362/561 All Problems

Prof. Jared Saia, University of New Mexico

1 Asymptotic Notation

1. Prove that $\log n! = \Theta(n \log n)$ and that $n! = \omega(2^n)$ and $n! = o(n^n)$

2. Assume you have functions f and g, such that $f(n)$ is $O(g(n))$. For each of the following statements, decide whether you think it is true or false and give either a proof or a counterexample

   (a) $\log_2 f(n)$ is $O(\log_2 g(n))$

   (b) $2^{f(n)}$ is $O(2^{g(n)})$

   (c) $f(n)^2$ is $O(g(n)^2)$

2 Probability

1. Let X be a random variable that is equal to the number of heads in two flips of a fair coin. What is $E(X^2)$? What is $E^2(X)$?

2. Problem 7-3 (Alternative quicksort analysis)

3. A frog is jumping across a line of lily pads. It starts at lily pad 1. When the frog is at lily pad i for any $i \geq 1$, it jumps to lily pad $i+1$ with probability 1/2 and to lily pad $i+2$ with probability 1/2.

   (a) Let $p(i)$ be the probability that the frog ever visits lily pad i, for any $i \geq 1$. Write a recurrence relation for $p(i)$. Don't forget the base case(s).

   (b) Use annihilators to solve for a general solution to your recurrence relation.

   (c) Use the base case(s) of your recurrence to solve for an exact solution.

(d) Now, let X be a random variable giving the number of lily pads between lily pad 1 and n that the frog visits, for some number n. Compute $E(X)$ by using: linearity of expectation, indicator random variables, and your solution to the recurrence $p(i)$ that you found above.

4. Solve Problem 5 on the midterm (Cartel)

5. You are buying pizza for n friends. You ask each friend to give you a \$1 coin. When you get to the pizza place, it's closed, so you return the coins to your friends in a random order.

   (a) Consider a single friend. What is the probability that they get back their own coin? Solution: $1/n$

   (b) What is the expected number of friends who will get back their own coins? Solution: $n * 1/n = 1$

   (c) Use Markov's inequality to get an upper bound on the probability that at least 2 of your friends will get their coin back Solution: Let X be the number of friends getting their coin back. $Pr(X \geq 2) \leq 1/2$

   (d) What is the expected number of pairs of friends i and j, such that friend i gets j's coin and friend j gets i's coin? Solution: $\binom{n}{2}(1/n^2)$

   (e) Say that you lose a random subset of $n/2$ coins on the way back, and you randomly distribute the remaining coins to a random subset of $n/2$ of your friends. Now use a union bound to get an upper bound on the probability that at least one of your friends gets their own coin back. Solution: Let S be the random set of friends. For $i \in S$, let ξ_i be the event that person i gets their own coin back. This is the probability that their coin is not lost times the probability that they get it out of the $n/2$ friends chosen. So, $Pr(\xi_i) = (1/2)(2/n) = 1/n$. By a union bound, $Pr(\cup_{i \in S} \xi_i) \leq (n/2)(1/n) = 1/2$

6. The game of Match is played with a special deck of 27 cards. Each card has three attributes: color, shape and number. The possible color values are {red, blue, green}, the possible shape values are {square, circle, heart}, and the possible number values are {1, 2, 3}. Each of the $3 * 3 * 3 = 27$ possible combinations is represented by a card in the deck. A match is a set of 3 cards with the property that for every

one of the three attributes, either all the cards have the same value for that attribute or they all have different values for that attribute. For example, the following three cards are a match: (3, red, square), (2, blue, square), (1, green, square).

(a) If we shuffle the deck and turn over three cards, what is the probability that they form a match? Hint: given the first two cards, what is the probability that the third forms a match?

(b) If we shuffle the deck and turn over n cards where $n \leq 27$, what is the expected number of matches, where we count each match separately even if they overlap? Note: The cards in a match do not need to be adjacent! Is your expression correct for $n = 27$?

(c) Drunken Debutants: Imagine that there are n debutants, each with her own Porsche. After a late and wild party, each debutante stumbles into a Porsche selected independently and uniformly at random (thus, more than one debutant may wind up in a given Porsche). Let X be a random variable giving the number of debutants that wind up in their own Porsche. Use linearity of expectation to compute the expected value of X. Now use Markov's inequality, to bound the probability that X is larger than k for any positive k.

(d) A *big* square with side length 1 is partitioned into x^2 *small* squares, each with side length $1/x$, for some positive x. Then n points are distributed independently and uniformly at random in the big square.

   i. What is the expected number of pairs of points that are both in the same small square? For what value of x is this expected value greater than or equal to 1.

   ii. Use Markov's inequality and the expectation that you computed to bound the probability that there are at least 2 points in the same small square. For what values of x is your bound less than 1?

   iii. If 2 points are in the same small square, what is the maximum distance between them?

   iv. Given all of the above, what would you think the smallest distance between two points would be asymptotically?

(e) **Primes and Probability**. In this problem, you will use the following facts: (1) any integer can be uniquely factored into primes; (2) the number of primes less than any number m is $\Theta(m/\log m)$ (this is the prime number theorem).

We will also make use of the following notation for integers x and y: 1) $x|y$ means that x "divides" y, which means that there is no remainder when you divide y by x. and 2) $x \equiv y \pmod{p}$ means that x and y have the same remainder when divided by p, or in other words, $p|(x-y)$.

i. Show that for any positive integer x, x factors into at most $\log x$ unique primes. Hint: 2 is the smallest prime. Solution: Let f be the number of prime factors of x. Since 2 is the smallest prime, we know that $2^f \geq x$. Taking logs of both sides, we have that $f \leq \log x$.

ii. Let x be a positive integer and let p be a prime chosen uniformly at random from all primes less than or equal to m. Use the prime number theorem to show that the probability that $p|x$ is $O((\log x)(\log m)/m)$.
Solution: x factors into at most $logx$ primes. The number of primes less than y is $\theta(m/\log m)$ by the prime number theorem. Thus, the probability that $p|x$ is no more than $O(logx/(m/\log m))$ or $O((\log x)(\log m)/m)$

iii. Now let x and y both be positive integers less than n, such that $x \neq y$, and let p be a prime chosen uniformly at random from all primes less than or equal to m. Using the previous result, show that the probability that $x \equiv y \pmod{p}$ is $O((\log n)(\log m)/m))$. Solution: $O((\log n)(\log m)/m))$

iv. If $m = \log^2 n$ in the previous problem, then what is the probability that $x \equiv y \pmod{p}$. Hint: If you're on the right track, you should be able to show that this probability is "small", i.e. it goes to 0 as n gets large. Solution: $O(\log\log n/\log n)$

v. Finally, show how to apply this result to the following problem. Alice and Bob both have large numbers x and y where x and y are both at most n, for n a very large number. They want to check to see if their numbers are the same, but Alice does not want to have to send her entire number to Bob.[1] What is an efficient randomized algorithm for Alice and Bob that has "small" probability of failure? How many bits does Alice need to send to Bob as a function of n, and what is

[1]For example, x and y represent large binary files (think terabytes), and Alice and Bob want to check that their files are equal, without Alice having to send her entire file to Bob

the probability of failure, where failure means that this algorithm says x and y are equal, but in fact they are different? Solution: Alice and Bob exchange information on the size of x and y and they let n be the maximum of the two numbers. Then Alice chooses a prime p uniformly at random from all primes no more than $\log^2 n$. Alice then sends Bob p and $x \bmod p$ (this takes $O(\log\log n)$ bits). Bob checks to see if $y \bmod p$ equals $x \bmod p$ and reports the results of this check to Alice. If the two values are equivalent, Alice and Bob assume that the numbers are equal. Otherwise, they assume they are not equal. The probability of failure can be computed by the above questions as $O(\log\log n/\log n)$. Note: You can make this probability arbitrarily small by repeating the experiment with different primes chosen independently.

