CS 124 Homework 3: Spring 2022

Your name: Jamin

Collaborators: Elizabeth Li

No. of late days used on previous psets: 4
No. of late days used after including this pset: 6

Homework is due Wednesday at midnight ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignme nt must be a nonnegative integer at most **two** (college)/**four** (extension school).

Try to make your answers as clear and concise as possible; style may count in your grades. Assignments must be submitted in pdf format on Gradescope. If you do assignments by hand, you will need to scan in your results to turn them in.

You can collaborate with other students that are currently enrolled in this course in brainstorming and thinking through approaches to solutions but you should write the solutions on your own: you must wait one hour after any collaboration or use of notes from collaboration before any writing in your own solutions that you will submit.

For all homework problems where you are asked to give an algorithm, you must prove the correctness of your algorithm and establish the best upper bound that you can give for the running time. Generally better running times will get better credit; generally exponential time algorithms (unless specifically asked for) will receive no or little credit. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail, but keep in mind pseudocode does NOT substitute for an explanation. Answers that consist solely of pseudocode will receive little or not credit. Again, try to make your answers clear and concise.

1. Explain how to solve the following two problems using heaps. (No credit if you're not using heaps!)

   (a) **(12 points)** Give an $O(n \log k)$ algorithm to merge k sorted lists with n total elements into one sorted list.

   Solution:

   To merge k sorted lists with n total elements into one sorted list in $O(n \log k)$ runtime, consider the following algorithm:

   1. Initialize a list L of size n and run BUILD-HEAP on the first element of every sorted list to generate a MIN-HEAP H of size k.

   2. Pop the minimum element x from H and append it to L.

   3. Add the next element x' from the list initially containing x to H. If no such x' exists (all elements in the list have been processed), repeat step 2. Proceed to step 4 if

all k sorted lists have been processed.

4. Rebalance the MIN-HEAP H by calling MIN-HEAPIFY on x' or the next smallest element of H if no such x' was added during step 3 (all k sorted lists have been processed and only the elements in H have not been added to L).

5. Repeat steps 2, 3, and 4 until L is full populated with all n elements from our k sorted lists. Return L, which is a sorted list.

Proof of Correctness:

To prove our algorithm's correctness, we will prove its completeness and soundness. To prove completeness, we show that if given k sorted lists with n elements, our algorithm will return a sorted list L with all n elements. To prove soundness, we show that if there exists a valid sorting of the k sorted lists, our algorithm will identify it. We assume that MIN-HEAP and its corresponding functions are correct.

To prove completeness, we show that if given k sorted lists with n elements, our algorithm will return a sorted list L with all n elements. Our algorithm takes the smallest elements of each of the k sorted lists and runs BUILD-HEAP to construct H, so we know that the smallest element popped from the MIN-HEAP and added to the list L will be the smallest element of all n elements. Updating H with the next-smallest element in the list that the first smallest element came from and rebalancing with MIN-HEAPIFY, it is evident that proceeding pops from H will continue to add the next-smallest elements into L, as H is continually updated with the next-smallest elements not yet added to L. Thus, upon termination, L will be a correctly sorted list of n elements returned by our algorithm.

To prove soundness, we show that if there exists a valid sorting of the k sorted lists, our algorithm will identify it. Above we have shown that our algorithm will add all of the n elements in the given k sorted sets into L from the smallest to the largest. Moreover, with each iteration of our algorithm we guarantee that the element added to L will always be the smallest element yet. Thus, if there exists a valid sorting of the n elements within our k sorted lists, our algorithm will find it.

Formally, we prove our algorithm's correctness by inducting on t, the number of iterations of our popping from H, appending to L, and rebalancing. We assert that through every iteration, L will be correctly sorted, and will contain the t smallest elements of all n elements.

Base Case: When $t = 1$, or when we've completed one iteration, only the smallest value of H has been appended to L. Thus, L is sorted correctly and contains the smallest element of all n elements.

Inductive Hypothesis: For all $t < m$, L is properly sorted after t iterations and contains the t smallest elements of all n elements.

Inductive Step: We show that if L is properly sorted after t iterations and contains the t smallest elements of all n element for $t < m$, then L will be properly sorted still after m iterations and L contains the t smallest elements of all n element. From our inductive hypothesis, we know that the first $m-1$ iterations L is properly sorted with the $m-1$ smallest elements of all n elements. Since H contains the next smallest elements of each list not yet in L, the minimum of H that is popped and appended will be the next-smallest of all n elements to be appended to L. Thus, upon completion of this iteration, L contains the m smallest elements of all n elements and is properly sorted.

