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.
- Please clearly print your real name, your university NetID, your Gradescope name, and your Gradescope email address in the boxes above. **We will not scan this page into Gradescope.**
- Please also print **only the name you are using on Gradescope** at the top of every page of the answer booklet, except this cover page. These are the pages we will scan into Gradescope.
- Please do not write outside the black boxes on each page; these indicate the area of the page that the scanner can actually see.
- **Please read the entire exam before writing anything.** Please ask for clarification if any question is unclear.
- **The exam lasts 180 minutes.**
- If you run out of space for an answer, continue on the back of the page, or on the blank pages at the end of this booklet, **but please tell us where to look.** Alternatively, feel free to tear out the blank pages and use them as scratch paper.
- As usual, answering any (sub)problem with "I don't know" (and nothing else) is worth 25% partial credit. **Yes, even for problem 1.** Correct, complete, but suboptimal solutions are *always* worth more than 25%. A blank answer is not the same as "I don't know".
- **Please return your cheat sheets and all scratch paper with your answer booklet.**
- ***Good luck!*** And thanks for a great semester!

Beware of the man who works hard to learn something,
learns it, and finds himself no wiser than before.

He is full of murderous resentment of people who are ignorant
without having come by their ignorance the hard way.

— Bokonon

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 1**	Gradescope name:

For each of the following questions, indicate ***every*** correct answer by marking the "Yes" box, and indicate ***every*** incorrect answer by marking the "No" box. **Assume P ≠ NP.** If there is any other ambiguity or uncertainty, mark the "No" box. For example:

☒ Yes ☐ No $2+2=4$

☐ Yes ☒ No $x+y=5$

☐ Yes ☒ No 3SAT can be solved in polynomial time.

☒ Yes ☐ No Jeff is not the Queen of England.

☒ Yes ☐ No If $P=NP$ then Jeff is the Queen of England.

There are 40 yes/no choices altogether. Each correct choice is worth +½ point; each incorrect choice is worth —¼ point. TO indicate "I don't know", write IDK to the left of the Yes/No boxes; each IDK is worth +1/8 point.

(a) Which of the following statements is true for ***every*** language $L \subseteq \{0,1\}^*$?

☐ Yes ☐ No L is infinite.

☐ Yes ☐ No L^* contains the empty string ε.

☐ Yes ☐ No L^* is decidable.

☐ Yes ☐ No If L is regular then $(L^*)^*$ is regular.

☐ Yes ☐ No If L is the intersection of two decidable languages, then L is decidable.

☐ Yes ☐ No If L is the intersection of two undecidable languages, then L is undecidable.

☐ Yes ☐ No If L is the complement of a regular language, then L^* is regular.

☐ Yes ☐ No If L has an infinite fooling set, then L is undecidable.

☐ Yes ☐ No L is decidable if and only if its complement $\overline{L}$ is undecidable.

(b) Which of the following statements is true for ***every*** directed graph $G = (V, E)$?

Yes	No	$E \neq \varnothing$.
Yes	No	Given the graph G as input, Floyd-Warshall runs in $O(E^3)$ time.
Yes	No	If G has at least one source and at least one sink, then G is a dag.
Yes	No	We can compute a spanning tree of G using whatever-first search.
Yes	No	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.

(c) Which of the following languages over the alphabet $\{\mathtt{0}, \mathtt{1}\}$ are ***regular***?

Yes	No	$\{\mathtt{0}^m\mathtt{1}\mathtt{0}^n \mid m \leq n\}$
Yes	No	$\{\mathtt{0}^m\mathtt{1}\mathtt{0}^n \mid m + n \geq 374\}$
Yes	No	Binary representations of all perfect squares
Yes	No	$\big\{xy \bigm\vert yx \text{ is a palindrome}\big\}$
Yes	No	$\big\{\langle M\rangle \bigm\vert M \text{ accepts a finite number of non-palindromes}\big\}$

(d) Which of the following languages are ***decidable***?