3 Recursion/Recurrences

1.

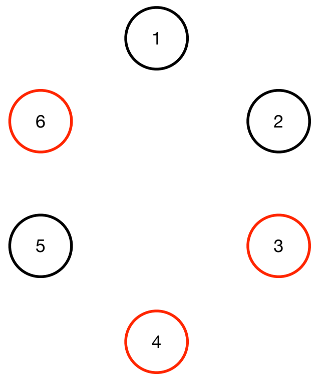
2. Write and solve a recurrence relation giving the number of strings of n digits containing at least one 7. For example, if $n = 5$, then 02307 would be one such string.
   In particular, let $f(n)$ be the number of strings of n digits with at least one 9. First, write an equation $f(n) = ***$, where the *** part contains smaller sub-problems, i.e. on the right, there will be $f(j)$ terms that all have $j < n$. Then give a base case for the recurrence. Finally, use guess and check to solve the recurrence to within $\Theta()$ bounds.

3. You are doing a stress test for the new iPhone XC, and have a collection of identical prototypes. You have a ladder with n rungs. You want to determine the highest rung from which you can drop a prototype without it breaking. You want to do this with the smallest number of drops.

   (a) You have exactly 2 phones. Devise an algorithm that can determine the highest safe rung using $o(n)$ drops.

   (b) You have k phones, for any fixed k. Devise an algorithm that can determine the highest safe rung with the smallest number of drops. If $f(k, n)$ is the number of drops that your algorithm needs, what is $f(k, n)$ asymptotically? Hint: you should ensure that $f(k, n) = o(f(k-1, n))$ for all k.

4. A cat hops on posts arranged in a circle. There are $2n$ posts, with n red and n black. The cat can start at any post, and always hops to the next post in the clockwise direction, until it visits all posts. It "wins" if, at every point during its trip, the number of red posts visited so far is always at least the number of black posts visited so far. In the figure below, the cat wins by starting at post 3, but loses if it starts at any other post.



Prove that the cat can always win: there is always some post, where the cat wins if it starts on that post. Prove this by induction on n for $n \geq 1$. Let your IH be that the cat can always win in the case where there are between 2 and $2(n-1)$ posts. Solution: BC: For 2 posts it is clear the cat can win if it chooses to start on the red post. IH: Suppose the cat can win for $1 \leq k < n$. IS: Consider $2n$ posts. There must be some location x where a red post is followed by a black. Remove these two posts. By the IH, there is some starting point that works for the remaining $2n-2$ posts. This starting point will also work for all n posts, since the two remaining posts increase the red count by 1 before decreasing it!

5. *Silly-Sort* Consider the following sorting algorithm

```
Silly-Sort(A,i,j)
  if A[i] > A[j]
    then exchange A[i] and A[j];
```

```
if i+1 >= j
  then return;
k = floor((j-i+1)/3);
Silly-Sort(A,i,j-k);
Silly-Sort(A,i+k,j);
Silly-Sort(A,i,j-k);
```

(a) Argue (by induction) that if n is the length of A, then Silly-Sort(A,1,n) correctly sorts the input array A[1...n]

(b) Give a recurrence relation for the worst-case run time of Silly-Sort and a tight bound on the worst-case run time

(c) Compare this worst-case runtime with that of insertion sort, merge sort and quicksort.

6. Consider the following function:

```
int f (int n){
  if (n==0) return 2;
  else if (n==1) return 5;
  else{
    int val = 2*f (n-1);
    val = val - f (n-2);
    return val;}}
```

(a) Write a recurrence relation for the *value* returned by f. Solve the recurrence exactly. (Don't forget to check it)

(b) Write a recurrence relation for the *running time* of f. Get a tight asymptotic bound (i.e. Θ) on the solution to this recurrence.

4 Induction

1. Prove by induction that any tree with n nodes has exactly $n-1$ edges. Don't forget to include the Base Case (BC), Inductive Hypothesis (IH) and Inductive Step (IS).

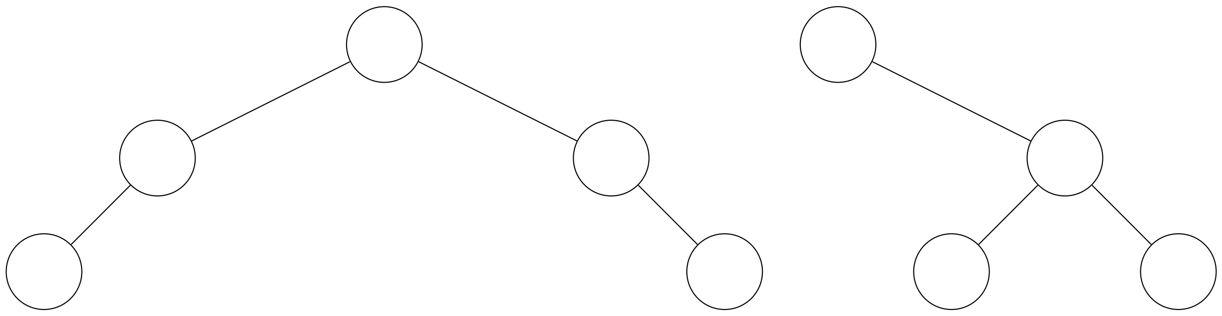
2. Prove by induction that the number of nodes in a binary tree of height h is at most $2^{h+1}-1$. Don't forget to include BC, IH and IS. (Recall that the height of a rooted tree is the maximum number of edges in a path from the root to any leaf node). Hint: Do induction on h. For

the inductive step, what do you get if you remove the root? Solution:
BC: $h = 0$. There can be at most 1 node, which is at most $2^1 - 1 = 1$.
IH: For all $j < h$, a binary tree of height j has at most $2^{j+1} - 1$ nodes.
IS: Consider a binary tree of height h. Remove the root node to obtain at most 2 binary trees of height at most $h - 1$. By the IH, the total number of nodes in each tree is at most $2^h - 1$. Thus the total number of nodes in both trees is at most $2^{h+1} - 2$. Adding the original root node back in, we get that the number of nodes in the original tree is at most $2^{h+1} - 1$.
GRADING: 2 points BC, 2 points correct IH, 6 points inductive step.

3. Prove by induction that any graph with maximum degree at most 3 can be colored with at most 4 colors. Recall that a *coloring* of a graph G is an assignment of a color to each node in G such that the endpoints of each edge in G are assigned different colors. Don't forget to include BC, IH and IS in your proof.
Hint: Perform induction on, n, the number of nodes in G. In the IS, think about how to make G smaller, so that you can use the IH.

4. In a *leaf-balanced binary tree*, any node with 2 children has the same number of leaves in the sub-trees rooted at both children.

   A full binary tree is leaf-balanced; below are two other examples.



   Let n be the number of nodes in the tree. Prove by induction on n that the number of leaf nodes in a leaf-balanced binary tree is always a power of 2.

   Solution: BC: For $n = 0$, there is one leaf node which is a power of 2 since $1 = 2^0$.
IH: For all $j < n$ the number of leaf nodes in a LB tree of size j is a power of 2.
IS: Consider an LB tree with n nodes. If the root node has 1 child

then by the IH, that child has a number of leaf nodes that is a power of 2. If the root node has 2 children that are roots of subtrees T_1 and T_2, then (1) the number of leaves in both trees is equal; and (2) by the IH, the number of leaves in each of T_1 and T_2 is some power of 2, say 2^x. Thus, the total number of leaves is $2 * 2^x = 2^{x+1}$ which is a power of 2.