Thus, by strong induction we have proved our algorithm is correct. **Running Time:** Initializing our list L takes time $O(n)$ and running BUILD-HEAP takes time $O(k)$. Popping the minimum element from H and appending to L takes time $O(1)$. Adding x' and rebalancing our heap with MIN-HEAPIFY takes $O(\log k)$. Since we do this for all n elements added to L, our algorithm has the runtime $O(n \log k)$.

(b) **(12 points)** Say that a list of numbers is k-close to sorted if each number in the list is less than k positions from its actual place in the sorted order. (Hence, a list that is 1-close to sorted is actually sorted.) Give an $O(n \log k)$ algorithm for sorting a list of n numbers that is k-close to sorted.

Solution:

To sort a list of n numbers that is k-close to sorted in $O(n \log k)$ runtime, consider the following algorithm:

1. Initialize a list L of size n and run BUILD-HEAP on the first k elements of the k-close to sorted list to generate a MIN-HEAP H of size k.

2. Pop the minimum element x from H and append it to L.

3. Add the next element x' from k-close to sorted list to H. If no such x' exists (all elements in list have been processed), proceed to step 4.

4. Rebalance the MIN-HEAP H by calling MIN-HEAPIFY on x' or the next smallest element of H if no such x' was added during step 3 (all k sorted lists have been processed and only the elements in H have not been added to L).

5. Repeat steps 2, 3, and 4 until L is full populated with all n elements from our k-close to sorted list. Return L, which is a sorted list.

Proof of Correctness:

To prove our algorithm's correctness, we will prove its completeness and soundness.

To prove completeness, we show that if given a k-close to sorted list with n elements, our algorithm will return a sorted list L with all n elements. To prove soundness, we show that if there exists a valid sorting of the k-close to sorted list, our algorithm will identify it. We assume that MIN-HEAP and its corresponding functions are correct.

To prove completeness, we show that if given a k-close to sorted list with n elements, our algorithm will return a sorted list L with all n elements. Our algorithm takes the first k elements of the k-close to sorted list and runs BUILD-HEAP to construct H. Since our algorithm puts the first k elements of the k-close to sorted list into H, we know the smallest element of the list, which is fewer than k positions away from the first position, is in the first k elements and is included in H. Thus, we know that the smallest element popped from the MIN-HEAP and added to the list L will be the smallest element of all n elements. Updating H with the next element k-close to sorted list guarantees that the second smallest element is added to H. Proceeding, it is evident that proceeding pops from H will continue to add the next-smallest elements into L, as H is continually updated with the next-smallest elements not yet added to L. Thus, upon termination, L will be a correctly sorted list of n elements returned by our algorithm.

To prove soundness, we show that if there exists a valid sorting of the k-close to sorted list, our algorithm will identify it. Above we have shown that our algorithm will add all of the n elements in the given k-close to sorted list into L from the smallest to the largest. Moreover, with each iteration of our algorithm we guarantee that the element added to L will always be the smallest element yet. Thus, if there exists a valid sorting of the n elements within our k-close to sorted list, our algorithm will find it.

Formally, we prove our algorithm's correctness by inducting on t, the number of iterations of our popping from H, appending to L, and rebalancing. We assert that through every iteration, L will be correctly sorted, and will contain the t smallest elements of all n elements.

Base Case: When $t = 1$, or when we've completed one iteration, only the smallest value of H has been appended to L and H contains the smallest value of n elements as the smallest element of the list, which is fewer than k positions away from the first position, is in the first k elements and is included in H. Thus, L is sorted correctly and contains the smallest element of all n elements.

Inductive Hypothesis: For all $t < m$, L is properly sorted after t iterations and contains the t smallest elements of all n elements.

Inductive Step: We show that if L is properly sorted after t iterations and contains the t smallest elements of all n element for $t < m$, then L will be properly sorted still after m iterations and L contains the t smallest elements of all n element. From our inductive hypothesis, we know that the first $m - 1$ iterations L is properly sorted with the $m - 1$ smallest elements of all n elements. Since H contains the next smallest elements of each list not yet in L, the minimum of H that is popped and appended will be the next-smallest of all n elements to be appended to L. Thus, upon completion of this iteration, L contains the m smallest elements of all n elements and is properly sorted.

Thus, by strong induction we have proved our algorithm is correct.

