Final Exam

Instructions:

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- Write your name below and circle your recitation at the bottom of this page.
- Write your solutions in the space provided. If you need more space, indicate so on front and write on the back of the sheet containing the problem. Pages will be separated for scanning and grading.
- Make sure you write your first name on **every** page of the exam after the first page.
- **You are allowed three double-sided, letter-sized sheets with your own notes**. No calculators or programmable devices are permitted. No cell phones or other communication devices are permitted.
- Refrain from discussing this exam until Wednesday afternoon, May 24.

Advice:

- You have 180 minutes to earn a maximum of 180 points. **Do not spend too much time on any single problem.** Read them all first, and attack them in the order that allows you to make the most progress. **This is a long exam, and it may be strategically best to not try to solve every problem.**
- When writing an algorithm, a **clear** description in English will suffice. Using pseudo-code is not required.
- Do not waste time re-deriving facts that we have studied. Simply state and cite them.

Question	Parts	Points
1: True or False	6	18
2: Continuous Optimization	3	15
3: Card Shuffling	1	12
4: Shuttling VIPs	1	25
5: HALF-VERTEX COVER Problem	1	21
6: Vertex Protection	3	30
7: Save Madry-Madry Birds!	1	25
8: Counting on Fashion Trends	4	34
Total:		180

Name: _____

Circle your recitation:	R01 Emanuele Ceccarelli **10 AM**	R02 Shraman Chaudhuri **11 AM**	R03 Mayuri Sridhar **12 PM**	R04 Kai Xiao **1 PM**	R05 Varun Mohan **2 PM**	R06 Isaac Grosof **3 PM**	R07 Devin Neal **11 AM**	R08 Lei "Jerry" Ding **12 PM**	R09 Sagar Indurkhya **1 PM**

Problem 1. [18 points] **True or False** (6 parts)
Please circle **T** or **F** for the following. *No justification is needed.*

(a) [3 points] **T F** There exists a data structure that, for a given set S of n integers (you may assume that each integer fits in a word), we can check whether some integer x is in S in $O(1)$ time, even in the worst case.

Solution: True. Perfect hashing.

(b) [3 points] **T F** If the VERTEX COVER problem is in P, then 3-SAT is also in P.

Solution: True, because we showed in class that 3-SAT can be reduced to vertex cover by a polynomial-time algorithm, and so by cascading the input for 3-SAT first into the reduction and then into the polynomial-time algorithm for vertex cover, a polynomial-time algorithm will result for 3-SAT.

(c) [3 points] **T F** Let $(S^*, V \setminus S^*)$ be the *unique* minimum s-t cut in a graph of capacity F^*. If a vertex $v \in S^*$ then the value of the maximum s-v flow is *strictly* greater than F^*.

Solution: False.

(d) [3 points] **T F** There exists a deterministic algorithm that finds the $n/4$-th largest element of a length n list in $O(n)$ time.

Solution: True.

(e) [3 points] **T F** Suppose an undirected graph $G = (V, E)$ has a vertex cover S of size $\leq k$ and that G contains a vertex v of degree $> k/2$. Then, v has to belong to S.

Solution: False.

(f) [3 points] **T F** Given an arbitrary function f, we want to find a local minimum using gradient descent. There always exists a setting of the step size that will ensure we will eventually get to a local minimum of f.

Solution: False, this is only true if the function is quadratic and can be written as just the sum of the first 3 terms in the Taylor series approximation).

Problem 2. [15 points] **Continuous Optimization** (3 parts)
Consider the one-dimensional function $f(x) = x^3 - 16x + 4$

(a) [5 points] If we run gradient descent starting from $x_0 = -1$ and with an appropriate step size, will a local minimum eventually be found, and if so what are the values of x and $f(x)$ at that minimum?

Solution: Sketch: The minimization will find a local minimum at $x = 4/\sqrt{3}, y = 4 - 128/(3\sqrt{3})$.

(b) [5 points] If instead we run gradient descent starting from $x_0 = -3$ and with an appropriate step size, again will a local minimum eventually be found, and if so what are the values of x and $f(x)$ at that minimum?

Solution: Sketch: The minimization will not find a local minimum. It will diverge to $x \to -\infty, y \to -\infty$.

(c) [5 points] Is the function f β-smooth for some value of β? If so, what is the smallest value of β for which this is true?

Solution: Sketch: Because the second derivative with respect to f is $\frac{\partial^2 f}{\partial x^2} = 6x$, it is unbounded, and so f is *not* β-smooth for any finite value of β.

