CS/ECE 374 A ✧ Spring 2018

Midterm 1 Study Questions

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

Questions from Jeff's past exams are labeled with the semester they were used: ⟨⟨*S14*⟩⟩, ⟨⟨*F14*⟩⟩, or ⟨⟨*F16*⟩⟩. Questions from this semester's homework are labeled ⟨⟨*HW*⟩⟩. Questions from this semester's labs are labeled ⟨⟨*Lab*⟩⟩. Some unflagged questions may have been used in exams by other instructors.

How to Use These Problems

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

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

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

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

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

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

Recursion and Dynamic Programming

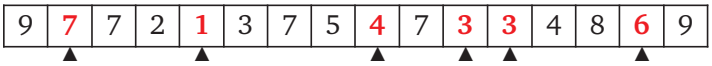
Elementary Recursion/Divide and Conquer

1. ⟨⟨*Lab*⟩⟩

   (a) Suppose $A[1..n]$ is an array of n distinct integers, sorted so that $A[1] < A[2] < \cdots < A[n]$. Each integer $A[i]$ could be positive, negative, or zero. Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists..

   (b) Now suppose $A[1..n]$ is a sorted array of n distinct ***positive*** integers. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. *[Hint: This is* **really** *easy.]*

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



   Describe and analyze a fast algorithm that returns the index of one local minimum. For example, given the array above, your algorithm could return the integer 5, because $A[5]$ is a local minimum. *[Hint: With the given boundary conditions, any array* ***must*** *contain at least one local minimum. Why?]*

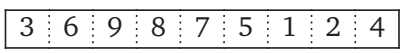
3. ⟨⟨*Lab*⟩⟩ Suppose you are given two sorted arrays $A[1..n]$ and $B[1..n]$ containing distinct integers. Describe a fast algorithm to find the median (meaning the nth smallest element) of the union $A \cup B$. For example, given the input

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

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

4. ⟨⟨*F14, S14*⟩⟩ An array $A[0..n-1]$ of n distinct numbers is ***bitonic*** if there are unique indices i and j such that $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ and $A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,

   is bitonic, but



   is *not* bitonic.

   Describe and analyze an algorithm to find the index of the *smallest* element in a given bitonic array $A[0..n-1]$ in $O(\log n)$ time. You may assume that the numbers in the input array are distinct. For example, given the first array above, your algorithm should return 6, because $A[6] = 1$ is the smallest element in that array.

5. ⟨⟨***F16***⟩⟩ Suppose you are given a sorted array $A[1..n]$ of distinct numbers that has been *rotated* k steps, for some ***unknown*** integer k between 1 and $n-1$. That is, the prefix $A[1..k]$ is sorted in increasing order, the suffix $A[k+1..n]$ is sorted in increasing order, and $A[n] < A[1]$. For example, you might be given the following 16-element array (where $k = 10$):

9	13	16	18	19	23	28	31	37	42	−4	0	2	5	7	8

Describe and analyze an efficient algorithm to determine if the given array contains a given number x. The input to your algorithm is the array $A[1..n]$ and the number x; your algorithm is ***not*** given the integer k.

6. ⟨⟨***F16***⟩⟩ Suppose you are given two unsorted arrays $A[1..n]$ and $B[1..n]$ containing $2n$ distinct integers, such that $A[1] < B[1]$ and $A[n] > B[n]$. Describe and analyze an efficient algorithm to compute an index i such that $A[i] < B[i]$ and $A[i+1] > B[i+1]$. *[Hint: Why does such an index i always exist?]*

7. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a *flip*—insert a spatula under the top k pancakes, for some integer k between 1 and n, and flip them all over.

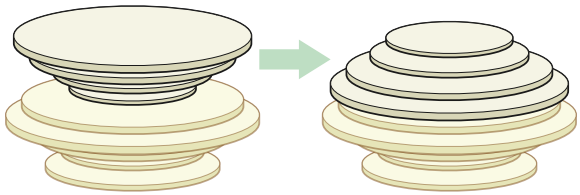


Figure 1. Flipping the top four pancakes.