Running Time:

Initializing our list L takes time $O(n)$ and running BUILD-HEAP takes time $O(k)$. Popping the minimum element from H and appending to L takes time $O(1)$. Adding x' and rebalancing our heap with MIN-HEAPIFY takes $O(\log k)$. Since we do this for all n elements added to L, our algorithm has the runtime $O(n \log k)$.

2. **(0 points, optional)**[1] Consider the following generalization of binary heaps, called d-heaps: instead of each vertex having up to two children, each vertex has up to d children, for some integer $d \geq 2$. What's the running time of each of the following operations, in terms of d and the size n of the heap?

   (a) delete-max()

   (b) insert(x, value)

   (c) promote(x, newvalue)

   The last operation, promote(x, newvalue), updates the value of x to *newvalue*, which is guaranteed to be greater than x's old value. (Alternately, if it's less, the operation has no effect.)

3. The year is 2124 and Harvard SEAS has n campuses spread across the greater Boston area. The campuses are all connected by special Harvard roads: for every pair of campuses, there is a road straight from one to the other, not passing through other campuses or intersecting other roads. (There may be underpasses and tunnels, such that roads that cross each other don't necessarily have an intersection.) However, these roads were built decades ago and they are in disrepair. You would like to repair D of the roads. Repairing a road requires closing it down, but students still need a way to go from each campus to each other campus. Moreover, each road r has a positive integer length $l(r)$ and the cost to repair a road is proportional to its length.

   (a) **(5 points)** As a function of n, what's the greatest value of D for which it's possible to conduct D such repairs simultaneously?

   Solution:

   If there are n campuses each connected by a single road, then there are a total of $\binom{n}{2}$ roads connecting campuses. In order for students to still be able to go from each campus to each other campus, a minimum of $n-1$ roads are necessary. As such, as a function

[1]We won't use this question for grades. Try it if you're interested. It may be used for recommendations/TF hiring.

of n, the greatest value of D which is possible to conduct D repairs simultaneously is given by:

$$\begin{aligned} f(n) &= \binom{n}{2} - (n-1) \\ &= \frac{n!}{2!(n-2)!} - n - 1 \\ &= \frac{n(n-1)}{2} - (n-1) \\ &= \frac{(n-1)(n-2)}{2} \end{aligned}$$

One way of picturing this is that the Harvard SEAS campuses are a fully connected undirected graph with n vertices representing each campus and $\binom{n}{2}$ edges representing each road. In order for students to still be able to travel to each campus while repairs happen, the graph must remain connected. For an undirected graph of n vertices to be connected, there must be a minimum of $n-1$ edges. We can think of this as removing all edges except the ones coming into a single vertex. The one vertex is connected to all other $n-1$ vertices, so the graph is connected. As such, the maximum number of roads we can repair at once D, or the maximum number of the edges we can remove from the graph so that it still remains connected is $\binom{n}{2} - (n-1)$.

(b) **(20 points)** Assuming D is between 0 and that value, give an algorithm to find the cheapest set of D road repairs. A $\Theta(n^2 \log n)$ algorithm will receive only partial credit.

Solution. To find the cheapest set of D road repairs, consider the following algorithm:

1. Construct a graph $G = (V, E)$ with $|V| = n$ vertices to represent each of the Harvard's SEAS campuses, and $|E| = \binom{n}{2}$ undirected edges to represent the roads connecting each campus as discussed in part (a). Assign edge weights $l(r)$ to each edge for the cost of repairing that road. Create a list N to track the weights of non-tree edges, and create a list R to track the cheapest set of road repairs.

2. Multiply the weight $l(r)$ on each edge by -1, and run Prim's Algorithm with a Fibonacci Heap implementation on G, generating an MST.

3.Multiply the weight of all of the edges again iterate through all of the them. Add the weights of the edges not included in the MST to W (these are the lowest-cost roads in G).

4. Next, run a selection algorithm with lower bound linear time as discussed in section to find the Dth smallest edge weight in W and iterate through W, adding the corresponding edges to R if the weight of the edge is less than the Dth smallest edge weight.

5.If $|R|$ is less than D after iterating through all of W, add the next $D - |R|$ smallest elements from W to R, using the same selection algorithm as above.

6. Return R, the cheapest set of D road repairs.

Proof of Correctness:

We prove correctness by proving our algorithm outputs the cheapest set of D road repairs, and by proving that if there is a cheapest set of D road repairs, our algorithm is guaranteed to find it.

First, we confirm that our graph construction is guaranteed to model the described Harvard's SEAS campus correctly because of the specifications of the campus description. Using campuses as vertices and roads as undirected edges, with edge weights as road repair costs, we have modeled the campus correctly.