Problem 3. [12 points] **Card Shuffling** (1 part)
Consider a deck of n cards and the following shuffling method. In each step, we take the top card and with probability $\frac{1}{2}$ we put it back in the top position; otherwise, we randomly choose a position $i \in \{2, 3, 4, \ldots, n\}$ and swap the first card with the i-th card.

Will this shuffling method eventually lead to a perfect shuffling of the deck? Justify your answer.

Note: Recall that a deck is perfectly shuffled if all possible orderings are equally probable.

Solution: Yes, this shuffling method will eventually lead to a perfect shuffling of the deck. We can approach this problem by modeling it as a random walk over the different orderings of the deck. Specifically, consider a weighted, directed graph $G = (V, E, p : E \to \mathbb{R})$ where:

- V is the set of all orderings of the deck.
- $p(u, v) = PR[SWAP\_TOP(u) = v]$ is the probability of going from ordering u to ordering v.
- $E = \{(u, v) | p(u, v) > 0\}$.

which references the following two procedures:

```
SHUFFLE(deck, t):
  for i in 1 to t:
    deck = SWAP_TOP(deck)
  return deck

SWAP_TOP(deck):
  p = random real numbered value from [0, 1].
  if p <= 1/2:
    r = random integer value from [2, 52]
    Swap the top card in the deck with the card in position r.
  return deck.
```

Then we see that when we call $SHUFFLE(deck, t)$ performs a random walk over G of t steps. We will now show that as $t \to \infty$, the probability distribution (over V) of our deck approaches a uniform distribution over all permutations of the cards. The essence of the argument is that a random walk from any starting vertex converges to the stationary distribution and that the unique stationary distribution is uniform.

Now we will show that G is both *strongly-connected* and *aperiodic*:

- To show that G is strongly connected, we can consider any two orderings $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_n]$ and show that we can go from A to B via applications of SWAP_TOP. Here is one way to do this:

1. First swap a_1 and a_x where $a_x = b_1$. This places b_1 at the top of the re-ordered deck.
2. Next, if we want to switch the positions of a_i and a_j for $i, j > 1$, without disturbing the rest of the deck, note that we can apply the following sequence of swaps:

$$swap(a_1, a_i) \rightarrow swap(a_i, a_j) \rightarrow swap(a_j, a_1)$$

Now consider that by repeatedly swapping pairs we can reorder A to B - to see this, consider how we might use the sequence of swaps used by BubbleSort to re-order A into B.

Since we can get from any ordering A to any ordering B with a sequence of TOP-SWAPs, the graph G is thus strongly connected.

- To show that G is aperiodic, if suffices to note that every vertex in G has a self-loop (corresponding to the top card remaining at the top).

Since G is strongly-connected and aperiodic, it will converge to a unique stationary distribution. Thus we simply need to show that the uniform distribution is stationary. Suppose that we have a probability distribution, q over the nodes of G at step t that is the uniform distribution:

$$q_t(u) = Pr[\text{At vertex } u \text{ at step t}] = \frac{1}{n!}\ , \forall u \in V$$

Next, consider any node $u \in V$ and let $S = \{v \in V | (v, u) \in E\}$. Now we note that there are n nodes that have edges leading into u (including u itself through the self-loop); thus $|S| = n$. By definition of the random walk, the probability of being on node u at step $t + 1$ is:

$$\begin{aligned} q_{t+1}(u) &= \sum_{v \in S} q_t(v) p(v, u) && (1) \\ &= (n-1)\left(\frac{1}{2(n-1)}\frac{1}{n!}\right) + \frac{1}{2}\frac{1}{n!} && (2) \\ &= \frac{1}{n!} && (3) \\ &= q_t(u) && (4) \end{aligned}$$

(Note that in equation (2) we divide the sum into the $n - 1$ edges coming from other vertices and the self-loop) Thus we can conclude that the uniform distribution is the unique stationary distribution that is converged to.

Problem 4. [25 points] **Shuttling VIPs** (1 part)
Andrew and Bethany (A and B for short) are shuttle drivers for Gotta Catch 'Em All Airport Shuttles, Inc. The shuttle company needs them to pick up a total of n customers. The two of them need to leave from the airport, pick up all n customers, and return to the airport to drop off the customers. Each customer must be picked up by exactly one of the drivers.