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

   (b) Now suppose one side of each pancake is burned. Describe an algorithm to sort an arbitrary stack of n pancakes, so that the burned side of every pancake is facing down, using as few flips as possible. *Exactly* how many flips does your algorithm perform in the worst case?

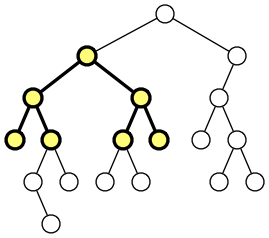
*[Hint: This problem has **nothing** to do with the Tower of Hanoi!]*

8. (a) Describe an algorithm to determine in $O(n)$ time whether an arbitrary array $A[1..n]$ contains more than $n/4$ copies of any value.

   (b) Describe and analyze an algorithm to determine, given an arbitrary array $A[1..n]$ and an integer k, whether A contains more than k copies of any value. Express the running time of your algorithm as a function of both n and k.

Do not use hashing, or radix sort, or any other method that depends on the precise input values, as opposed to their order.

9. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return both the root and the depth of this subtree.



The largest complete subtree of this binary tree has depth 2.

Dynamic Programming

1. ⟨⟨***Lab***⟩⟩ Describe and analyze efficient algorithms for the following problems.

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

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

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

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

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

2. ⟨⟨***F16, HW***⟩⟩ 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 $Score[k]$ 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 $Wait[1..n]$.

3. ⟨⟨***S16***⟩⟩ After the Revolutionary War, Alexander Hamilton's biggest rival as a lawyer was Aaron Burr. (Sir!) In fact, the two worked next door to each other. Unlike Hamilton, Burr cannot work non-stop; every case he tries exhausts him. The bigger the case, the longer he must rest before he is well enough to take the next case. (Of course, he is willing to wait for it.) If a case arrives while Burr is resting, Hamilton snatches it up instead.

   Burr has been asked to consider a sequence of n upcoming cases. He quickly computes two arrays $profit[1..n]$ 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 $i+skip[i]$.

Design and analyze an algorithm that determines the maximum total profit Burr can secure from these n cases, using his two arrays as input.

4. ⟨⟨*S14*⟩⟩ Recall that a *palindrome* is any string that is the same as its reversal. For example, I, DAD, HANNAH, AIBOHPHOBIA (fear of palindromes), and the empty string are all palindromes.

   (a) Describe and analyze an algorithm to find the length of the longest substring (not *subsequence*!) of a given input string that is a palindrome. For example, **BASEESAB** is the longest palindrome substring of BUB**BASEESAB**ANANA ("Bubba sees a banana."). Thus, given the input string BUBBASEESABANANA, your algorithm should return the integer 8.

   (b) ⟨⟨*Lab, F16*⟩⟩ Describe and analyze an algorithm to find the length of the longest subsequence (not *substring*!) of a given input string that is a palindrome. For example, the longest palindrome subsequence of **M**A**H**D**Y**NA**M**ICP**RO**G**R**AMZLET**M**ESHOW**Y**OUT**H**E**M** is MHYMRORMYHM, so given that string as input, your algorithm should output the number 11.

   (c) ⟨⟨*HW*⟩⟩ Any string can be decomposed into a sequence of palindrome substrings. For example, the string BUBBASEESABANANA can be broken into palindromes in the following ways (and many others):

$$\mathtt{BUB} + \mathtt{BASEESAB} + \mathtt{ANANA}$$
$$\mathtt{B} + \mathtt{U} + \mathtt{BB} + \mathtt{A} + \mathtt{SEES} + \mathtt{ABA} + \mathtt{NAN} + \mathtt{A}$$
$$\mathtt{B} + \mathtt{U} + \mathtt{BB} + \mathtt{A} + \mathtt{SEES} + \mathtt{A} + \mathtt{B} + \mathtt{ANANA}$$
$$\mathtt{B} + \mathtt{U} + \mathtt{B} + \mathtt{B} + \mathtt{A} + \mathtt{S} + \mathtt{E} + \mathtt{E} + \mathtt{S} + \mathtt{A} + \mathtt{B} + \mathtt{A} + \mathtt{N} + \mathtt{A} + \mathtt{N} + \mathtt{A}$$

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

5. ⟨⟨*F16*⟩⟩ A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways.