Next, we prove that our algorithm is guaranteed to output the cheapest D road repairs. First, by multiply the weight of each edge before running Prim's algorithm, our MST contains the roads with the highest cost of repair due to the negation. As such, the list W with non-tree edges represent the edges with the smallest weight in the graph and therefore the cheapest repair costs. From here, selecting the D smallest of these edge weights guarantees we select the D cheapest roads to repair. As explained in part (a), students are still able to travel the campus as removing D edges still allows the graph to be connected. As such, we've shown that our algorithm correctly outputs the cheapest set of D edges.

Finally, we prove that if there exists a cheapest set D of road repairs, our algorithm is guaranteed to find it. By the construction of our algorithm, the edges inputted into W will be the smallest weights for edges in the graph and therefore the cheapest repairs. Thus, by iterating through W as described, R is guaranteed to have the D cheapest road repairs of G, proving that if there exists a cheapest set D of road repairs, our algorithm will find it.

Runtime: Constructing G takes $O(n + \binom{n}{2}) = O(n^2)$ time as there are n vertices and $\binom{n}{2}$ edges. Initializing W and R take $O(\binom{n}{2})$ which is on the order of $O(n^2)$ time as well. Multiplying the weights on each edge also takes $O(n^2)$ time, and running Prim's using a Fibonacci heap takes $O(|E| + \log |V|) = O(n^2 + \log n)$ time. Multiplying the edges again takes $O(n^2)$ time, as does adding the weights of the edges to. The selection algorithm takes $O(n^2)$ time since it's linear and we have on the order of n^2 edges, adding the appropriate edges to R takes $O(n^2)$ time. Thus, runtime of the algorithm can be generalized to $O(n^2)$.

4. **(5 points)** Give a family of set cover problems where the set to be covered has n elements, the minimum set cover is size $k = 3$, and the greedy algorithm returns a cover of size $\Omega(\log n)$. That is, you should give a description of a set cover problem that works for a set of values of n that grows to infinity – you might begin, for example, by saying, "Consider the set $X = \{1, 2, 3, \ldots, 2^b\}$ for any $b \geq 10$, and consider subsets of X of the form...", and finish by saying "We have shown that for the example above, the set cover returned by the greedy algorithm is of size $b = \Omega(\log n)$." (Your actual wording may differ substantially, of course, but this is the sort of thing we're looking for.) Explain briefly how to generalize your construction

for other (constant) values of k. (You need not give a complete proof of your generalization, but explain the types of changes needed from the case of $k = 3$.)

Solution: Consider the family of sets with sizes belonging to $X = \{6, 12, \ldots, 3 \cdot 2^p\}$ for any $p > 0$. Then the family contains sets with sizes $n = 3 \cdot 2^p$ elements where $p > 0$. For each set size n, consider the set S of the size $\lceil \log n \rceil + 3$ containing subsets generated in the following manner:

$\lceil \log n \rceil$ disjoint subsets of size $\frac{n}{2}, \frac{n}{4}, \ldots, \frac{n}{2^p}, \lceil \frac{n}{2^{p+1}} \rceil, \lceil \frac{n}{2^{p+2}} \rceil$ covering all elements of the set.

3 disjoint subsets of size $\frac{n}{3}$ covering all elements of the set.

First, note that the minimum set cover is always 3. This set cover is composed of the 3 disjoint subsets of size $\frac{n}{3}$ covering all elements of the set in all cases except the case of minimum n when $n = 6$. In this specific case, the disjoint sets of $\frac{n}{2}, \lceil \frac{n}{2^{p+1}} \rceil, \lceil \frac{n}{2^{p+2}} \rceil$ also form a set cover of 3. When running the greedy algorithm to identify a set cover for S, since the algorithm selects the set containing the largest number of uncovered elements, the algorithm will first select to include the set of size $\frac{n}{2}$ in the set cover. Next, the algorithm will select to include the set of size $\frac{n}{4}$ as it has the largest number of uncovered elements as it is disjoint from the first selected set. By comparison, the sets of size $\frac{n}{3}$ would have $\frac{n}{6}$ uncovered elements. The greedy algorithm would then continually choose the sets of size $\frac{n}{2^p}$ to include in the set cover, never selecting to include any of the sets of size $\frac{n}{3}$ because they never contain more uncovered elements than the sets of size $\frac{n}{2^p}$. Thus, to cover the entire set of n elements, the algorithm will choose all $\lceil \log n \rceil$ subsets of size $\frac{n}{2}, \frac{n}{4}, \ldots, \frac{n}{2^p}, \lceil \frac{n}{2^{p+1}} \rceil, \lceil \frac{n}{2^{p+2}} \rceil$. As such, this set cover would have size $\lceil \log n \rceil$, which is a set cover of size $\Omega(\log n)$ ($\log n$ is an upper bound for the size of our set cover because our set cover is always the ceiling of $\log n$). This is true even for the minimum set cover of size 6, which meets both the conditions of a minimum set cover of always size 3 and size $\Omega(\log n)$.