The customers are numbered in order of how important they are. c_1 is the most important customer, c_2 is the second most important, and so on, until c_n, who is the least important. Customers take a lot of pride in their importance, and will get extremely upset if any less important people get on their shuttle before them, so Alice and Bethany must be careful to pick up their customers in the right order — if c_i is on A's shuttle then A cannot subsequently pick up any c_j where $j < i$.

Each customer c_i is located at point p_i, and the airport is located at p_a. You also have access to the distance between each pair of points, $d(p_i, p_j)$, and you can look each such pairwise distance up in constant time.

Design and analyze an algorithm to find the combined schedules of both drivers, *i.e.*, which driver picks up which customer in what order, to minimize the **total distance traveled** by the two drivers.

Hint: Use Dynamic Programming.

Note: To get full credit, your solution should run in $O(n^2)$ time; an $O(n^3)$-time solution will receive partial credit.

Solution: We will solve this problem using dynamic programming. There are a variety of ways to solve it, so we will present three different ways: Shortest Suffix, Shortest Prefix, and Switchover. All of these solutions are valid, and take $O(n^2)$ time.

Let us first observe that there is no reason for a driver to travel except when going from one customer to the next. We want our DP to ensure that all customers are picked up, with the ordering constraint, and that no customer gets picked up twice.

Shortest Suffix:

We will define ShortestSuffix(i, j) (SS for short). SS(i, j) is the shortest distance that A and B can travel starting with A at p_i, B at p_j, where all customers up to $c_{max(i,j)}$ have already been picked up, to pick up the rest of the customers and return to the airport. SS(i, i) will not be considered.

We will allow i or j to be 0, to represent the case where that driver has not yet left the airport. We will define $p_0 := p_a$.

As base cases, we will define $SS(i, n) = SS(n, i) = d(p_n, p_a) + d(p_i, p_a) \forall i, 0 \leq i < n - 1$. If one of the indices is n, then all customers have already been picked up, and we simply need to return to the airport.

Recursively, we will calculate $SS(i, j)$ as follows:

$$\text{Let } k := max(i, j) + 1. \; SS(i, j) = min(SS(i, k) + d(p_j, p_k), SS(k, j) + d(p_i, p_k))$$

When the drivers have picked up all the customers up to $max(i, j)$, the next customer they need to pick up is $max(i, j) + 1 = k$. We know that one of the drivers will pick up that customer,

and we know that once a driver does so, we will arrive at a new subproblem, so we can recurse. The two sides of the min correspond to which driver picks up the next customer.

To extract the optimal distance, we take the minimum over $SS(0,1) + d(p_a, p_1), SS(1,0) + d(p_a, d_1)$. One of the drivers must pick up the first passenger, and so the optimal route must go through one of the corresponding states.

By memoizing the SS values, we can find the optimal distance in $O(n^2)$ time, as there are $O(n^2)$ suproblems, each of which takes $O(1)$ time. Additionally, by adding parent pointers, we will be able to extract the optimal route as well.

Shortest Prefix:

In this solution, rather than considering the optimal way to finish picking up passengers, we will find the optimal way to start picking up passengers.

We will define ShortestPrefix(i, j) (SP for short). SP(i, j) is the shortest distance that A and B can travel to reach the point where S is at p_i, B is at p_j, and all of the customers up to $c_{max(i,j)}$ have been picked up. SP(i, i) will not be considered.

We will allow i or j to be 0, ro represent the case where that driver has not yet left the airport. We will define $p_0 := p_a$.

As base cases, we will deine $SS(0,1) = SS(1,0) = d(p_a, p_1)$. If only the first customer has been picked up, then the only traveling that needs to have occured is from the airport to that customer.

Recursively, there are several cases to consider when calculating $SP(i,j)$:

First, suppose that $i > j + 1$. Then, the only possible preceding state is that A picked up c_{i-1}, then picked up c_i. B cannot have picked up c_{i-1}, due to the ordering constraint.

Thus, $SP(i,j) = SP(i-1,j) + d(p_{i-1}, p_i)$ if $i > j + 1$.

Likewise, $SP(i,j) = SP(i,j-1) + d(p_{j-1}, p_j)$ if $j > i + 1$.

On the other hand, suppose $i = j + 1$. Again, we know that A just picked the least important customer so far, but we don't know where A was coming from. To figure out the optimal previous step, we will find the minimum over all possible predecessors:

$SP(i, i-1) = min_k(SP(k, i-1) + d(p_k, d_i))$

Similarly, $SP(j-1, j) = min_k(SP(j-1, k) + d(p_k, d_j))$

With that, we have completed the recursive definition. In every case, we consider all possible positions the lead driver (the one at a higher index) could have come from, and what the length of the optimal prefix leading to that point would have been.