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

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

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

   Describe and analyze an efficient algorithm to determine, given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, whether C is a shuffle of A and B.

6. Suppose we are given an n-digit integer X. Repeatedly remove one digit from either end of X (your choice) until no digits are left. The *square-depth* of X is the maximum number

of perfect squares that you can see during this process. For example, the number 32492 has square-depth 3, by the following sequence of removals:

$$32492 \to \underline{\mathbf{3249}}\mathbf{2} \to \underline{\mathbf{324}}\not{9} \to \not{3}24 \to \not{2}\underline{\mathbf{4}} \to \not{4}.$$

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

7. Suppose you are given a sequence of non-negative integers separated by $+$ and $\times$ signs; for example:

$$2 \times 3 + 0 \times 6 \times 1 + 4 \times 2$$

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

$$2 \times (3 + (0 \times (6 \times (1 + (4 \times 2))))) = 6$$
$$(((((2 \times 3) + 0) \times 6) \times 1) + 4) \times 2 = 80$$
$$((2 \times 3) + (0 \times 6)) \times (1 + (4 \times 2)) = 108$$
$$(((2 \times 3) + 0) \times 6) \times ((1 + 4) \times 2) = 360$$

Describe and analyze an algorithm to compute, given a list of integers separated by $+$ and $\times$ signs, the smallest possible value we can obtain by inserting parentheses.

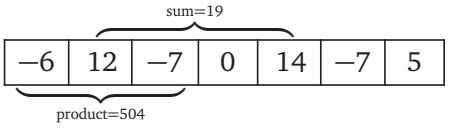
Your input is an array $A[0..2n]$ where each $A[i]$ is an integer if i is even and $+$ or $\times$ if i is odd. Assume any arithmetic operation in your algorithm takes $O(1)$ time.

8. Suppose you are given an array $A[1..n]$ of numbers, which may be positive, negative, or zero, and which are ***not*** necessarily integers.

   (a) Describe and analyze an algorithm that finds the largest sum of of elements in a contiguous subarray $A[i..j]$.

   (b) Describe and analyze an algorithm that finds the largest *product* of of elements in a contiguous subarray $A[i..j]$.

For example, given the array $[-6, 12, -7, 0, 14, -7, 5]$ as input, your first algorithm should return the integer 19, and your second algorithm should return the integer 504.



For the sake of analysis, assume that comparing, adding, or multiplying any pair of numbers takes $O(1)$ time.

[Hint: Problem (a) has been a standard computer science interview question since at least the mid-1980s. You can find many correct solutions on the web; the problem even has its own Wikipedia page! But at least in 2016, a significant fraction of the solutions I found on the web for problem (b) were either significantly slower than necessary or actually incorrect. Remember that the product of two negative numbers is positive.]

9. Suppose you are given three strings $A[1..n]$, $B[1..n]$, and $C[1..n]$.

   (a) Describe and analyze an algorithm to find the length of the longest common subsequence of all three strings. For example, given the input strings

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

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

   (b) Describe and analyze an algorithm to find the length of the shortest common supersequence of all three strings. For example, given the input strings

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

   your algorithm should output the number 21, which is the length of the shortest common supersequence `yzyAxzzxBxxCDxyzEyFzy`.

10. (a) Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

    (b) Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which ***every*** pair of segments intersects.

11. Suppose you are given an $m \times n$ bitmap, represented by an array $M[1..n, 1..n]$ of 0s and 1s. A *solid square block* in M is a subarray of the form $M[i..i+w, j..j+w]$ containing only 1-bits. Describe and analyze an algorithm to find the largest solid square block in M.

12. You and your six-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

    Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it's his turn, Elmo *always* takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

    (a) Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do *not* follow the same greedy strategy as Elmo.

    (b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

(c) Five years later, thirteen-year-old Elmo has become a *much* stronger player. Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against a *perfect* opponent.

13. ⟨⟨***S16***⟩⟩ Your nephew Elmo is visiting you for Christmas, and he's brought a different card game. Like your previous game with Elmo, this game is played with a row of n cards, each labeled with an integer (which could be positive, negative, or zero). Both players can see all n card values. Otherwise, the game is almost completely different.