Yes	No	Binary representations of all perfect squares
Yes	No	$\big\{xy \in \{\mathtt{0}, \mathtt{1}\}^* \bigm\vert yx \text{ is a palindrome}\big\}$
Yes	No	$\big\{\langle M\rangle \bigm\vert M \text{ accepts the binary representation of every perfect square}\big\}$
Yes	No	$\big\{\langle M\rangle \bigm\vert M \text{ accepts a finite number of non-palindromes}\big\}$
Yes	No	The set of all regular expressions that represent the language $\{\mathtt{0}, \mathtt{1}\}^*$. (This is a language over the alphabet $\{\mathtt{Ø}, \mathtt{Ɛ}, \mathtt{0}, \mathtt{1}, \star, \mathtt{+}, \mathtt{(}, \mathtt{)}\}$.)

(e) Which of the following languages can be proved undecidable ***using Rice's Theorem***?

Yes No $\{\langle M\rangle \mid M \text{ accepts a finite number of strings}\}$

Yes No $\{\langle M\rangle \mid M \text{ accepts both } \langle M\rangle \text{ and } \langle M\rangle^R\}$

Yes No $\{\langle M\rangle \mid M \text{ accepts exactly 374 palindromes}\}$

Yes No $\{\langle M\rangle \mid M \text{ accepts some string } w \text{ after at most } |w|^2 \text{ steps}\}$

(f) Suppose we want to prove that the following language is undecidable.

$$\textsc{Chalmers} := \{\langle M\rangle \mid M \text{ accepts both } \texttt{STEAMED} \text{ and } \texttt{HAMS}\}$$

Professor Skinner suggests a reduction from the standard halting language

$$\textsc{Halt} := \{\langle M\rangle\#w \mid M \text{ halts on inputs } w\}.$$

Specifically, suppose there is a Turing machine Ch that decides Chalmers. Professor Skinner claims that the following algorithm decides Halt.

```
DecideHalt($\langle M\rangle\#w$):
    Encode the following Turing machine:
        AuroraBorealis($x$):
            if $x = \texttt{STEAMED}$ or $x = \texttt{HAMS}$ or $x = \texttt{ALBANY}$
                run $M$ on input $w$
                return False
            else
                return True
    return $Ch(\langle\textsc{AuroraBorealis}\rangle)$
```

Which of the following statements is true for all inputs $\langle M\rangle\#w$?

Yes No If M accepts w, then AuroraBorealis accepts CLAMS.

Yes No If M rejects w, then AuroraBorealis rejects UTICA.

Yes No If M rejects w, then AuroraBorealis halts on every input string.

Yes No If M accepts w, then Ch accepts ⟨AuroraBorealis⟩.

Yes No DecideHalt decides the language Halt. (That is, Professor Skinner's reduction is actually correct.)

Yes No DecideHalt actually runs (or simulates) M.

Yes No We could have proved Chalmers is undecidable using Rice's theorem instead of this reduction.

(g) Consider the following pair of languages:

- 3Color := $\big\{G \;\big|\; G \text{ is a 3-colorable undirected graph}\big\}$
- Tree := $\big\{G \;\big|\; G \text{ is a connected acyclic undirected graph}\big\}$

(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?

□ Yes □ No Tree ∪ 3Color is NP-hard.

□ Yes □ No Tree ∩ 3Color is NP-hard.

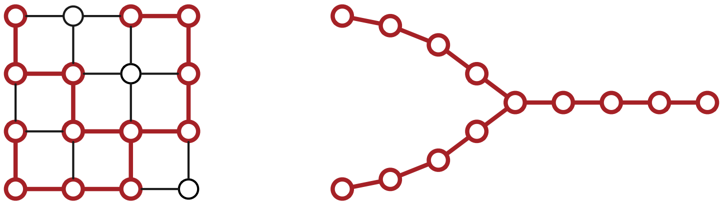
□ Yes □ No 3Color is undecidable.

□ Yes □ No There is a polynomial-time reduction from 3Color to Tree.

□ Yes □ No There is a polynomial-time reduction from Tree to 3Color.

CS/ECE 374 A ✧ Spring 2018 Final Exam ♫ Problem 2	Gradescope name:

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.

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 3**	Gradescope name:

Fix the alphabet $\Sigma = \{0, 1\}$. Recall that a *run* in a string $w \in \Sigma^*$ is a maximal non-empty substring in which all symbols are equal. For example, the string 000001000111111101 consists of exactly six runs: 00000100011111110 = 00000 • 1 • 000 • 1111111 • 0 • 1.

(a) Let L be the set of all strings in Σ^* where every run has odd length. For example, L contains the string 000100000, but L does not contain the string 00011.