To extract the optimal distance, we take the minimum over $SP(n,l) + d(p_n, p_a) + d(p_l, p_a)$ and $SP(l,n) + d(p_n, p_a) + d(p_l, p_a)$ for all l. Once a driver reaches c_n, all the customers have been picked up. By considering all possible ending states, we will find the optimal solution.

By memoizing the SP values, we can find the optimal distance in $O(n^2)$ time. There are $O(n^2)$ supbproblems, of which $O(n)$ take $O(n)$ time to calculate, and the other $O(n^2)$ take $O(1)$ time to calculate. The former are the cases where $i = j + 1$ or $j = i + 1$, and the later are the cases

where $i > j+1$ or $j > i+1$. Additionally, by adding parent pointers, we will be able to extract the optimal route as well.

Switchover:

Rather than thinking of our DP state as the positions of both drivers, we can only consider the cases where one driver has just stopped picking up customers, and the other driver has just started picking up drivers, if we think of all customers being picked up in strict order of importance.

More specifically, our DP states will be of the form $Switch_A(i)$ and $Switch_B(j)$. $Switch_A(i)$ represents the shortest distance the drivers must travel to pick up the first i customers, ending with A picking up c_i, and B picking up c_{i-1}. $Switch_B(j)$ will be defined similarly, but with B picking up c_j, and A picking up c_{j-1}.

We will allow i or j to be 0, ro represent the case where that driver has not yet left the airport. We will define $p_0 := p_a$.

As base cases, we will define $Switch_A(1) = Switch_B(1) = d(p_a, p_1)$. If only the first customer has been picked up, then the only traveling that needs to have occured is from the airport to that customer.

Recursively, we will calculate $Switch_A(i)$ as follows:

$$Switch_A(i) = min_{k:1 \leq k < i}(Switch_B(k) + d(p_{k-1}, p_i) + \sum_{m=k}^{i-1} d(p_m, p_{m+1}))$$

$$Switch_B(j) = min_{k:1 \leq k < j}(Switch_A(k) + d(p_{k-1}, p_j) + \sum_{m=k}^{j-1} d(p_m, p_{m+1}))$$

Since the opposite driver picked up the customer immediately preceding the current one, there must be a most recent time in the past where a switch happened. We find the minimal distance over all possible switch points. If a switch happened at k, then the current driver came from $k-1$, and the other driver picked up every passenger in between, which the recursion calculates.

This recursion runs slower than we would like, due to the time necessary to calculate $\sum_{m=k}^{i-1} d(p_m, p_{m+1})$. To save work, we will precompute $s(k, i) = \sum_{m=k}^{i-1} d(p_m, p_{m+1})$ for all possible values k, i, $0 \leq k \leq n-1$, $1 \leq i \leq n$. We will use the precomputed value whenever we need that sum.

To extract the optimal distance, we take the minimum over

$Switch_A(l) + s(l, n) + d(p_{l-1}, p_a), Switch_B(l) + s(l, n) + d(p_{l-1}, p_a)$, for all l, We optimize over all possible ending switchovers, ensuring that we find the optimal value.

By memoizing the $Switch_A$ and $Switch_B$ values, we can find the optimal distance in $O(n^2)$ time. There are $O(n)$ subproblems, which take $O(n)$ time to calculate each, given the precomputation. The precomputation process takes $O(n^2)$ time. Thus, the overall algorithm takes $O(n^2)$. Additionally, by adding parent pointers, we will be able to extract the optimal route as well.

Additional comments:

All of the above solutions maintain an assymetry between A doing something and B doing something. However, this is not necessary, as both drivers take the same distance to travel between two locations. A correct solution may collapse the pair of possibilities.

Each of these solutions had close variants that took n^3 time, which received some partial credit, up to 20/25 points. For the Shortest Suffix solution, we received many solutions that considered k as a separate state of the DP subproblem. For the Shortest Prefix solution, we received many $O(n^2)$ solutions that only proved the algorithm ran in $O(n^3)$. For the Switchover solution, leaving out the precomputation resulted in an $O(n^3)$ algorithm.

Problem 5. [21 points] **HALF-VERTEX COVER Problem** (1 part)
In the HALF-VERTEX COVER problem, we are given a graph $G = (V, E)$ and want to decide whether there exists in G a vertex cover whose size is at most $\frac{|V|}{2}$.