On each turn, the current player must take the leftmost card. The player can either keep the card or give it to their opponent. If they keep the card, their turn ends and their opponent takes the next card; however, if they give the card to their opponent, the current player's turn continues with the next card. In short, the player that does *not* get the ith card decides who gets the $(i+1)$th card. The game ends when all cards have been played. Each player adds up their card values, and whoever has the higher total wins.

For example, suppose the initial cards are $[3,-1,4,1,5,9]$, and Elmo plays first. Then the game might proceed as follows:

- Elmo keeps the 3, ending his turn.
- You give Elmo the -1.
- You keep the 4, ending your turn.
- Elmo gives you the 1.
- Elmo gives you the 5.
- Elmo keeps the 9, ending his turn. All cards are gone, so the game is over.
- Your score is $1+4+5=10$ and Elmo's score is $3-1+9=11$, so Elmo wins.

Describe an algorithm to compute the highest possible score you can earn from a given row of cards, assuming Elmo plays first and plays perfectly. Your input is the array $C[1..n]$ of card values. For example, if the input is $[3,-1,4,1,5,9]$, your algorithm should return the integer 10.

14. ⟨⟨***F14***⟩⟩ 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.

15. ⟨⟨***F14***⟩⟩ Farmers Boggis, Bunce, and Bean have set up an obstacle course for Mr. Fox. The course consists of a row of n booths, each with an integer painted on the front with bright red paint, which could be positive, negative, or zero. Let $A[i]$ denote the number painted on the front of the ith booth. Everyone has agreed to the following rules:

- At each booth, Mr. Fox ***must*** say either "Ring!" or "Ding!".
- If Mr. Fox says "Ring!" at the ith booth, he earns a reward of $A[i]$ chickens. (If $A[i] < 0$, Mr. Fox pays a penalty of $-A[i]$ chickens.)
- If Mr. Fox says "Ding!" at the ith booth, he pays a penalty of $A[i]$ chickens. (If $A[i] < 0$, Mr. Fox earns a reward of $-A[i]$ chickens.)
- Mr. Fox is forbidden to say the same word more than three times in a row. For example, if he says "Ring!" at booths 6, 7, and 8, then he *must* say "Ding!" at booth 9.
- All accounts will be settled at the end; Mr. Fox does not actually have to carry chickens through the obstacle course.
- If Mr. Fox violates any of the rules, or if he ends the obstacle course owing the farmers chickens, the farmers will shoot him.

Describe and analyze an algorithm to compute the largest number of chickens that Mr. Fox can earn by running the obstacle course, given the array $A[1..n]$ of booth numbers as input.

Greedy algorithms will **not** be covered on this semester's exams;
these problems are provided only for reference.

Greedy Algorithms

Remember that you will receive ***zero*** points for a greedy algorithm, even if it is perfectly correct, unless you also give a formal proof of correctness.

1. ⟨⟨***Lab***⟩⟩ 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.]

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

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

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

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

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

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

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

   (h) 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.

   (i) 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.

2. ⟨⟨*S14*⟩⟩ Binaria uses coins whose values are $1, 2, 4, \ldots, 2^k$, the first k powers of two, for some integer k. As in most countries, Binarian shopkeepers always make change using the following greedy algorithm:

```
MakeChange(N):
  if N = 0
      say "Thank you, come again!"
  else
      c ← largest coin value such that c ≤ N
      give the customer one c cent coin
      MakeChange(N − c)
```

For example, to make 37 cents in change, the shopkeeper would give the customer one 32 cent coin, one 4 cent coin, and one 1 cent coin, and then say "Thank you, come again!" (For purposes of this problem, assume that every shopkeeper has an unlimited supply of each type of coin.)