An *elegant* string is defined recursively as follows:

- The empty string is elegant
- If S_1 and S_2 are elegant strings, then the string aS_1bS_2a is elegant

So, for example, the strings aba, $aababa$ and $aabababaa$ are all elegant.

Prove by induction that any elegant string of length n has exactly $(2/3)n$ a's in it. Don't forget the BC, IH and IS.

Solution: BC: For $n = 0$, the empty string has $2//3 * 0 = 0$ a's in it.
IH: All elegant strings of length $j < n$ contain $(2/3) * j$ a's.
IS: Consider an elegant string S of length $n > 0$. It must be of the form aS_1bS_2a. Since $|S_1|$ and $|S_2|$ are both less than n, by the IH, they contain $(2/3)|S_1|$ and $(2/3)|S_2|$ a's respectively. Hence S has $(2/3)|S_1| + (2/3)|S_2| + 2 = (2/3)(|S_1| + |S_2| + 3) = (2/3)|S|$ a's.

5. A c-tree is a tree with each node colored either red, green or silver that obeys the following rules:

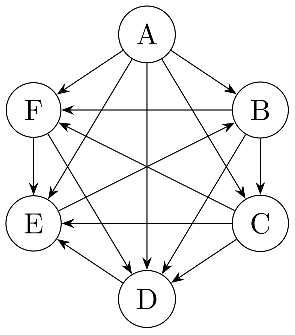
   - Each red node has two children, exactly one of which is green.
   - Each green node has exactly one child, which is not green
   - Silver nodes have no children.

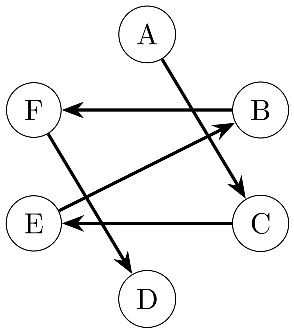
   Let R and S respectively denote the number of red and silver nodes, and n be the total number of nodes. Prove by induction that in any c-tree with $n \geq 1$, $S = R + 1$.

6. A *tournament graph* is a directed graph, $G = (V, E)$ where each pair of nodes has exactly one directed edge between them. I.e., for all $u, v \in V$, either $u \to v$ or $v \to u$ exists in E.

   A *Hamiltonian path* in a directed graph, is a directed path that visits each vertex exactly once. In general, determining whether an arbitrary graph has a Hamiltonian path is NP-Hard; but, here you'll be showing that every tournament graph has a Hamiltonian path.

Below is an example tournament graph on the left, with a Hamiltonian path through it on the right.





- Prove by induction that every tournament graph has a Hamiltonian path.

HINT: Prove this by induction on n. In your IS, let v be any node, and let V_{in} be the nodes with edges pointing into v; and V_{out} be the nodes to which v points. If you can apply the IH to the sub-graph over V_{in} and the sub-graph over V_{out}, can you then piece together everything to create a Hamiltonian path that includes v? A correct inductive proof also gives a recursive algorithm.

Solution: BC: A tournament graph with $n = 1$ vertex has a trivial Hamiltonian path
IH: For all $j < n$, a tournament graph with j vertices has a Hamiltonian path.
IS: Consider a tournament graph with n nodes. let v be any node, and let V_{in} be the nodes with edges pointing into v; and V_{out} be the nodes to which v points. Let G_1 and G_2 be the graphs induced by vertex sets V_{in} and V_{out}. Both $|V_{in}|$ and $|V_{out}|$ are both less than n, and G_1 and G_2 are tournament graphs. Thus, by the IH, there are Hamiltonian paths H_1 and H_2 through G_1 and G_2. We create a Hamiltonian path through G as follows. First, take the path H_1; then take the edge from the last node in that path to v; then take the edge from v to the first node in H_2; then take the path H_2. This path, H, visits each vertex in G exactly once.

7. You are given a graph over n nodes. You must color each node red or black. An edge in the coloring is *good* if the endpoints of that edge are two *different* colors.

Prove by induction on n that there exists some coloring where at least half the edges are good.

Solution: BC: For $n = 1$, any coloring of that nodes ensures half the edges are good.
IH: For all graphs of less than n nodes, there is a coloring ensuring that half the edges are good.
IS: Consider a graph with n nodes. Take out some node v and all edges touching v to get a graph G'. By the IH, G' can be colored in such a way that half its edges are good. Now let c be the color that appears in most of v's neighbors in this coloring. Color node v the opposite of c. This ensures that at least half the edges touching v are good in the new coloring.

5 Dynamic Programming

5.1 Easy DP

1. A thief repeatedly robs the same bank. To avoid capture, he never robs the bank fewer than 10 days after the last robbery. He has obtained information, for the next n days, on the amount of money b_i that is held at the bank on day i.
   But now, the thief is lazy: (1) for each day, there is an integer value giving the amount of work w_i that the thief must perform to rob the bank on that day (due to the amount of security on that day); and (2) there is an additional constraint that the sum of work the thief ever performs is less than some value W.

   Let $r(i, j)$ be the maximum amount of revenue obtainable on days 1 through i, with at most j total work.

   (a) Give a recurrence relation for $r(i, j)$.

   (b) Describe a dynamic program based on this recurrence. What is the runtime of your algorithm?

2. You have n feet of cable to be cut it into pieces for resale. On a given day, pieces of length 1, 3, and 7 can resell for values of v_1, v_2 and v_3. You want to cut the cable into pieces with maximum total resell value. For example, if $n = 14$, and $v_1 = 1$, $v_2 = 4$ and $v_3 = 8$, then the optimal cutting is: 4 pieces of length 3, and 2 pieces of length 1, for a total resell value of $4 * 4 + 2 * 1 = 18$.

You decide to solve this problem with dynamic programming. For any number $i \in [0, n]$, let $m(i)$ be the maximum resell value you can get from optimally cutting a cable of length i. Write a recurrence relation for $m(i)$. Don't forget the base case(s)

3. Now consider a variant of the above problem where the maximum number of cuts you can make is some integer k. As before, pieces of length 1, 3, and 7 resell for values of v_1, v_2 and v_3. Any pieces of other lengths have zero value. For example, if $n = 14$, $k = 1$ and $v_1 = 1$, $v_2 = 4$ and $v_3 = 8$, then the optimal cutting is: 2 pieces of length 7 for a total resell value of $2 * 8 = 16$.

   Write a recurrence relation for a dynamic program for this variant. In particular, for any numbers $i \in [0, n]$ and $j \in [0, k]$, let $m(i, k)$ be the maximum resell value you can get from cutting a cable of length i with at most k cuts. Write a recurrence relation for $m(i, k)$. Don't forget the base case(s).

4. On a dark and quiet fall night, you find yourself climbing a creaky staircase with n stairs. Every stair, $i \in [1, n]$ has a creakiness value (or cost) c_i. In each step, you can go up either 2 or 3 stairs, and your goal is to get to the top of the staircase in a way that minimizes the sum of costs for all the stairs you visit. The bottom and top of the staircase are not creaky and so have no cost. On the last stair, either step size will take you to the top.

   For example, if $n = 4$ and the costs are $[2, 1, 5, 8]$, you can first go up 2 stairs for a cost of 1, and then go up 3 stairs for a cost of 0, for a total cost of 1.

   (a) Consider a greedy algorithm for this problem which always decides the next move based on which of the two choices has least cost. Give an example where this greedy algorithm is not optimal. Solution: Consider the case where $n = 5$, and the c array is $[1, 1, 5, 20, 20]$. The greedy algorithm has cost $1 + 20$ but the optimal is 5.