Prove that the HALF-VERTEX COVER problem is NP-hard.

Solution: To show that HALF-VERTEX-COVER is NP-hard, we need to give a reduction from another NP-hard problem. It's important to get the direction of reduction right. The most obvious candidate for this is the well known VERTEX-COVER problem. It is also possible to give a simple reduction from 3-SAT problem by modifying the original 3-SAT $\leq_p$ VERTEX-COVER reduction. Here we give both reductions.

Recall that the VERTEX-COVER problem asks, given a graph $G = (V, E)$ and an integer k, is there a vertex cover of size at most k on G? Given a graph G and k we want to output a graph G' such that VERTEX-COVER(G, k) is YES if and only if HALF-VERTEX-COVER(G') is YES.

Reduction 1: The reduction works in two cases.

1. If $k \geq \lfloor \frac{|V|}{2} \rfloor$, then we first let $G' \leftarrow G$, then add x isolated vertices such that $k' = \frac{|V'|}{2}$. Since we're adding isolated vertices, the size of the minimum vertex cover for G' is the same as G, so $k' = k$. And we now have $|V'| = |V| + x$. Thus we just need to solve $2k = |V| + x$ to find that we need to add $2k - |V|$ isolated vertices to G'. (It's also possible to do this by adding other types of isolated gadgets following the same reasoning).

2. If $k < \frac{|V|}{2}$, in this case we need to increase the size of the minimum vertex cover with respect to the number of nodes. We start with $G' \leftarrow G$ as before and this time we add x isolated triangles (a clique of size 3). Since it takes 2 vertices to cover a tringle, each adds 2 to the minimum cover, but 3 to the number of vertices. Thus we now have $|V'| = |V| + 3x$ and $k' = k + 2x$. We now simply solve $2k' = |V'|$ for x to get a value $x = |V| - 2k$. (It's also possible to do this by adding other types of gadgets following the same reasoning.)

   Another way to do this case would be to add a single clique of size y. Since it takes $y - 1$ vertices to cover this clique, we would have $k' = k + y - 1$ and $|V'| = |V| + y$. Here we can again solve $2k' = |V'|$ for y to get $y = |V| - 2k + 2$.

The correctness argument is that any cover of size $\leq k$ for G yields a cover of size $\leq \frac{|V'|}{2}$ for G' by construction. Similarly by construction, any cover of size $\leq \frac{|V'|}{2}$ for G' must use $\geq \frac{|V'|}{2} - k$ vertices to cover the additional structures in G', which means it uses $\leq k$ vertices to cover G.

Reduction 2: This reduction is simpler as it requires no case analysis. Given an input G and k for VERTEX-COVER, we first let $G' \leftarrow G$ and add x isolated vertices and a single clique of size y to G'. By the same argument as above, the clique affects the size of the minimum vertex cover, so $k' = k + y - 1$, and $|V'| = |V| + x + y$. We still want to have $2k' = |V'|$,

but we only have one equation for two variables. Since the exact size of V' doesn't matter, we can set it to any favorable value to get a second constraint. We will let $|V'| = 2|V|'$ to get the constraint $x + y = |V|$. We can solve these equations to get $x = k - 1$ and $y = |V| - k + 1$. Thus we force that if G contains a cover of size $\leq k$, then G' contains a cover of size $\leq \frac{|V'|}{2}$, and vice versa by the same correctness argument given above.

We now give a reduction from 3-SAT by modifying the original reduction from lecture. Given a 3-SAT formula ϕ over n variables and m clauses, we construct a graph G as input to HALF-VERTEX-COVER as follows.

- Create two vertices P_i and N_i for each variable x_i and draw an edge (P_i, N_i) between every such pair.
- For each clause c_j, create an isolated triangle graph, one node representing each literal in c_j. For each literal x_i or $\neg x_i$ in c_j, draw an edge from the literal to it's corresponding variable gadget N_i if it appears in negation form, otherwise draw an edge to P_i.
- So far, this construction is identical to the one given in lecture. G contains $|V| = 2n + 3m$ vertices. Furthermore, if ϕ is satisfiable, then there exists a cover of size $\leq k = n + 2m$. Since we want to have $k = \frac{|V|}{2}$ all we need to do is add m isolated vertices to G. This does not affect k, but now $V = 2n + 4m = 2k$. Thus the same reduction from lecture with this small modification gives a reduction to HALF-VERTEX-COVER.