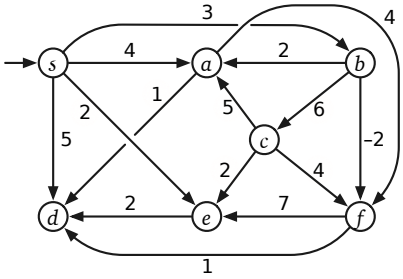
Prove that this greedy algorithm always uses the smallest possible number of coins. *[Hint: Prove that the greedy algorithm uses at most one coin of each denomination.]*

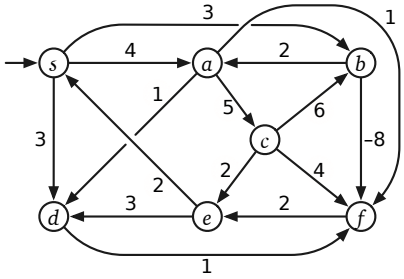
3. Let X be a set of n intervals on the real line. We say that a set P of points *stabs* X if every interval in X contains at least one point in P. Describe and analyze an efficient algorithm to compute the smallest set of points that stabs X. Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, representing the left and right endpoints of the intervals in X. If you use a greedy algorithm, don't forget to ***prove*** that it is correct.

Graph Algorithms

Sanity Check

1. ⟨⟨*S14, F14, F16*⟩⟩ Indicate the following structures in the example graphs below. If the requested structure does not exist, just write NONE. To indicate a subgraph, draw over the entire edge with a heavy black line; your answer should be visible from across the room.





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

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

(c) A shortest-path tree rooted at node s.

(d) The set of all vertices reachable from node c. (Circle each vertex.)

(e) The set of all vertices that can reach node c. (Circle each vertex.)

(f) The strongly connected components. (Circle each one.)

(g) A simple cycle containing vertex s.

(h) The shortest directed cycle.

(i) A depth-first pre-ordering of the vertices. (List the vertices in order.)

(j) A depth-first post-ordering of the vertices. (List the vertices in order.)

(k) A topological order. (List the vertices in order.)

(l) A walk from s to d with the maximum number of edges.

(m) A walk from s to d with the largest total weight.

[On an actual exam, we would only ask about one graph, we would not ask for all these structures, and we would give you several copies of the graph on which to mark your answers.]

Reachability/Connectivity/Traversal

1. Describe and analyze algorithms for the following problems; in each problem, you are given a graph $G = (V, E)$ with unweighted edges, which may be directed or undirected. You may or may not need different algorithms for directed and undirected graphs.

   (a) Find two vertices that are (strongly) connected.

   (b) Find two vertices that are not (strongly) connected.

   (c) Find two vertices, such that neither can reach the other.

   (d) Find all vertices reachable from a given vertex s.

   (e) Find all vertices that can reach a given vertex s.

   (f) Find all vertices that are strongly connected to a given vertex s.

   (g) Find a simple cycle, or correctly report that the graph has no cycles. (A simple cycle is a closed walk that visits each vertex at most once.)

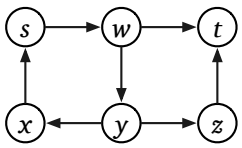
   (h) Find the *shortest* simple cycle, or correctly report that the graph has no cycles.

   (i) Determine whether deleting a given vertex v would disconnect the graph.

   [On an actual exam, we would not ask for all these structures, and we would specify whether the input graph is directed or undirected.]

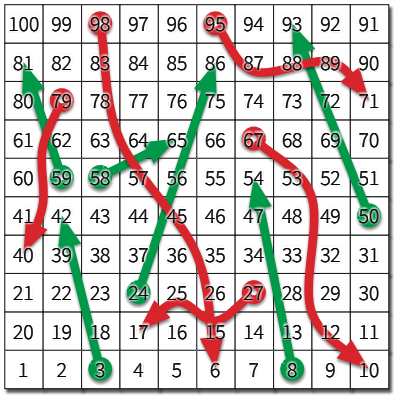
2. ⟨⟨***F14***⟩⟩ Suppose you are given a directed graph $G = (V, E)$ and two vertices s and t. Describe and analyze an algorithm to determine if there is a walk in G from s to t (possibly repeating vertices and/or edges) whose length is divisible by 3.

   For example, given the graph below, with the indicated vertices s and t, your algorithm should return TRUE, because the walk $s \to w \to y \to x \to s \to w \to t$ has length 6.



   [Hint: Build a (different) graph.]