Describe both a regular expression for L and a DFA that accepts L.

(b) Let L' be the set of all strings in Σ^* that have the same number of even-length runs and odd-length runs. For example, L' does not contain the string 000011101, because it has three odd-length runs but only one even-length run, but L' does contain the string 0000111011, because it has two runs of each parity.

Prove that L' is not regular.

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 4**	Gradescope name:

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.

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 5**	Gradescope name:

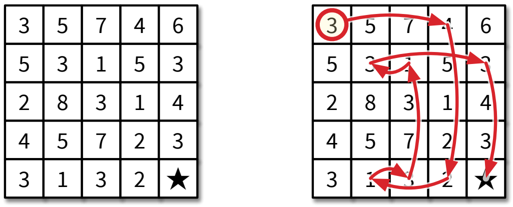
(a) Fix the alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$. Describe and analyze an efficient algorithm for the following problem: Given an NFA M over Σ, does M accept at least one string? Equivalently, is $L(M) \neq \varnothing$?

(b) Recall from Homework 10 that deciding whether a given NFA accepts *every* string is NP-hard. Also recall that the complement of every regular language is regular; thus, for any NFA M, there is another NFA M' such that $L(M') = \Sigma^* \setminus L(M)$. So why doesn't your algorithm from part (a) imply that P=NP?

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 6**	Gradescope name:

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.



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.

(scratch paper)

(scratch paper)

(scratch paper)

(scratch paper)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

FeasibleILP: Given a matrix $A \in \mathbb{Z}^{n \times d}$ and a vector $b \in \mathbb{Z}^n$, determine whether the set of feasible integer points $\max\{x \in \mathbb{Z}^d \mid Ax \leq b, x \geq 0\}$ is empty.

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

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

SteamedHams: Aurora borealis? At this time of year, at this time of day, in this part of the country, localized entirely within your kitchen? May I see it?

CS/ECE 374 A ✧ Spring 2018

Final Exam ♫

May 8, 2018

Real name:	
NetID:	

Gradescope name:	
Gradescope email:	

- ***Don't panic!***
- If you brought anything except your writing implements and your two 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.
- Please clearly print your real name, your university NetID, your Gradescope name, and your Gradescope email address in the boxes above. **We will not scan this page into Gradescope.**
- Please also print **only the name you are using on Gradescope** at the top of every page of the answer booklet, except this cover page. These are the pages we will scan into Gradescope.
- Please do not write outside the black boxes on each page; these indicate the area of the page that the scanner can actually see.
- **Please read the entire exam before writing anything.** Please ask for clarification if any question is unclear.
- **The exam lasts 180 minutes.**
- If you run out of space for an answer, continue on the back of the page, or on the blank pages at the end of this booklet, **but please tell us where to look.** Alternatively, feel free to tear out the blank pages and use them as scratch paper.
- As usual, answering any (sub)problem with "I don't know" (and nothing else) is worth 25% partial credit. **Yes, even for problem 1.** Correct, complete, but suboptimal solutions are *always* worth more than 25%. A blank answer is not the same as "I don't know".
- **Please return your cheat sheets and all scratch paper with your answer booklet.**
- ***Good luck!*** And thanks for a great semester!

Beware of the man who works hard to learn something,
learns it, and finds himself no wiser than before.

He is full of murderous resentment of people who are ignorant
without having come by their ignorance the hard way.

— Bokonon

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 1**	Gradescope name:

For each of the following questions, indicate ***every*** correct answer by marking the "Yes" box, and indicate ***every*** incorrect answer by marking the "No" box. **Assume P ≠ NP.** If there is any other ambiguity or uncertainty, mark the "No" box. For example:

☒ Yes ☐ No $2+2=4$

☐ Yes ☒ No $x+y=5$

☐ Yes ☒ No 3SAT can be solved in polynomial time.

☒ Yes ☐ No Jeff is not the Queen of England.

☒ Yes ☐ No If $P = NP$ then Jeff is the Queen of England.

There are 40 yes/no choices altogether. Each correct choice is worth +½ point; each incorrect choice is worth −¼ point. TO indicate "I don't know", write IDK to the left of the Yes/No boxes; each IDK is worth +1/8 point.