   (b) Now write a recurrence relation to solve this problem. Don't forget to first define the function in words whose solutions enable solving the big problem. Solution: Let $m(i)$ be the minimum cost way to get to location i. BC: $\forall i < 0$, $m(i) = \infty$. $m(0) = 0$. For all $i \geq 1$ $m(i) = c_i + \min(m(i-3), m(i-2))$

(c) Describe a dynamic program to solve the problem using your recurrence. What are the dimensions of your table? How do you fill it in? What is the final value returned? What is the runtime of your algorithm? Solution: This is a 1 dimensional table of size $n+3$. We fill in the base cases first. Then, we fill in the remaining values from left to right. We return the minimum of $m(n-2)$, $m(n-1)$, and $m(n)$. The runtime is linear.

5. Find the optimal parenthesization for a matrix-chain product whose sequence of dimensions is: $(4, 2, 5, 1, 3)$. Please include the tables used to compute your result.

5.2 Hard DP

1. **Chocolate with Friends.** You have a chocolate bar consisting of n chunks aligned in a single row. Each chunk, i, for $1 \leq i \leq n$ has some positive value v_i (for example, chunks with high nougat content are more valuable than those without!)

   You must break the bar into exactly k *parts* to share with your friends, where each part consists of some number of contiguous, unbroken chunks. The value of a part is the sum of the value of all chunks in that part. Your (greedy) friends chose their parts first, and you get the part remaining, i.e the part of smallest value. Thus, your goal is to break the bar in such a way that you maximize the value of the minimum value part.

   Write a recurrence relation and give a dynamic program to solve this problem. What is the runtime of your algorithm?

   Solution: Let $f(i,j)$ be the maximum over the minimum value piece when dividing the first i chunks into j pieces. Then $f(i,j) = \max_{x:1 \leq x \leq i}(\min(f(i-x, j-1))$, value of part from $i-x+1$ to $j)$. More precisely, $f(i,j) = \max_{x:1\leq x \leq i}(\min(f(i-x, j-1), \sum_{r=i-x+1}^{j} v_r))$ Base Case: $f(i,1) = \sum_{r=1}^{i} v_i$; and for all j, $f(0,j) = 0$.

2. Problem 15-7 "Viterbi Algorithm" (note: this is one example of Dynamic programming used for machine learning)

3. *Dance, Dance Revolution (DDR)* is played on a platform with 4 squares. You are given as input a sequence σ over the symbols A, B, C and D, representing the four squares. In round $i \geq 1$, one of your feet must be on the square $\sigma[i]$. Your feet must always be in different squares,

and you can move at most one foot at the start of each round to any new square. Your left foot starts in square A and right foot in square B.

You are a lazy dancer. So your goal is to maximize the following *laziness score*: the number of rounds in which neither foot moves. Below is an example game play.

σ	A	C	A	D	C	D	B
Feet position	(A,B)	(A,C)	(A,C)	(D,C)	(D,C)	(D,C)	(B,C)
Point?	1	0	1	0	1	1	0

You scored 4 points since there are 4 rounds where neither foot moved.

(a) Write a recurrence relation for the value $m(i, \ell, r)$ which gives the maximum score possible on the first i symbols of σ if your left foot ends in square ℓ and your right foot ends in square r.

Solution: BC: $m(0, \ell, r) = 0$ if $\ell = A$ and $r = B$, otherwise it is $-\infty$. For $i > 0$:

$$m(i, \ell, r) = \begin{cases} -\infty \text{ if } \ell = r \text{ or } \sigma[i] \notin \{\ell, r\}\text{; ELSE:} \\ \max_{\ell', r' : (\ell' = \ell) \vee (r' = r)} m(i-1, \ell', r') + I(\ell' = \ell \wedge r' = r) \end{cases}$$

(b) Describe a dynamic program to return the max score for any input σ of length n based on your recurrence. What are the dimensions of your table? How do you fill it in? What is the final value returned? What is the runtime of your algorithm?

Solution: Let n be the length of σ. Fill in a table that is n by 4 by 4 using the recurrence. Fill in the values in increasing order of i in $m(i, \ell, r$, starting with $i = 0$. Return the maximum value $m(n, \ell, r)$ over all values of ℓ and r. Total runtime is $\Theta(n)$.

4. You are given n balloons, each with a number painted on it. You are asked to pop all the balloons. The numbers on the balloons are started in a *nums* array of length n. If you pop the balloon at index i; and ℓ is the index of the closest un-popped balloon to the left; and r is the index of the closes un-popped balloon to right, you get a number of coins equal to $nums[\ell] * nums[i] * nums[r]$. If ℓ or r goes out of bounds of the array, then treat it as if there is a balloon with a 1 painted on it. Your goal is to return the maximum coins you can collect by bursting the balloons in a smart order.

(a) Give an example input showing that if in every step, you select the balloon that gives you the largest number of coins for that step, this may not maximize the total number of coins.

(b) To set up a dynamic program, describe in words the smaller problem(s) whose solutions can help you solve the big problem.

(c) Write a recurrence relation for the dynamic program.

(d) Describe the dynamic program and give the runtime.

5. The ancient game of *NIM* is played by two players who alternate taking any positive number of stones[2] from one of 3 piles. The person taking the last stone loses. An example game starting with piles of size 13, 9 and 1 is below. Here, the first move of Player 1 is to take the single stone in pile 1, and player 1 eventually wins.

Player Turn	1	2	1	2	1	2
Stones left	(13, 9, 1)	(13, 9, 0)	(13, 2, 0)	(2, 2, 0)	(1, 2, 0)	(1, 0, 0)

In this problem, you will write a dynamic program to determine if Player 1 can force a win for a given input specifying the sizes of the 3 piles.

(a) Describe in words a function whose solutions for smaller problems will help you solve the big problem. Solution: Let $f(x_1, x_2, x_3)$ be 1 if there is a winning strategy for the current player when there are x_1, x_2, x_3 stones left, and 0 otherwise.

(b) Write a recurrence relation for the dynamic program using the function you described above. Solution: Base Case: $f(x_1, x_2, x_3) = 0$ if exactly one of the 3 arguments is 1 and the others are 0. Otherwise,

$$f(x_1, x_2, x_3) = 1 - \min \begin{cases} \min_{i \in [0, x_1 - 1]} m(i, x_2, x_3) \\ \min_{i \in [0, x_2 - 1]} m(x_1, i, x_3) \\ \min_{i \in [0, x_3 - 1]} m(x_1, x_2, i) \end{cases}$$

(c) Describe a dynamic program to solve the problem for any initial input consisting of x_1, x_2, x_3 stones in the piles. What are the dimensions of your table? How do you fill it in? What is the

[2]knuckle bones, dried spiders, or (perhaps apocryphally) candy corns are rumored replacements for stones.

final value returned? What is the runtime of your algorithm if $x_1 = x_2 = x_3 = n$?

Solution: The table will be 3 dimensional with measurements $x_1 + 1$ by $x_2 + 1$ by $x_3 + 1$. We first fill in the values $f(i, 0, 0)$ using the base case and recurrence, so that $f(1, 0, 0) = 0$, and $f(i, 0, 0) = 1$ if $i > 0$. Then we fill in the entries of the array $f(i, j, 0)$ in increasing order of j. Finally, we fill in the $f(i, j, k)$ entries in increasing order of k. We return the entry $f(x_1, x_2, x_3)$. There are $x_1x_2x_3$ entries in teh table and each takes time $O(x_1 + x_2 + x_3)$ to fill in so total runtime is $O(x_1x_2x_3(x_1 + x_2 + x_3))$ or $O(n^4)$ if $x_1 = x_2 = x_3 = n$.