3. ⟨⟨***Lab***⟩⟩ ***Snakes and Ladders*** is a classic board game, originating in India no later than the 16th century. The board consists of an $n \times n$ grid of squares, numbered consecutively from 1 to n^2, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares, always in different rows, are connected by either "snakes" (leading down) or "ladders" (leading up). Each square can be an endpoint of at most one snake or ladder.



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

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

Describe and analyze an algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.

4. Let G be a connected undirected graph. Suppose we start with two coins on two arbitrarily chosen vertices of G, and we want to move the coins so that they lie on the same vertex using as few moves as possible. At every step, each coin *must* move to an adjacent vertex.

   (a) ⟨⟨***Lab***⟩⟩ Describe and analyze an algorithm to compute the minimum number of steps to reach a configuration where both coins are on the same vertex, or to report correctly that no such configuration is reachable. The input to your algorithm consists of a graph $G = (V, E)$ and two vertices $u, v \in V$ (which may or may not be distinct).

   (b) Now suppose there are three coins. 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. *[Hint: Some people already considered this problem in lab.]*

   (c) ⟨⟨***Lab***⟩⟩ Finally, suppose there are *forty-two* coins. Describe and analyze an algorithm to determine whether it is possible to move all 42 coins to the same vertex. Again, *every* coin must move at *every* step. For full credit, your algorithm should run in $O(V + E)$ time.

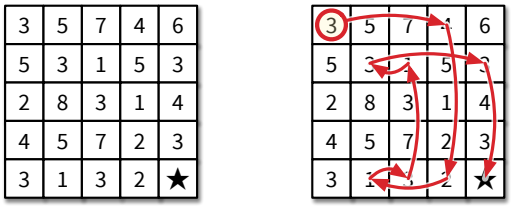
5. A graph (V, E) is bipartite if the vertices V can be partitioned into two subsets L and R, such that every edge has one vertex in L and the other in R.

   (a) Prove that every tree is a bipartite graph.

   (b) Describe and analyze an efficient algorithm that determines whether a given undirected graph is bipartite.

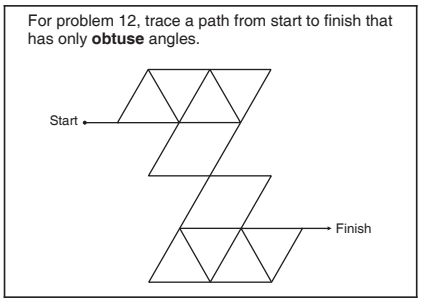
6. ⟨⟨*F14*⟩⟩ A ***number maze*** is an $n \times n$ grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner. On each turn, you are allowed to move the token up, down, left, or right; the distance you may move the token is determined 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.

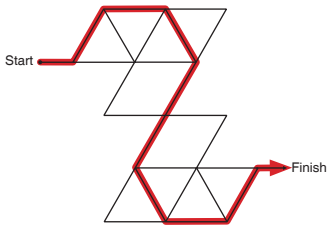
   Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution. For example, given the maze shown below, your algorithm would return the number 8.



A 5 × 5 number maze that can be solved in eight moves.

7. ⟨⟨*F16*⟩⟩ The following puzzle appears in my younger daughter's math workbook.[1] (I've put the solution on the right so you don't waste time solving it during the exam.)





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

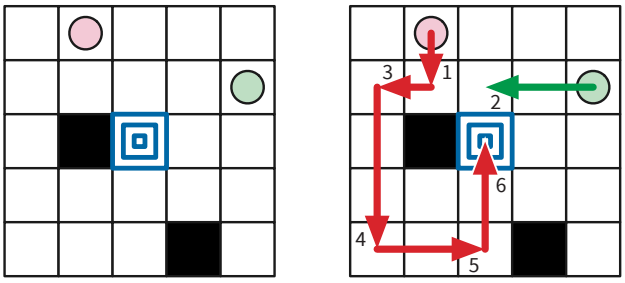
   You are given a connected undirected graph G, whose vertices are points in the plane and whose edges are line segments. Edges do not intersect, except at their endpoints. For example, a drawing of the letter X would have five vertices and four edges; the maze above has 17 vertices and 26 edges. You are also given two vertices Start and Finish.