Note: Many students incorrectly assumed it only takes 1 vertex to cover a clique of size y, where others assumed it takes all y. Imagine a set of all vertices from a clique of size n except 2. The edge between these two nodes would not be covered, therefore need to take only one of the two. Thus it takes $y - 1$ vertices to cover a clique of size y.

Note: Several students assumed that because HALF-VERTEX-COVER was a special case of VERTEX-COVER where $k = \frac{|V|}{2}$, it must also be hard. It is **not** true that a special case of a hard problem is automatically hard. Take for example the special case where we fix $k = 1$ (constant). In this case it is clearly easy to check all vertices in polynomial time and decide the problem, therefore a special case is not always hard. However, if a special case of a problem is hard, then the general case is also hard - this is exactly what we show with reductions.

Problem 6. [30 points] **Vertex Protection** (3 parts)
Given a graph $G = (V, E)$, we define a subset $S \subseteq V$ of vertices to be a *vertex protector* of G iff every vertex in G is either in S or adjacent to a vertex in S. In the MINIMUM VERTEX PROTECTOR (MVP) problem, given a graph G we want to find a minimum (in cardinality) vertex protector in G.

(a) [5 points] Show that the notion of a vertex protector is *different* from the notion of a vertex cover. Specifically, draw an example of a graph in which the size of a *minimum* vertex cover S is different from the size of a *minimum* vertex protector S'. Don't forget to specify what S and S' are.

Solution: The simplest counterexample is any graph which contains an isolated node. An isolated node must be included in any vertex protector, and must not be included in any minimum vertex cover.

(b) [5 points] Argue that as long as the graph G does not have isolated vertices, the size of the minimum vertex protector is always upper-bounded by the size of the minimum vertex cover.

Solution: We show that for graphs containing no isolated vertices, any vertex cover is also a vertex protector.

By definition of a vertex cover S, $\forall\ (u, v) \in E$, at least one of u and v is contained in S.

Since all nodes have edges in a graph with no isolated vertices, this means that each node is either adjacent to a node in S or is contained in S itself. This satisfies the definition of a vertex protector.

(c) [20 points] Let d be the maximum degree of the graph G. Design a polynomial-time $(d+1)$-approximation algorithm for the MINIMUM VERTEX PROTECTOR (MVP) problem.

Don't forget to carefully justify that your algorithm indeed always delivers a $(d+1)$-approximate solution.

Hint: Your algorithm can be *really* simple.

Solution: Our approximation algorithm will simply return every vertex of the graph. This way, we have that $|ALG| = |V|$. To find a lower bound on the size of $|OPT|$, note that any vertex can protect at most $d+1$ nodes (itself, and a maximum of d neighbors). Therefore any vertex protector must contain at least $|V|/(d+1)$ vertices, so in particular, every minimum vertex protector must contain at least this many vertices.

Therefore, $|ALG|/|OPT| \leq |V|/(|V|/(d+1)) = d+1$, as desired.

Problem 7. [25 points] **Save Madry-Madry Birds!** (1 part)
You are the chief road engineer for a nation of islands. Each island can be connected to other islands via bridges which cost different amounts to build (this can be modeled as a network where the nodes are islands and the weighted edges are possible bridges). You have recently finished planning a network of bridges connecting all the islands; of course, being a good 6.046 student, you found the MST of the island network in order to minimize the total construction cost.

Unfortunately, it turns out that all of the bridges that you built have disrupted the nesting grounds of the rare Madry-Madry bird. However, none of the other possible bridges in the network interfere with the nesting grounds.

Now, environmental groups are demanding that **one** of the planned bridges be replaced with one which does not endanger the Madry-Madry bird. Of course, you still want to minimize the cost of the road network after the adjustment is made!

That is, given a weighted, undirected graph G, and a minimum spanning tree of bridges, T, you want to find the minimum weight tree in G which differs by exactly one edge from T. (If there are multiple such trees, you just need to find any one.)

Note: To get full credit, your solution should run in $O(V^2)$ time; an $O(VE)$-time solution will receive partial credit.

Solution: In order to find a tree which differs by exactly one edge, we must choose an edge f to add to T and an edge e to remove from T. Our goal is minimize the additional cost we add to T, which means we want to minimize $w(f) - w(e)$ such that $(T \setminus \{e\}) \cup f$ is still a spanning tree on G.