To generalize this family of set cover problems to any minimum set cover size k where $k \geq 3$, consider the set $X = \{2k, 4k, \ldots, k \cdot 2^p\}$ for any $p > 0$. For each set of size $n = k \cdot 2^p$, we can generate a set S of $\lceil \log n \rceil + k$ subsets:

$\lceil \log n \rceil$ disjoint subsets of size $\frac{n}{2}, \frac{n}{4}, \ldots, \frac{n}{2^p}, \lceil \frac{n}{2^{p+1}} \rceil, \lceil \frac{n}{2^{p+2}} \rceil$ covering all elements of the set.

k disjoint subsets of size $\frac{n}{k}$ covering all elements of the set.

Following the same reasoning as stated above, the minimum set cover would always be k. The greedy algorithm will always choose to include sets of size $\frac{n}{2^k}$ to include in the set cover over including sets of size $\frac{n}{k}$ because they would not contain more uncovered elements. Thus, to compose a set cover the greedy algorithm would choose all $\lceil \log n \rceil$ subsets of size $\frac{n}{2}, \frac{n}{4}, \ldots, \frac{n}{2^p}, \lceil \frac{n}{2^{p+1}} \rceil, \lceil \frac{n}{2^{p+2}} \rceil$, which satisfies the requirement of a set cover of size $\Omega(\log n)$. For the case of minimum n, $\Omega(\log n)$ and the minimum set cover size k are the same.

5. "Hey Upper East Siders, Gossip Girl here." You are secretly Gossip Girl, an anonymous gossip blogger who keeps track of friendships at Constance Billard High School. You publish

an up-to-date map of the friendships at Constance on your website.[2] You maintain this map by a stream of tips from anonymous followers of the form "A is now friends with B."

(a) **(5 points)** You call some groups of people a "squad": each person is in the same squad as all their friends, and every member of a squad has some chain of friendships to every other member. For example, if Dan is friends with Serena, Serena is friends with Blair, and Alice is friends with Donald, then Dan, Serena, and Blair are a squad (You make up the name "The Gossip Girl Fan Club") and Alice and Donald are another squad ("The Constance Constants"). Give an algorithm that takes in a stream of (a) tips and (b) requests for a specified person's squad name. You should answer requests that come in between tips consistently—if you make up the name "The Billard billiards players" for Dan's squad, and you're asked for Serena's squad's name before any new tips come in, you should report that it's "The Billard billiards players".

Solution:

The structure of squad groupings can be modeled with a Union-Find data structure. For each person, there will be a node initially points to themselves. Each node will initially store a random squad name. The squad name will be stored at the root of a Union-Find component.

Once a tip comes in, the algorithm will run the $UNION(x, y)$ operation on the two nodes, x and y, that the tip regards. Using unionization as discussed in lecture, the smaller of the two squads will join the bigger squad, taking their squad name and merging. If the two squads are equal in size, the new name is randomly chosen between the two.

To find which squad a certain person x is apart of, we will run the $FIND(x)$ method, which will track down the name of the squad by tracing the node back to the root of the squad, which has the squad name stored.

(b) **(10 points)** A "circular squad" is defined to be a squad such that there is some pair of friends within the group that have both a friendship and a chain of friendships of length more than 1. In the example above, if Dan and Blair also became friends, then the group would be a circular squad. If Dan and Donald also became friends, they would all be in one circular squad. Modify your algorithm from the previous part so that you report names that contain the word "circle" for all circular squads (and not for any other squads).

(c) **(15 points)** Chatter Charlie, a rival gossip blogger, also begins to publish friendship maps informed by a stream of anonymous tips. Some tips are sent to just you, some just to him, and some to both of you. After you're both done taking tips for the year, it turns out that the whole school is a squad, and both you and Charlie know it. Give an

[2] There are no ethical concerns here, because you're a character in a highly-rated teen drama.

algorithm to find the smallest subset of the tips that would've sufficed to convince both you and Charlie that the whole school was a squad.