   Your algorithm should return TRUE if G contains a walk from Start to Finish that has only obtuse angles, and FALSE otherwise. Formally, a walk through G is valid if $\pi/2 < \angle uvw \leq \pi$ for every pair of consecutive edges $u \to v \to w$ in the walk. Assume you have a subroutine that can determine whether the angle between any two segments is acute, right, obtuse, or straight in $O(1)$ time.

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

8. ***Kaniel Dane*** is a solitaire puzzle played with two tokens on an $n \times n$ square grid. Some squares of the grid are marked as *obstacles*, and one grid square is marked as the *target*. In each turn, the player must move one of the tokens from is current position *as far as possible* upward, downward, right, or left, stopping just before the token hits (1) the edge of the board, (2) an obstacle square, or (3) the other token. The goal is to move either of the tokens onto the target square.

   For example, in the instance below, we move the red token down until it hits the obstacle, then move the green token left until it hits the red token, and then move the red token left, down, right, and up. In the last move, the red token stops at the target *because* the green token is on the next square above.



An instance of the Kaniel Dane puzzle that can be solved in six moves.
Circles indicate the initial token positions; black squares are obstacles; the center square is the target.

   Describe and analyze an algorithm to determine whether an instance of this puzzle is solvable. Your input consist of the integer n, a list of obstacle locations, the target location, and the initial locations of the tokens. The output of your algorithm is a single boolean: TRUE if the given puzzle is solvable and FALSE otherwise. The running time of your algorithm should be a small polynomial in n.

9. ⟨⟨***F16***⟩⟩ Suppose you have a collection of n lockboxes and m gold keys. Each key unlocks *at most* one box; however, each box might be unlocked by one key, by multiple keys, or by no keys at all. There are only two ways to open each box once it is locked: Unlock it properly (which requires having a matching key in your hand), or smash it to bits with a hammer.

   Your baby brother, who loves playing with shiny objects, has somehow managed to lock all your keys inside the boxes! Luckily, your home security system recorded everything, so you know exactly which keys (if any) are inside each box. You need to get all the keys back out of the boxes, because they are made of gold. Clearly you have to smash at least one box.

   (a) Your baby brother has found the hammer and is eagerly eyeing one of the boxes. Describe and analyze an algorithm to determine if smashing the box your brother has chosen would allow you to retrieve all m keys.

   (b) Describe and analyze an algorithm to compute the minimum number of boxes that must be smashed to retrieve all the keys. *[Hint: This subproblem should really be in the next section.]*

Depth-First Search, Dags, Strong Connectivity

1. ⟨⟨***Lab***⟩⟩ Inspired by an earlier question, you decided to organize a Snakes and Ladders competition with n participants. In this competition, each game of Snakes and Ladders involves three players. After the game is finished, they are ranked first, second and third. Each player may be involved in any (non-negative) number of games, and the number needs not be equal among players.

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

   You are given the list of players involved and the ranking in each of the m games. Describe and analyze an algorithm to produce an overall ranking of the n players that satisfies the condition, or correctly reports that it is impossible.

2. Let G be a directed acyclic graph with a unique source s and a unique sink t.

   (a) A *Hamiltonian path* in G is a directed path in G that contains every vertex in G. Describe an algorithm to determine whether G has a Hamiltonian path.

   (b) Suppose the vertices of G have weights. Describe an efficient algorithm to find the path from s to t with maximum total weight.

   (c) Suppose we are also given an integer ℓ. Describe an efficient algorithm to find the maximum-weight path from s to t, such that the path contains at most ℓ edges. (Assume there is at least one such path.)

   (d) Suppose several vertices in G are marked *essential*, and we are given an integer k. Design an efficient algorithm to determine whether there is a path from s to t that passes through at least k essential vertices.

   (e) Suppose the vertices of G have integer labels, where $label(s) = -\infty$ and $label(t) = \infty$. Describe an algorithm to find the path from s to t with the maximum number of edges, such that the vertex labels define an increasing sequence.