6. **Cake Cutting** After the holidays, you buy a large cake wholesale, in order to cut it into smaller cakes to resell. The original cake is a large rectangle that has width m inches and height n inches. You want to cut it into smaller rectangles of various integer dimensions, so as to maximize the sum of the prices of all these rectangles. You have a lookup array that tells you the resale price, $P[x, y]$, of any x-inch by y-inch rectangle. The resale prices are unusual, depending on aesthetics, customer demand, etc., so don't make any assumptions about them.

   You can make horizontal or vertical cuts across any rectangle with your knife, *but you must cut all the way through the rectangle.*

   Given integers m and n, and an array P, describe a dynamic program to compute the maximum resell value you can obtain if you subdivide the original cake optimally. Be sure to include the following:

   (a) English description of the subproblems (e.g. "minimum edit distance of first i characters of string A and first j characters of string B").

   (b) Mathematical description of the recurrence (e.g. "Base case(s): e(0,i) = i, e(j,0) = j; Recurrence e(i,j) = min(e(i,j-1), e(i-1,j), e(i-1,j-1) + 1 - I(A[i]=B[j])").

   (c) How to compute the final answer using the recurrence (e.g. "Return e(m,n)").

   (d) How to solve the subproblems and analysis of your runtime (e.g. "Fill in a table left to right, top down. Runtime is $O(nm)$")

   Solution: Subproblems: "f(x,y) = Optimal profit one can obtain from a x by y dimensional rectangle." Recurrence. Base Case(s): $f(0, x) = 0$,

$f(y,0)=0$, $f(1,1)=P(1,1)$. The recurrence relation takes the maximum over: no cut, any horizontal cut, and vertical cut. Recurrence: $f(x,y)=\max(P(x,y), \max_{k:1\leq k\leq x-1} f(k,y)+f(x-k,y), \max_{k:1\leq k\leq y-1} f(x,k)+f(x,y-k))$. (you can cut the range of k in half, but it doesn't effect asymptotics). Finally, return $f(m,n)$. To use this recurrence, fill in an array with the f values, starting with the base case, and then going left to right, top down. Each entry takes at most $O(m+n)$ time to fill in. Thus the runtime is $O((m+n)mn)$. This is cubic in the maximum of m and n.

7. Problem about 2D array where can't choose nodes too close to neighbors?

6 Greedy

1. Consider the following alternative greedy algorithms for the activity selection problem discussed in class. For each algorithm, either prove or disprove that it constructs an optimal schedule.

   (a) Choose an activity with shortest duration, discard all conflicting activities and recurse

   (b) Choose an activity that starts first, discard all conflicting activities and recurse

   (c) Choose an activity that ends latest, discard all conflicting activities and recurse

   (d) Choose an activity that conflicts with the fewest other activities, discard all conflicting activities and recurse

2. Now consider a weighted version of the activity selection problem. Imagine that each activity, a_i has a *weight*, $w(a_i)$ (weights are totally unrelated to activity duration). Your goal is now to choose a set of non-conflicting activities that give you the largest possible sum of weights, given an array of start times, end times, and values as input.

   (a) Prove that the greedy algorithm described in class - Choose the activity that ends first and recurse - does not always return an optimal schedule for this problem

   (b) Describe an algorithm to compute the optimal schedule in $O(n^2)$ time. Hint: 1) Sort the activities by finish times. 2) Let $m(j)$ be the maximum weight achievable from activities $a_1, a_2, \ldots, a_j$. 3)

Come up with a recursive formulation for $m(j)$ and use dynamic programming. Hint 2: In the recursion in step 3, it'll help if you precompute for each job j, the value x_j which is the largest index i less than j such that job i is compatible with job j. Then when computing $m(j)$, consider that the optimal schedule could either include job j or not include job j.

3. Consider the following problem.
   INPUT: Positive integers $r_1, \ldots, r_n$ and $c_1, \ldots, c_n$.
   OUTPUT: An n by n matrix A with 0/1 entries such that for all i the sum of the ith row in A is r_i and the sum of the ith column in A is c_i, if such a matrix exists.
   Think of the problem this way. You want to put pawns on an n by n chessboard so that the ith row has r_i pawns and the ith column has c_i pawns. Consider the following greedy algorithm that constructs A row by row. Assume that the first $i-1$ rows have been constructed. Let a_j be the number of 1's in the jth column in the first $i-1$ rows. Now the r_i columns with maximum $c_j - a_j$ are assigned 1's in row i, and the rest of the columns are assigned 0's. That is, the columns that still need the most 1's are given 1's. Formally prove that this algorithm is correct using an exchange argument.

7 Amortized Analysis

1. Walt is making a device for his friend Hector that counts how many times Hector rings a bell. The software for the device requires a binary counter data structure with INCREMENT and RESET operators.

   In class we discussed an INCREMENT algorithm for incrementing a binary counter in $O(1)$ amortized time. Now we want to include a RESET algorithm that sets all the bits in the counter to 0. Below are the algorithms for INCREMENT and RESET. They maintain a bit array B, and a value m. The value m is initially set to 0 and equals the largest index in B set to 1, or 0 if all bits in B are 0.

Algorithm 1 INCREMENT

```
1: $i \leftarrow 0$
2: while $B[i] = 1$ do
3:     $B[i] \leftarrow 0$
4:     $i \leftarrow i + 1$
5: end while
6: $B[i] \leftarrow 1$
7: if $i > m$ then
8:     $m \leftarrow i$
9: end if
```

Algorithm 2 RESET

```
1: for $i \leftarrow 0$ to $m$ do
2:     $B[i] \leftarrow 0$
3: end for
4: $m \leftarrow 0$
```

Let n be the number of operations on this binary counter. Give the following costs as a function of n.

(a) What is the worst-case run time of INCREMENT? Solution: $O(n)$

(b) What is the worst-case run time of RESET? Solution: $O(n)$

(c) Prove that in an arbitrary sequence of calls to INCREMENT and RESET, each call has amortized cost O(1). Hint: Use the accounting method and save up dollars during INCREMENT for future calls to RESET.

Solution: Accounting method. We maintain the invariant that: a) each 1 bit has a dollar on it; and b) that we always have a separate "reset pool" containing at least \$m's to pay for a RESET. We charge INCREMENT \$3 and RESET \$1. On a call to INCREMENT (as in class), we spend \$1 in flipping a single 0 to a 1 and use the dollars stored on the 1 bits to flip them to 0's. Finally, we add the remaining \$1 to the "reset pool". On a call to RESET, we spend the \$1 charged immediately to pay for the call overhead. We then use the money stored in the RESET pool to pay for the cost of setting m bits to 0. Note that the number of dollars in the "reset pool" is at least $2^m \geq m$. Thus there is always enough in this pool to pay for the cost m of setting m bits to zero.

2. Suppose we can insert or delete an element into a hash table in O(1) time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
- After a deletion, if the table is less than 1/4 full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still O(1). Hint: Do not use potential functions.

3. An extendable array is a data structure that stores a sequence of items and supports the following operations.

   - **AddToFront**(x) adds x to the beginning of the sequence.
   - **AddToEnd**(x) adds x to the end of the sequence.
   - **LookUp**(k) returns the kth item in the sequence, or NULL if the current length of the sequence is less than k.

   Describe a simple data structure that implements an extendable array. Your **AddToFront** and **AddToBack** algorithms should take O(1) amortized time, and your LOOKUP algorithm should take O(1) worst-case time. The data structure should use O(n) space, where n is the current length of the sequence.