6. On July 21st, 2019, our favorite TF, Tarun, participated in a 6-hour long Pokemon Go marathon. It was Mudkip Community Day, where Tarun caught as many mudkips as possible in 6 hours.

   Unfortunately, Tarun's backpack has limited carrying capacity. He has M extra mudkips that he would like to give to his friends. . . for a price. His trainer friends, including $t_0 =$ Eric and $t_1 =$ Phyllis, are indicated by $T = \{t_0, \ldots, t_{|T|-1}\}$. Each trainer t_i would pay Tarun some positive integer p_i dollars for some positive integer m_i mudkips. The offers are all-or-nothing: each trainer t_i will walk away with either 0 or m_i mudkips. Let a "profile" be a set of values of M, $|T|$, p_i, and m_i as above.

   For example, one profile has $t_0 =$ Eric, $t_1 =$ Phyllis, and $t_2 =$ Richard, $m_0 = 5$, $p_0 = \$900$, $m_1 = 7$, $p_1 = \$1000$, $m_2 = 1$, $p_2 = \$1$, and $M = 10$. Given that profile, the best Tarun can do is to give 7 mudkips to Phyllis for \$1000 and one mudkip to Richard for \$1.

   Consider the following two algorithms:

   - The "value-per-mudkip" greedy algorithm where Tarun selects, among the trainers whom he has enough mudkips to satisfy, the person offering the highest value per mudkip, sells them their requested mudkips, and repeats.
   - The "lazy" greedy algorithm where Tarun decides that dividing is too difficult and selects, among the trainers whom he has enough mudkips to satisfy, the person offering the highest total price, sells them their requested mudkips, and repeats.

   (a) **(5 points)** Consider the profile where $M = 10, |T| = 3, (m_0, m_1, m_2) = (10, 5, 1), (p_0, p_1, p_2) = \{8, 5, 1\}$. What does the "value-per-mudkip" greedy algorithm return? What does the "lazy" greedy algorithm return? What is the optimal allocation?

   Solution:

   Following the "value-per-mudkip" greedy algorithm, Tarun will first look at the "value-per-mudkip" for each trainer:

$$t_0 : \frac{p_0}{m_0} = \frac{8}{10} = 0.8$$
$$t_1 : \frac{p_1}{m_1} = \frac{5}{5} = 1$$
$$t_2 : \frac{p_2}{m_2} = \frac{1}{1} = 1$$

   First, Tarun would select t_1 to sell to as they have the highest "value-per-mudkip" sale. Next, he will select t_2 as they have the highest "value-per-mudkip" sale of the remaining people that he has enough mudkips for. After this he does not have enough mudkips to make any other sales, so he is done. Upon completion he has made \$6 and sold 6 mudkips.

Following the "lazy" greedy algorithm, Tarun will look at t_0, t_1, and t_2 to see how much money each trainer offers. He will first select t_0 to sell to as they offer the highest total price. After this he does not have enough mudkips to make any other sales, so he is done. Upon completion, he has made \$8 and sold 10 mudkips.

The optimal allocation is the "lazy" greedy algorithm because Tarun makes the most money (\$8) even though he doesn't get the highest value per mudkip.

(b) **(5 points)** Let the optimal allocation result in profit P_{opt}. Let the "value-per-mudkip" greedy algorithm result in profit P_{vpm}. Prove the smallest real number upper bound you can on $\dfrac{P_{\text{opt}}}{P_{\text{vpm}}}$, or prove that no such real number upper bound exists.

Solution:

To prove the above result, consider the following profile: $|T| = 2$, $M = n$, $\{m_0, m_1\} = \{n, 1\}$, and $\{p_0, p_1\} = \{n-1, 1\}$, where $n \geq 2$. First let us calculate P_{vpm}. To do this Tarun will first look at the "value-per-mudkip" for each trainer:

$$t_0 : \frac{p_0}{m_0} = \frac{n-1}{n}$$
$$t_1 : \frac{p_1}{m_1} = \frac{1}{1} = 1$$

Since $\frac{n-1}{n} < 1$, Tarun will first choose t_1 as they offer the highest value-per-mudkip, selling a single mudkip for \$1. After this he does not have enough mudkips to make any other sales, so he is done. Upon completion, he has made \$1 and sold 1 mudkip. As such, for any value of n, $P_{\text{vpm}} = 1$. To calculate P_{opt}, Tarun should obviously choose t_0 as they offer the highest profit, selling n mudkips for $\$n-1$. In this case, even though he can't make the highest vpm trade, he has still guaranteed the highest profit. From here, we may observe that $\frac{P_{\text{opt}}}{P_{\text{vpm}}} = \frac{n-1}{1} = n-1$. Since we only specified that $n \geq 2$, n may be an infinitely large integer greater than or equal to 2, so $n-1 \to \infty$ and has no real number upper bound. Thus, using this profile, we have proven there exists no real number upper bound for $\frac{P_{\text{opt}}}{P_{\text{vpm}}}$.