(a) Which of the following statements is true for ***every*** language $L \subseteq \{0, 1\}^*$?

☐ Yes ☐ No L is finite.

☐ Yes ☐ No L^* contains the empty string ε.

☐ Yes ☐ No L^* is decidable.

☐ Yes ☐ No If L is regular then $\Sigma^* \setminus L^*$ is regular.

☐ Yes ☐ No If L is the intersection of two decidable languages, then L is decidable.

☐ Yes ☐ No If L is the intersection of two undecidable languages, then L is undecidable.

☐ Yes ☐ No If L^* is the complement of a regular language, then L is regular.

☐ Yes ☐ No If L is undecidable, then every fooling set for L is infinite.

☐ Yes ☐ No L is decidable if and only if its complement $\overline{L}$ is undecidable.

(b) Which of the following statements is true for ***every*** directed graph $G = (V, E)$?

Yes	No	$E \neq \varnothing$.
Yes	No	Given the graph G as input, Floyd-Warshall runs in $O(E^3)$ time.
Yes	No	If G has at least one source and at least one sink, then G is a dag.
Yes	No	We can compute a spanning tree of G using whatever-first search.
Yes	No	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.

(c) Which of the following languages over the alphabet $\{\mathtt{0}, \mathtt{1}\}$ are ***regular***?

Yes	No	$\{\mathtt{0}^m\mathtt{1}\mathtt{0}^n \mid m \geq n\}$
Yes	No	$\{\mathtt{0}^m\mathtt{1}\mathtt{0}^n \mid m - n \geq 374\}$
Yes	No	Binary representations of all integers divisible by 374
Yes	No	$\big\{xy \bigm\vert yx \text{ is a palindrome}\big\}$
Yes	No	$\big\{\langle M\rangle \bigm\vert M \text{ accepts a finite number of non-palindromes}\big\}$

(d) Which of the following languages are ***decidable***?

Yes	No	Binary representations of all integers divisible by 374
Yes	No	$\big\{xy \in \{\mathtt{0}, \mathtt{1}\}^* \bigm\vert yx \text{ is a palindrome}\big\}$
Yes	No	$\big\{\langle M\rangle \bigm\vert M \text{ accepts the binary representation of every integer divisible by } 374\big\}$
Yes	No	$\big\{\langle M\rangle \bigm\vert M \text{ accepts a finite number of non-palindromes}\big\}$
Yes	No	The set of all regular expressions that represent the language $\{\mathtt{0}, \mathtt{1}\}^*$. (This is a language over the alphabet $\{\mathtt{\emptyset}, \mathtt{\varepsilon}, \mathtt{0}, \mathtt{1}, \mathtt{*}, \mathtt{+}, \mathtt{(}, \mathtt{)}\}$.)

(e) Which of the following languages can be proved undecidable ***using Rice's Theorem***?

Yes No $\{\langle M\rangle \mid M \text{ accepts an infinite number of strings}\}$

Yes No $\{\langle M\rangle \mid M \text{ accepts either } \langle M\rangle \text{ or } \langle M\rangle^R\}$

Yes No $\{\langle M\rangle \mid M \text{ accepts } \texttt{001100} \text{ but rejects } \texttt{110011}\}$

Yes No $\{\langle M\rangle \mid M \text{ accepts some string } w \text{ after at most } |w|^2 \text{ steps}\}$

(f) Suppose we want to prove that the following language is undecidable.

$$\textsc{Chalmers} := \{\langle M\rangle \mid M \text{ accepts both } \texttt{STEAMED} \text{ and } \texttt{HAMS}\}$$

Professor Skinner suggests a reduction from the standard halting language

$$\textsc{Halt} := \{\langle M\rangle\#w \mid M \text{ halts on inputs } w\}.$$

Specifically, suppose there is a Turing machine Ch that decides Chalmers. Professor Skinner claims that the following algorithm decides Halt.

```
DecideHalt(⟨M⟩#w):
  Encode the following Turing machine:
        AuroraBorealis(x):
          if $x$ = STEAMED or $x$ = HAMS or $x$ = ALBANY
              run $M$ on input $w$
              return True
          else
              return False
  return $Ch$(⟨AuroraBorealis⟩)
```

Which of the following statements is true for all inputs $\langle M\rangle\#w$?