4. Suppose we are maintaining a data structure under a series of operations. Let $f(n)$ denote the actual running time of the nth operation. For each of the following functions f, determine the resulting amortized cost of a single operation.

   (a) $f(n) = n$ if n is a power of 2, and $f(n) = 1$ otherwise.

   (b) $f(n) = n^2$ if n is a power of 2, and $f(n) = 1$ otherwise.

5. Design and analyze a data structure that maintains a bag of numbers and supports the following operations:

   - INSERT(x) inserts the number x into the bag.
   - DELETE-LARGER-HALF() deletes the largest $\lceil n/2 \rceil$ items from the bag where n is the number of elements in the bag.

   Show how you can implement this so that the amortized cost of both operations is $O(1)$ and so that you can output all n numbers in the bag in $O(n)$ time.

6. Describe and analyze a data structure to support the following operations on an array $A[1 \dots n]$ as quickly as possible. Initially, $A[i] = 0$ for all i.

   - **SetToOne(i)** Given an index i such that $A[i] = 0$, set $A[i]$ to 1.
   - **GetValue(i)** Given an index i, return $A[i]$
   - **GetClosestRightZero(i)** Given an index i, return the smallest index $j \geq i$ such that $A[j] = 0$, or report that no such index exists.

   The first two operations should run in worst-case constant time, and the amortized cost of the third operation should be as small as possible.

7. An ordered stack is one that supports the following operations

   - OrderedPush(x): Pops all items off the stack that are less than x, and then pushes x onto the stack
   - Pop(): If the stack is non-empty, pops the next item off the stack, otherwise returns ERROR

   Suppose you implement an ordered stack using a linked list. Show that you can achieve an amortized cost of $O(1)$ for all operations.

8. Professor Curly conjectures that if we do union by rank, *without path compression*, the amortized cost of all operations is $o(\log n)$. Prove him wrong by showing that if we do union by rank without path compression, there can be m MAKESET, UNION and FINDSET operations, n of which are MAKESET operations, where the total cost of all operations is $\Theta(m \log n)$.

9. Problem 16-2 (Making Binary Search Dynamic)

10. Exercise 17.2-3 (Counter with reset)

11. Exercise 17.3-2 ("Redo Exercise 17.1-3..." Potential method, when i-th operation is a power of 2)

12. Exercise 17.3-7 (Insert and Delete-Larger-Half)

13. Exercise 21.1-3

14. Exercise 21.3-3 ("Give a sequence of m Make-Set, Union and Find-Set operations")

15. Prove the following fact, from Slide 114, amortized lecture, about the Union-Find data structure using PC-Union:

    - For any set X, size(X) $\geq 2^{rank(leader(X))}$

    Prove this by induction on the size of X. Don't forget to include the BC, IH and IS.

8 Graph Theory

9 MST

1. Exercise 23.1-2 ("Professor Sabatier conjectures")

2. Exercise 23.1-3 ("Show that if an edge (u,v) is contained in some minimum spanning tree")

3. Exercise 23.1-4 ("Give a simple example of a connected graph such that the set of edges ...")

4. Professor Moe conjectures that for any connected graph G, the set of edges {(u,v) : there exists a cut (S,V-S) such that (u,v) is a light edge crossing (S, V-S)} always forms a minimum spanning tree. Given a simple example of a connected graph that proves him wrong.

5. Consider a connected graph $G = (V, E)$. Call a subset of edges, F, a *cycle cover* if every cycle in G contains at least one edge in F. In other words, removing the edges of F from G results in an acyclic graph. You want to find a cycle cover, F, of G with *minimum* weight, i.e. the sum of the weight of all edges in F is minimized over all cycle covers. Give an efficient algorithm to solve this, and give the runtime of your algorithm as a function of $n = |V|$ and $m = |E|$. Hint: Think about the maximum-weight spanning tree problem.

6. Professor Matsumoto conjectures the following converse of the safe-edge theorem:
   Let $G = (V, E)$ be a connected, undirected, weighted graph, with weight function w. Let A be a subset of E that is included in some minimum spanning tree of G. Let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a safe edge for A that crosses $(S, V - S)$. Then (u, v) is a light edge for the cut.
   Is this conjecture true? If so, prove it. If not, give a counterexample.

7. Show that if an edge (u, v) is in some minimum spanning tree for a graph G, then (u, v) is a light edge crossing some cut in G.

10 Shortest paths

1. Prove Claim 1 from Single Source Shortest Paths Lecture (SSSP Lecture)

2. Prove Claim 2 from Single Source Shortest Paths Lecture (SSSP Lecture)

3. Problem 24-3 Arbitrage

4. Saia Trucking is a very safety conscious (and algorithm loving) trucking company. Given a pair of cities, they always try to find the *safest* route between that pair. They are thus faced with the following problem.

   There is a directed graph $G = (V, E)$, where the vertices represent cities and the edges represent roads. Each edge has a value associated with it that gives the probability of safe transport on that edge i.e the probability that there will be no accident when driving across that edge. The probability of safe transport along any path in the graph is the *product* of the probabilities of safe transport on each edge in that path.

   The goal is to find a path from a given node s to a given node t that maximizes the probability of safe transport. Describe an efficient algorithm to solve this problem.

11 Floyd-Warshall

1. Exercise 23.2-1 in CLRS, "Run the Floyd Warshall algorithm ..."

2. Recursion Cat on a Tree. You are given a tree with all nodes colored either red or black. Call a path *valid* if at any step of the path, the number of red nodes visited so far is greater than or equal to the number of black nodes visited so far. A cat starts at the root node of the tree and wants to find a valid path to some leaf node.

   For each node v, let $f(v)$ be $-\infty$ if there is no valid path to v. Otherwise, let $f(v)$ be the number of red nodes visited minus the number of black nodes for the path ending at v.

(a) Give a recurrence relation for f. Hint: you may find it useful to let $p(v)$ be the parent of v, for every node v that is not the root. Solution: BC: $f(root) = 1$ if root is red, or $-\infty$ otherwise. For all $v \neq root$:

$$f(v) = \begin{cases} -\infty & \text{If } f(p(v)) = -\infty \text{, or if } f(p(v)) = 0 \text{ and } v \text{ is black} \\ f(p(v)) + 1 & \text{if } v \text{ is red} \\ f(p(v)) - 1 & \text{if } v \text{ is black} \end{cases}$$

(b) Briefly describe a dynamic program that uses the recurrence above to return a valid path from root to some leaf, if such a path exists. Solution: Fill in f values starting at the root node, and continuing down the tree from parent to child. If some leaf node has a non-negative f value, then follow parent pointer up from that node to the root, reverse this path to get a valid path from root to leaf.

3. **Recursion Cat on a Graph** Now recursion cat wants to find valid paths on any graph. Define a *red cycle* to be a cycle that has more red than black nodes in it. Assume you are given a graph, G, with no red cycles. For any pair of nodes, you want to determine if there is a valid path from u to v. Taking inspiration from Floyd-Warshall, you first assign labels 1 to n to all n nodes in the graph. Then you consider paths from nodes u to v that visit intermediate nodes with label at most i. For a given path, let the *black excess* of that path be the maximum over all steps of the path of the number of black nodes minus the number of red nodes at any step. For example, a path of the form R, B, R, B, B, B, R, R has black excess of 2.

Define $f(u, v, i, b) = -\infty$ if there is no path from u to v using intermediate nodes of label at most i, with black excess at most b. Otherwise, define $f(u, v, i, b)$ to be the maximum, over all paths from u to v, with black excess at most b that visit intermediate nodes with label at most i, of the number of red nodes minus the number of black nodes in that path. For example, if the only path from u to v has form R, B, R, B, B, B, R, R, then $f(u, v, n, 2) = 0$.