(c) **(5 points)** The "value-per-mudkip" algorithm skips the next highest VPM trainer when they want more mudkips than Tarun has left. Suppose Tarun cheats a bit and sells to this highest trainer at full price even though he cannot allocate them all the mudkips they request—possibly even 0 mudkips. (Then Tarun flees town.) That is, Tarun sells to the first k highest VPM trainers until he has either allocated more than M mudkips or sold to all trainers. Prove the smallest real number upper bound you can on the ratio of P_{opt} to the profit from this "illegitimate" auction, or prove that no such real number upper bound exists.

Solution:

The smallest real number upper bound you can on the ratio of P_{opt} to the profit from this illegitimate auction P_{ill} is 1. We proceed with our proof.

Let total m be the number of mudkips Turan is being asked to sell by all trainers. First, let us consider the case in which $m \leq M$. In this case, Tarun's optimal allocation of mudkips to maximize profit is to simply sell to all trainers. So, P_{opt} is equal to the sum of all the money that the trainers are offering. For this same case, let us consider the illegitimate approach. Since the the illegitimate auction runs the VPM algorithm until Tarun has allocated more than M mudkips or sold to all trainers, Tarun will not need to engage in illegal actions because he finishes selling to all trainers first since $m \leq M$. As such, P_{ill} is equal to the sum of all the money that the trainers are offering, meaning $P_{\text{ill}} = P_{\text{opt}}$. Thus, for this case $\frac{P_{\text{opt}}}{P_{\text{ill}}} = 1$.

Next let's consider the case where $m > M$. In this case, the illegitimate auction will sell to the highest VPM trainers until the number of mudkips allocated to trainers exceeds M. Let VPM_{ill} be the average VPM of all sales of the illegitimate auction and let VPM_{opt} be the average VPM of all sales of the optimal allocation. Since the illegitimate auction follows the VPM algorithm till the number of mudkips allocated to trainers exceeds M, we know that $VPM_{ill} \geq VPM_{opt}$. Let n_{ill} be the number of mudkips sold by the illegitimate auction and let n_{opt} be the number of mudkips sold by the optimal allocation. Then, $n_{ill} > n_{opt}$. Let us note that the average VPM of sales multiplied by the number of mudkips sold gives total profit. Then $VPM_{ill}\dot{n}ill = P_{\text{ill}}$ and $VPM_{opt}\dot{n}opt = P_{\text{opt}}$. Since $VPM_{ill} \geq VPM_{opt}$ and $n_{ill} > n_{opt}$, we know then that for the case $m > M$, $P_{\text{opt}} < P_{\text{ill}}$, so $\frac{P_{\text{opt}}}{P_{\text{ill}}} < 1$.

As such, we have shown that for all $m > M$, $\frac{P_{\text{opt}}}{P_{\text{ill}}} < 1$ and for all $m \leq M$, $\frac{P_{\text{opt}}}{P_{\text{ill}}} = 1$, meaning $\frac{P_{\text{opt}}}{P_{\text{ill}}} \leq 1$. Thus, we have shown that the smallest real number upper bound you can on the ratio of P_{opt} to the profit from this illegitimate auction P_{ill} is 1.

(d) **(15 points)** Give a new algorithm such that $\dfrac{P_{\text{opt}}}{P_{\text{new}}}$ is always at most 2.

(**Hint:** Sorting by VPM, which trainers does the "value-per-mudkip" greedy algorithm select? How does the value of the trainer Tarun cheats relate to the "lazy" greedy algorithm?)

Solution:

Consider the following algorithm for computing P_{new}:

1. For any profile M, $|T|$, p_i, and m_i, first disregard the trainers who are asking for more than M mudkips as Tarun cannot engage in these trades legally.

2.Next, find the highest price offered out of all trades by running the lazy algorithm once over the altered profile. Let this value be P_{highest}.

3. Find P_{vpm} of the altered profile by calculating the VPM of each trade, sorting trades by VPM, and choosing trades with the highest VPM till no more valid trades are available.