Yes No If M accepts w, then AuroraBorealis accepts `CLAMS`.

Yes No If M rejects w, then AuroraBorealis rejects `UTICA`.

Yes No If M hangs on w, then AuroraBorealis accepts every input string.

Yes No If M accepts w, then Ch accepts ⟨AuroraBorealis⟩.

Yes No DecideHalt decides the language Halt. (That is, Professor Skinner's reduction is actually correct.)

Yes No DecideHalt actually runs (or simulates) M.

Yes No We could have proved Chalmers is undecidable using Rice's theorem instead of this reduction.

(g) Consider the following pair of languages:

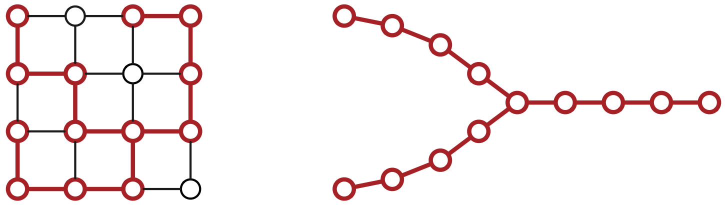
- 3Color := $\left\{G \;\middle|\; G \text{ is a 3-colorable undirected graph}\right\}$
- Tree := $\left\{G \;\middle|\; G \text{ is a connected acyclic undirected graph}\right\}$

(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?

Yes	No	Tree ∪ 3Color is NP-hard.
Yes	No	Tree ∩ 3Color is NP-hard.
Yes	No	3Color is undecidable.
Yes	No	There is a polynomial-time reduction from 3Color to Tree.
Yes	No	There is a polynomial-time reduction from Tree to 3Color.

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 2**	Gradescope name:

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.

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 3**	Gradescope name:

Fix the alphabet $\Sigma = \{0, 1\}$. Recall that a *run* in a string $w \in \Sigma^*$ is a maximal non-empty substring in which all symbols are equal. For example, the string `0000010001111111101` consists of exactly six runs: `00000100011111110` = `00000` • `1` • `000` • `1111111` • `0` • `1`.

(a) Let L be the set of all strings in Σ^* that contains at least one run whose length is divisible by 3. For example, L contains the string `00111111000`, but L does not contain the string `1000011`.

Describe both a regular expression for L and a DFA that accepts L.

(b) Let L' be the set of all strings in Σ^* that have the same number of even-length runs and odd-length runs. For example, L' does not contain the string `000011101`, because it has three odd-length runs but only one even-length run, but L' does contain the string `0000111011`, because it has two runs of each parity.

Prove that L' is not regular.

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 4**	Gradescope name:

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 *quality* of such a partition as the minimum, over all k intervals, of the sum of the values in that interval; our goal is to maximize quality. Describe and analyze an algorithm to compute the maximum quality 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 35, which is the quality 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.

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 5**	Gradescope name:

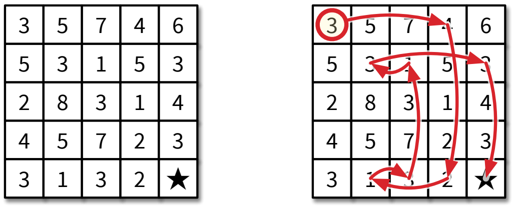
(a) Fix the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. Describe and analyze an efficient algorithm for the following problem: Given a DFA M over Σ, does M reject *any* string? Equivalently, is $L(M) \neq \Sigma^*$?

(b) Recall from Homework 10 that the corresponding problem for NFAs is NP-hard. But any NFA can be transformed into equivalent DFA using the incremental subset construction. So why doesn't your algorithm from part (a) imply that P=NP?

CS/ECE 374 A ✧ Spring 2018 **Final Exam ♫ Problem 6**	Gradescope name:

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.



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.

(scratch paper)

(scratch paper)

(scratch paper)

(scratch paper)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

FeasibleILP: Given a matrix $A \in \mathbb{Z}^{n \times d}$ and a vector $b \in \mathbb{Z}^n$, determine whether the set of feasible integer points $\max\{x \in \mathbb{Z}^d \mid Ax \leq b, x \geq 0\}$ is empty.

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

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

SteamedHams: Aurora borealis? At this time of year, at this time of day, in this part of the country, localized entirely within your kitchen? May I see it?