(a) Write a recurrence relation for $f(u, v, i, b)$. It may help to assume that $-\infty + x = -\infty$ for any value x. Hint: Let the base case(s) be $f(u, v, 0, b)$ for any values of u, v and any b, $0 \leq b \leq n$. It may help to define for a node v, $color(v)$ to be 1 if the node is red, and -1 if the node is black. Solution: Base Case: $f(u, v, 0, b) = -\infty$ if the black excess of the at most 1 hop path from u to v is greater than b, otherwise, $f(u, v, 0, b)$ equals $color(u) + color(v)$. For $i \geq 1$: $f(u, v, i, b) = \max(f(u, v, i-1, b), f(u, i, i-1, b) + f(i, v, i-1, b + f(u, i, i-1, b) - color(i)) - color(i)$

(b) Briefly describe a dynamic program that uses the recurrence above to determine if a valid path exists from u to v for every u and v. What is the runtime as a function of n, the number of nodes, and m the number of edges? Solution: Fill in a n by n by n by n table with the f values, starting with the base case and with increasing i and b. This takes $O(n^4)$ time. For any pair of nodes u and v, return that there's a valid path from u to v iff $f(u, v, n, 0)$ is nonnegative.

12 NP-Hardness

1. **MAX-INSIDE-EDGES.** For a given set of nodes S in a graph G, call an edge of G *inside* S if both endpoints of the edge are nodes in S.

   In the MAX-INSIDE-EDGES problem, you are given a graph $G = (V, E)$, and a number x. You must output the *maximum* number of inside edges in any set S such that $S \subseteq V$ and $|S| \leq x$.

   (a) Prove that MAX-INSIDE-EDGES is NP-Hard by a reduction from one of the following: 3-SAT, VERTEX-COVER, CLIQUE, SUBGRAPH-ISOMORPHISM, INDEPENDENT-SET, 3-COLORABLE, HAMILTONIAN-CYCLE, or TSP. Solution: Reduce from INDEPENDENT-SET. Given a INDEPENDENT-SET(G,k) problem with graph G and a number k, return YES for iff MIN-INSIDE-EDGES(G,k) returns 0.

   (b) Consider the randomized algorithm that picks a subset S of size x, uniformly at random from all subsets of size x. Compute the expected number of inside edges for this algorithm. Let $n = |V|$ and $m = |E|$. Hint: Use indicator random variables and linearity of expectation. Solution: For each edge i, $1 \leq i \leq m$, let X_i be an indicator random variable that is 1 if the i-th edge is an inside edge. The probability that an edge is an inside edge equals the probability that both endpoints are in V'. This probability is $(x/n)((x-1)/(n-1))$. So $E(X_i) = x(x-1)/(n(n-1))$. Let $X = \sum_{i=1}^{m} X_i$. By linearity, $E(X) = \sum_{i=1}^{m} E(X_i) = mx(x-1)/(n(n-1))$.

2. In the **RADIO-TOWERS** problem, you are given (1) a set S of towers; (2) a set T of subsets of towers; and (3) a set of k radio frequencies. You must determine: Is it possible to assign exactly one of k possible radio frequencies to each tower in such a way that each tower in each set in T has a unique radio frequency?

   As an example, let $S = \{a, b, c, d\}$, $T = \{\{a, b, c\}, \{a, d\}, \{a, b, d\}\}$, and $k = 3$. Then the answer is YES since we can assign tower a frequency 1, tower b frequency 2, and towers c and d frequency 3, thereby ensuring that each tower in each set in T has a unique frequency. Prove that this problem is NP-Hard by reduction from a problem we have covered in class. Solution: Reduce from 3-COLORABLE. Nodes become towers, edges become conflicting subsets, $k = 3$.
   GRADING: 2 points choosing correct problem to reduce from, 2 points correct setup and direction of reduction; 2 points mapping nodes to towers; 2 points mapping edges to conflicting subsets; 2 points $k = 3$

3. Imagine someone gives you a polynomial time algorithm to solve 3-SAT. Describe how you could use this algorithm to efficiently find a satisfying assignment for any satisfiable 3-CNF formula.

4. Exercise 34.5-1 (Subgraph Isomorphism)

5. Show that the next problem is NP-Hard via a reduction from one of the following problems: 3-SAT, VERTEX-COVER, INDEPENDENT-SET, 3-COLORABLE, HAMILTONIAN-CYCLE, or CLIQUE.
   WEIGHTED-ITEM-COVER: You are given (1) a set S of weighted items; (2) a set T of subsets of items; and (3) a number W. You are asked: can you choose a subset S' of items in S with total weight of items in S' no more than W, such that every subset in T contains at least one item in S'? As an example, let $S = \{a, b, c, d\}$, $w(a) = w(b) = w(c) = 1$ and $w(d) = 2$; $T = \{\{a, b, d\}, \{c, d\}, \{b, d\}, \{a, c\}\}$; and $W = 3$. Then the answer is YES since we can set $S' = \{a, d\}$, which has total weight 3 and also ensures that every set in T contains at least one item from S'.

6. Exercise 34.5-2 (0-1 Integer Programming)

7. The problem INDEPENDENT-SET asks: "Does there exist a set of k vertices in a graph G with no edges between them?" Show that this problem is NP-Complete. (hint: Reduce from CLIQUE)

8. In the **IGOR** problem, Dr. Frankenstein has tasked his assistant Igor to dig up graves and collect various body parts he has denoted on a list. Each grave contains an assortment of body parts and rubbish. Each buried item has a corresponding weight. Due to the circumstances of the midnight task, Igor must hastily choose a selection of graves to dig up before dawn, and in the darkness, he will have no time to sort through the remains. He must collect ALL buried items from each grave he digs up and place them upon his corpse wagon. Furthermore, his wagon has a weight capacity which shall not be exceeded.

   Prove that it is NP-hard to decide whether Igor can collect the required body parts for Dr. Frankenstein. The input for the IGOR problem consists of a list R (duplicates allowed) of required body parts, and a list of graves $G = [g_1, g_2, \ldots, g_n]$, each of which is a list of buried items. Each item $i \in \bigcup_{j=1}^{n} g_j$ has a weight defined by $\mathrm{w}(i) = w_i, \quad w_i > 0$. Finally, the wagon capacity is denoted by K where $K > 0$. The output is either TRUE or FALSE.

   Example:

   $R = [\text{skull}, \text{torso}, \text{brain}^*, \text{brain}^*, \text{brain}^*]$

   $G = [g_1 = [\text{skull}, \text{brain}],\ g_2 = [\text{skull}, \text{torso}, \text{brain}, \text{brain}],\ g_3 = [\text{torso}, \text{pocketwatch}]]$

   $w(\text{skull}) = 2, w(\text{torso}) = 5, w(\text{brain}) = 1, w(\text{pocketwatch}) = 0.2$

   $K = 15$

   The example input returns TRUE since Igor may choose graves g_1 and g_2. Igor places 6 items totaling 12 weight $\leq K = 15$ into the wagon, fulfilling the requisition R.

   **Dr. Frankenstein understands the importance of finding a good brain.*

   Solution: Reduction from Vertex Cover. Edges are needed body parts. Graves are vertices. Each grave needs to include enough unneeded stuff so that weight of all graves are equal.