4. Let P_{new} be $\max\{P_{\text{highest}}, P_{\text{vpm}}\}$.

Proof of Correctness:

We seek to prove that $\frac{P_{\text{opt}}}{P_{\text{new}}} \leq 2$. This is the same as proving $P_{\text{opt}} \leq 2P_{\text{new}}$. From part (c), we have that $\frac{P_{\text{opt}}}{P_{\text{ill}}} \leq 1$, effectively, $P_{\text{opt}} \leq P_{\text{ill}}$. So, by showing that $P_{\text{ill}} \leq 2P_{\text{new}}$, we have $\frac{P_{\text{opt}}}{P_{\text{new}}} \leq 2$.

When broken down, the illegitimate auction returns the amount of profit generated by the VPM algorithm P_{vpm} generated by the VPM plus the amount of profit P_{next} generated by the trainer with the next-highest VPM. As such, we have $P_{\text{ill}} = P_{\text{vpm}} + P_{\text{next}}$. As such, we know that $P_{\text{ill}} \leq P_{\text{vpm}} + P_{\text{highest}}$, since $P_{\text{highest}} \geq P_{\text{next}}$ since it is the highest profit. Since P_{new} is defined as the maximum of P_{vpm} and P_{highest}, we have that $P_{\text{vpm}} + P_{\text{highest}} \leq 2P_{\text{new}}$. Accordingly, we have:

$$P_{\text{ill}} = P_{\text{vpm}} + P_{\text{next}} \leq P_{\text{vpm}} + P_{\text{highest}} \leq 2P_{\text{new}}.$$

So, $P_{\text{ill}} \leq 2P_{\text{new}}$. As such, by our reasoning as stated above, we have that $\frac{P_{\text{opt}}}{P_{\text{new}}} \leq 2$. Thus, our algorithm is correct.

Runtime:

Disregarding trainers who request more than M mudkips takes up at most $O(|T|)$ time. One run of the lazy algorithm to find P_{high} takes $O(|T|)$ time as does finding P_{vpm} by calculating VPMs of all trades. Sorting trades by VPM takes $(O(|T| \log |T|)$ time using Mergesort, and choosing valid trades with highest possible VPMs ($O(|T|)$ time). Finding the max of P_{highest} and P_{vpm} takes $O(1)$ time. Thus, the runtime of our algorithm is $O(|T| \log |T|)$.

(e) **(10 points)** Using your new algorithm from the previous part, show that for every $\epsilon > 0$, there exists some profile such that $\dfrac{P_{\text{opt}}}{P_{\text{new}}} \geq 2 - \epsilon$. This proves that your bound from the previous part is tight—you cannot find a better upper bound.

Solution:

We seek to prove that for every $\epsilon > 0$, there exists a profile such that $\frac{P_{\text{opt}}}{P_{\text{new}}} \geq 2 - \epsilon$. We may accomplish this by showing $\lim_{n\to\infty} \frac{P_{\text{opt}}}{P_{\text{new}}} = 2$.

Consider the profile $M = 2n$, $|T| = n+1$, $\{m_0, m_1, \ldots, m_n\} = (n+1, 1, 1, \ldots, 1)$, $\{p_0, p_1, p_2, \ldots, p_n\} = \{n, 1, 1, \ldots, 1\}$ for some integer n. When running our algorithm for finding P_{new}, we will always get $P_{\text{new}} = n$ since $P_{\text{vpm}} = n$ and $P_{\text{highest}} = n$. However, $P_{\text{opt}} = 2n-1$ or the optimal allocation is for Tarun to maximize his profit is to trade with t_0 through t_1, and selling all of his mudkips to maximize the profit.
Now, consider $\frac{P_{\text{opt}}}{P_{\text{new}}}$. From above, we have that this ratio is $\frac{2n-1}{n} = 2 - \frac{1}{n}$. Taking the limit of $\frac{P_{\text{opt}}}{P_{\text{new}}}$ as $n \to \infty$ we then have:

$$\lim_{n\to\infty} \frac{P_{\text{opt}}}{P_{\text{new}}} = \lim_{n\to\infty} 2 - \frac{1}{n} = 2.$$

As such, we've shown $\lim_{n\to\infty} \frac{P_{\text{opt}}}{P_{\text{new}}} = 2$. Thus, for every $\epsilon > 0$, there exists a profile such that $\frac{P_{\text{opt}}}{P_{\text{new}}} \geq 2 - \epsilon$.