$O(VE)$ **Solutions**: There are two general possible approaches,

The first possible solution iterates through each possible edge $e \in T$ and tries removing e from T. This leaves two disconnected components $T_1, T_2 \subset T$. In order to ensure that our solution is spanning and minimizes the final cost, we need to select the minimum edge which links T_1 and T_2 which is not the edge we just removed. We identify the nodes in T_1 and T_2 using BFS on $T \setminus \{e\}$ in $O(V)$ time, then iterate over all other edges in G to identify the minimum weight edge which crosses the cut in $O(E)$ time. We then select the edges e and f which minimizes the additional cost given above. Overall, this takes $O(VE)$ time.

The second possible solution essentially reverses the order of the loops in the first solution. Now, we iterate over each possible edge $f \in E \setminus T$ which we could add to T. For each f, we know that $f \cup T$ forms a cycle, and that the edge $e \in T$ which we should remove to minimize the overall cost while preserving the spanning tree property is e on the cycle which has the heaviest cost besides f. We can find the cycle and identify the heaviest edge on it in $O(V)$ time using DFS. Iterating over all possible f allows us to find the optimal f and e. This solution also takes $O(VE)$ time.

$O(V^2)$ **Solution**:

Note that in the second $O(VE)$ solution, we must perform a substantial amount of repeated work by running DFS to identify the heaviest edge on the induced cycle for every edge we

might add to T. This suggests that we can memoize some of this work! The key to the efficient solution is to precompute the $|V| \times |V|$ table $d[u][v]$ for $u, v \in V$, where $d[u][v]$ is the weight of the heaviest edge on the path between u and v using *only* edges in T. We can fill this table by using BFS on T starting from every possible vertex $v \in V$, using the recurrence $d[v][u] = \max(d[v][u.parent], w(u.parent, u))$. It takes $O(V)$ time to run BFS on T from each vertex, so filling d takes $O(V^2)$ time in total.
Now, we can run the same algorithm as the second $O(VE)$ solution above, but every time we would use DFS to find the heaviest edge in the cycle induced by the edge $f = (u, v)$, we instead perform an $O(1)$ lookup of $d[u][v]$ in the table. This brings the runtime down to $O(E) = O(V^2)$.

Common Pitfalls

- In the first $O(VE)$ solution, some responses only tried adding edges which are adjacent to the edge which was just removed. There may be a lighter edge which connects to the two disconnected components which is not adjacent to either of the endpoints of the edge which was removed.
- Some responses claimed that the first $O(VE)$ solution can run in $O(V^2)$: this is not correct as there may be $O(E)$ edges crossing any given cut induced by removing an edge from T.
- Some responses did not minimize the difference $w(f) - w(e)$. Common mistakes of this type included setting f to always be the lightest edge in $E \setminus T$ or e to always be the heaviest edge in T. Neither of these is necessarily true for the optimal solution.

Problem 8. [34 points] **Counting on Fashion Trends** (4 parts)
You landed an internship position at the wildly popular Theoretically Principled T-Shirt Company. Your job is to keep track of the order/return requests for the company's most popular product: the "Algorithmic Powers...Activate" T-shirt. Specifically, the orders and returns come as a sequence of requests, each of the form (i, k), where i is the ID of a given shop that sells company products and k is the number of T-shirts ordered/returned. (If k is positive, it denotes an order of k T-shirts; if k is negative, it means a return of $|k|$ T-shirts.)

Your task is to maintain for each shop i, the net number x_i of orders at a given time. (Note that one shop can order and return different quantities of T-shirts many times, but you are guaranteed that overall it never returns more T-shirts than it ordered, i.e., that x_i is always non-negative.)

However, due to the volume and frequency of orders — especially, in advance of certain MIT exams — you decide to use a solution that is fast (and space efficient) but provides only an approximate estimate of each x_i, which you indicate as $\widehat{x}_i$.

Specifically, the solution you decided to go with works as follows:

1. Maintain an array A of size s and a hash function h that hashes IDs to $[s]$ and satisfies the simple uniform hashing assumption (SUHA).
2. Initially, all entries of A are equal to 0.
3. Upon receiving a request (i, k), you update A by adding k to its entry $A[h(i)]$, i.e., set $A[h(i)] \leftarrow A[h(i)] + k$. (Remember that k can be negative.)

Now, whenever you want to estimate the current balance of shop i, you output $\widehat{x}_i = A[h(i)]$.

(a) [8 points] Show that, for any i, we always have that $x_i \leq \widehat{x}_i$.