   (f) ⟨⟨***Lab***⟩⟩ Describe an algorithm to compute the number of distinct paths from s to t in G. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

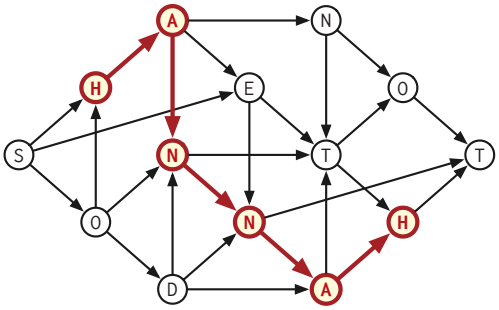
3. Suppose you are given a directed graph G in which ***every edge has negative weight***, and a source vertex s. Describe and analyze an efficient algorithm that computes the shortest path distances from s to every other vertex in G. Specifically, for every vertex t:

   - If t is not reachable from s, your algorithm should report $dist(t) = \infty$.
   - If the shortest-path distance from s to t is not well-defined because of negative cycles, your algorithm should report $dist(t) = -\infty$.
   - If neither of the two previous conditions applies, your algorithm should report the correct shortest-path distance from s to t.

   [Hint: First think about graphs where the first two conditions never happen.]

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

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



Shortest Paths

1. ⟨⟨*F14*⟩⟩ 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.

2. ⟨⟨*F14*⟩⟩ Suppose we are given an undirected graph G in which every *vertex* has a positive weight.

   (a) Describe and analyze an algorithm to find a *spanning tree* of G with minimum total weight. (The total weight of a spanning tree is the sum of the weights of its vertices.)

   (b) Describe and analyze an algorithm to find a *path* in G from one given vertex s to another given vertex t with minimum total weight. (The total weight of a path is the sum of the weights of its vertices.)

3. ⟨⟨*S14, Lab*⟩⟩ You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cites that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving toward each other. But where *exactly* should you meet?

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

   Describe and analyze an algorithm to find the target vertex t that allows you and your friend to meet as quickly as possible.

4. ⟨⟨*F16*⟩⟩ There are n galaxies connected by m intergalactic teleport-ways. Each teleport-way joins two galaxies and can be traversed in both directions. Also, each teleport-way uv has an associated cost of $c(uv)$ galactic credits, for some positive integer $c(uv)$. The same teleport-way can be used multiple times in either direction, but the same toll must be paid every time it is used.

   Judy wants to travel from galaxy s to galaxy t, but teleportation is rather unpleasant, so she wants to minimize the number of times she has to teleport. However, she also wants the total cost to be a multiple of 10 galactic credits, because carrying small change is annoying.

   Describe and analyze an algorithm to compute the minimum number of times Judy must teleport to travel from galaxy s to galaxy t so that the total cost of all teleports is an integer multiple of 10 galactic credits. Your input is a graph $G = (V, E)$ whose vertices are galaxies and whose edges are teleport-ways; every edge uv in G stores the corresponding cost $c(uv)$.

   [Hint: This is ***not*** *the same Intergalactic Judy problem that you saw in lab.]*

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

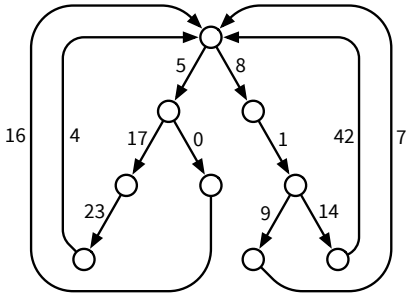


Figure 2. A looped tree.

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

   (b) Describe and analyze a faster algorithm.

6. ⟨⟨***F17***⟩⟩ Suppose you are given a directed graph G with weighted edges, where *exactly one* edge has negative weight and all other edge weights are positive, along with two vertices s and t. Describe and analyze an algorithm that either computes a shortest path in G from s to t, or reports correctly that the G contains a negative cycle. (As always, faster algorithms are worth more points.)

7. 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: This is **not** the same as Alice's bus problem on the homework.]*

8. When there is more than one shortest path from one node s to another node t, it is often convenient to choose a shortest path with the fewest edges; call this the ***best*** path from s to t. Suppose we are given a directed graph G with positive edge weights and a source vertex s in G. Describe and analyze an algorithm to compute best paths in G from s to every other vertex.