9. In the **KEYMASTER** problem, you are trapped in a maze filled with keys and locked doors. Your goal is to gather keys to open corresponding gates and find the exit! The input for the problem is a graph $G = (V, E)$ where all $v \in V$ represent rooms in the maze

and the edges represent corridors between rooms. Let the set of keys be denoted by $K = \{k_1, k_2, \ldots, k_n\}$ and the set of doors be denoted by $D = \{d_1, d_2, \ldots, d_n\}$. Every vertex v has a component defined by $\text{component}(v) = c$, where $c \in K \cup D \cup \{\emptyset\}$. And finally the start and target nodes are respectively denoted by $s, t \in V$ where $\text{component}(s) = \text{component}(t) = \emptyset$.
The goal is to open the doors such that a traversable path from s to t is constructed. Nodes with locked doors are *not* traversable, nor is any node u for which every path from s to u must pass through a locked door. However, if presented with arbitrary nodes x and y both traversable from s where $\text{component}(x) = d_i$ and $\text{component}(y) = k_i$, you may choose to unlock the door at node x. Should you make the choice, set both $\text{component}(x)$ and $\text{component}(y)$ to $\emptyset$. Note that multiple nodes can share the same component. For instance, $\text{component}(a) = \text{component}(b) = d_3$, where $a, b \in V$ and $a \neq b$. Furthermore, modifying the component of a does not influence the component of b, as each nodes component is assigned independently.

Prove that it is NP-Hard to decide whether you can open doors in a sequence such that a traversable path from s to t is constructed. The output is either TRUE or FALSE. Solution: Reduction from Hamiltonian Path. Vertices have keys, edges are replaced with 2 edges with a door between them. Corridor at the end that requires all keys to unlock.

10. 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. 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. The input for the Charon problem consists of the integers k and n and a n vertex graph G describing the pairs of enemies. The output is either TRUE or FALSE.

Solution: I have a reduction from independent set where given a Graph G = (V,E) and k, the size of the independent set. We then create G' which contains 2 disjoint subgraphs, each of which is an exact copy ofG.Let $n = |V|$. We then set the boat size $k' = 2 * (n - k) + 1 + 1$ The$2 * (n - k)$ is for Charon to hold ALL of the people across both subgraphs who are not guaranteed to be friendly to each other. The first +1comes from charon himself in the boat. The second+ is to afford the necessary space to actually carry out transportation without providing excess space. My rationale: The plus 1 passenger just facilitates transportation. We know his boat size ignoring himself: 2(n-k) + 1 is an odd number regarding the people he allows to board so one subgraph is guaranteed to take (n-k) passengers or less. Therefore, in at least 1 subgraph Charon will leave behind k or more people who do NOT eat each other should he complete the task. (which gives credence to the k independent set existence) Charon is provided just enough space for the "non-trusted" people and keeps them upon his boat for the majority duration of transporting while moving a single member of an IS one at a time.

13 Linear Programming

1. Rock, Paper, Scissors is a simple 2 person game. In a given round, both players simultaneously choose either Rock, Paper or Scissors. If they both choose the same object, it's a tie. Otherwise, Rock beats Scissors; Scissors beats Paper; and Paper beats Rock. Imagine you're playing the following betting variant of this game with a friend. When Scissors beats Paper, or Paper beats Rock, the loser gives the winner \$1. However, in the case when Rock beats Scissors, this is called a **smash**, and the loser must give the winner \$10.

   (a) Say you know that your friend will choose Rock, Scissors or Paper, each with probability 1/3. Write a linear program to calculate the probabilities you should use to maximize your expected winnings. Let p_1, p_2, p_3 be the variables associated with your optimal probabilities for choosing Rock, Scissors and Paper respectively. Note: If you want to check your work, there are several free linear program solvers on the Internets: check the Wikipedia page on linear programming.

   (b) Now say that your friend is smart and, also, semi-clairvoyant: she magically knows the exact probabilities you are using and will respond optimally. Write another linear program to calculate the probabilities you should now use in order to maximize your expected winnings. Hint 1: If your opponent knows your strategy, her strategy will be to choose one of the three objects with probability 1. Hint 2: Review the LP we wrote for the shortest paths problem.

14 Count-Min Sketch

1. The count-min sketch discussed in class only provides count estimates of the number of times each item has been seen. What if we want to augment the sketch to also keep track of the x items with largest count estimates for some value x? Describe how you would do this. You may find useful a heap; recall that a heap provides the functionality to (1) *insert* an item with a given value; (2) *increase the key* of an item already in the heap; and (3) *delete min*: delete the item in the heap with minimum key. Assume these heap operations all take $O(\log x)$ time. What is the run time of your augmented count-min sketch?

15 Gradient Descent

1. *X-STREAM Dance Dance Revolution (XDDR)* is played on a pogo-stick while blind-folded. At the beginning of each round, you can hop from your current square to any square. However, the target sequence, σ is not known in advance: the value $\sigma[i]$ is announced only at the end of round i, for each $i \in [1, n]$, where n is the length of σ.

   Your goal is to minimize *cost*: the number of rounds $i \in [1, n]$ in which you hop in a square different than $\sigma[i]$.

   Below is an example game; your cost is 5 because there are 5 rounds where the square you hop in does not match the target square in σ.

σ	A	C	A	D	C	D	D
Pogo position	A	B	B	D	A	C	A
Cost?	0	1	1	0	1	1	1

   In round i, you let $\vec{x}_i$ be a length 4 vector giving a probability distribution over the 4 possible squares on which you will hop, i.e. $\vec{x}_i[1]$, $\vec{x}_i[2]$, $\vec{x}_i[3]$, $\vec{x}_i[4]$ are the probabilities of hopping into squares A, B, C, D respectively.

   Then, for round i, you define $f_i(x_i)$ as your expected cost in round i. In round i, let c_i be a length 4 vector giving the cost outcome: $c_i[j] = 0$ when j matches the square given by $\sigma[i]$ and $c_i[j] = 1$ otherwise. For example, in round i, if $x_i = [1/8, 1/2, 1/8, 1/4]$ and $c_i = [1, 1, 1, 0]$, then your expected cost for this round is 3/4

(a) Give the mathematical expression for f_i as a function of $\vec{x}_i$ and $\vec{c}_i$. Solution: $f_i(\vec{x}_i) = \sum_{j:1 \leq j \leq 4} \vec{c}_i[j]\vec{x}_i[j]$. I.e., $< \vec{x}_i, \vec{c}_i >$

(b) Describe, algebraically, the convex search space κ. What is the diameter, D of κ? Solution: Fix some round i. Then κ is the four dimensional subspace where for all j, $j \in [1, 4]$, $x_i[j] \geq 0$ and $\sum_{j \in [1,4]} x_i[j] = 1$. The diameter is $\sqrt{2}$, since the distance between two vectors in the space is maximized when both of them have the entire probability mass of 1 on separate values of j.

(c) Zinkevich's theorem says that the cost of online gradient descent tracks the cost of the best offline solution, x^*. In particular, if OPT is the cost of the best offline solution, then the cost of our algorithm is at most $OPT + \sqrt{n}DG$. Give a precise 1 line definition of OPT for this problem using the c_i values. Solution: $\min_{j \in [1,4]} \sum_{i=1}^{n} c_i[j]$.

(d) Now, you want to use some history. In particular, you notice that the current square in σ often depends on the last square. So you want to use the outcome of the last round to help set your probability distribution for the current round. To do this, how would you change the convex search space κ? How many dimensions does it now have? Will OPT, G and D likely increase or decrease? Will your algorithm's expected cost increase or decrease?

Solution: Now the search space κ is a 16 dimensional space where there is a probability value for each of the possible outcomes in the last round and each possible choice in this round. So basically there are 4 probability distributions encoded in $\vec{x}_i$. The value of OPT can only decrease since it is now using a value conditioned on information from the past round. The value of G will likely stay the same and D will increase. The online algorithm's cost may decrease since OPT is smaller, but it may increase since G and D are larger.