Solution: Observe that

$$\widehat{x}_i = x_i + \sum_{i' \neq i : h(i') = h(i)} x_{i'} \geq x_i,$$

using the fact that $x_{i'} \geq 0$, for each i'.

(b) [8 points] Argue that, for any i, $E[\widehat{x_i} - x_i] \leq \frac{1}{s}|x|_1$, where $|x|_1 = \sum_j |x_j|$.
Hint: What is the probability that $h(j) = h(i)$ for an index $j \neq i$?

Solution: Pre-Solution Steps Following the hint, we notice that for any $j \neq i$, the probability that $h(j) = h(i)$ is $1/s$. This points us in the direction of looking for ways to break the expression $E[\widehat{x_i} - x_i]$ into something where we use that probability. Notice that $\widehat{x_i}$ may be broken into a sum of random variables (that is, the random variables that depend on other stores colliding with store i) which have a distribution that depend on the probability that $h(j) = h(i)$. We apply linearity of expectation on this sum, which leads us close to the result we want to prove.

Solution We have that

$$E[\widehat{x_i}] = x_i + E[\sum_{i' \neq i : h(i') = h(i)} x_{i'}] = x_i + \sum_{i' \neq i} Pr[h(i') = h(i)] x_{i'}.$$

Using linearity of expectation. By the fact that h satisfies SUHA, we have that $Pr[h(i') = h(i)] = \frac{1}{s}$. This gives us that

$$E[\widehat{x_i}] = x_i + \frac{1}{s} \sum_{i' \neq i} x_{i'} \leq x_i + \frac{1}{s}|x|_1,$$

as desired.

(c) [8 points] Prove that if we set $s = \frac{2}{\varepsilon}$ then, with probability at least $\frac{1}{2}$, $\widehat{x_i} \leq x_i + \varepsilon|x|_1$.
Note: You can use here the results from (a) and (b) without proof.
Hint: What can you say about the random variable $\Delta_i = \widehat{x_i} - x_i$?

Solution: If we define $\Delta_i = \widehat{x_i} - x_i$ then by (a) we know that $\Delta_i \geq 0$. Then, by (b) and Markov inequality, we have that

$$Pr[\Delta_i > \varepsilon|x|_1] \leq Pr[\Delta_i > 2E[\Delta_i]] \leq \frac{1}{2}.$$

(d) [10 points] Imagine now that you implement $r = \log \frac{m}{\delta}$ *independent* algorithms as above, where m is the number of different shops. That is, you have r different arrays $A_1, \ldots, A_r$ of size s each, and r different (and independent) hash functions $h_1, \ldots, h_r$, and then you record each request (i, k) by updating $A_j[h_j(i)] \leftarrow A_j[h_j(i)] + k$, for each $1 \leq j \leq r$.

Let $\widehat{x_i}^j$ be the estimate $\widehat{x_i}$ of the value of x_i output by the j-th copy of the algorithm. That is, $\widehat{x_i}^j = A_j[h_j(i)]$. Finally, define $\widehat{x_i}^* = \min_{j=1,\ldots,r} \widehat{x_i}^j$.

Prove that, with probability at least $1 - \delta$, we have that both:

1. $x_i \leq \widehat{x_i}^*$, and
2. $\widehat{x_i}^* \leq x_i + \varepsilon |x|_1$,

simultaneously for all i.

Note: This problem might be more challenging than the others. You can use the results of parts (a), (b), and (c) without proof.

Solution: By (a), we know that $x_i \leq \widehat{x_i}^j$ for all j. So, the same it must be true for their minimum $\widehat{x_i}^*$, always.

Also, observe that if for *at least one* j we have that $\widehat{x_i}^j \leq x_i + \varepsilon |x|_1$, then the same will be true for their minimum $\widehat{x_i}^*$. But by (c) we know that for a fixed j this happen with probability at least $\frac{1}{2}$. So, as all r algorithms are independent, the probability that such event will happen for at least one of them is at least

$$1 - 2^{-r} = 1 - \frac{\delta}{m}.$$

Taking a union bound over all m different values of i, tells us that This tells us that the probability that a particular shop i has $\widehat{x_i}^*$ that doesn't satisfy (2) is $\frac{\delta}{m}$. Taking a union bound over all m different values of i tells us that the probability that any shop i has $\widehat{x_i}^*$ which doesn't satisfy (2) is at most δ. This gives us the desired claim, that with probability $1 - \delta$, all the shops simultaneously satisfy (2).