CS 124 Homework 3: Spring 2024

Collaborators: Sophie Zhu, Collin Fan, Rowan Wang, Alex Gong, Kathryn Harper, Jude Partovi, Charlie Chen

No. of late days used on previous psets: 2
No. of late days used after including this pset: 2

Homework is due Wednesday Feb 28 at 11:59pm ET. Please remember to select pages when you submit on gradescope. Each problem with incorrectly selected pages will lose 5 points.

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 your papers to turn them in.

Collaboration Policy: You may collaborate on this (and all problem sets) only with other students currently enrolled in the class, and of course you may talk to the Teaching Staff or use Ed. You may also consult the recommended books for the course and course notes linked from the timetable. You may not use generative AI or large language models, or search the web for solutions, or post the questions on chat forums. Furthermore, you must follow the "one-hour rule" on collaboration. You may not write anything that you will submit within one hour of collaborating with other students or using notes from such sources. That is, whatever you submit must first have been in your head alone, or notes produced by you alone, for an hour. Subject to that, you can collaborate with other students, e.g. in brainstorming and thinking through approaches to problem-solving.

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 no credit. Again, try to make your answers clear and concise.

Problems

1. **Detecting whether an edge lies in an MST (10 points)**

   (a) **(5 points)** Let G be a weighted graph in which all edge weights are distinct. Prove that an edge e of a graph G belongs in some MST of G if and only if the following property holds: for every cycle in G that contains e, the edge with the highest weight is not e.

   Proof. We begin by proving that if an edge e of a graph G belongs in some MST of G, then for every cycle in G that contains e, the edge with the highest weight is not e. Let us assume, for the sake of contradiction, that edge e of a graph G belongs in some MST of G and there exists at least one cycle in G that contains e in which e has the highest weight. Thus, let us consider a cycle with vertices $V = \{v_1, v_2, ..., v_n\}$ and edges $E = \{e_1, e_2, ..., e_n\}$. Let e_1 be the highest weight edge in E such that $e_1 > e_i, \forall 1 < i \leq n$.

Since the MST of G by definition must span all vertices $v_i \in V$, it follows that our MST contains some sequence of $n-1$ edges of E such that the sum of the $n-1$ chosen edges is minimized. This means we need at minimum $n-1$ edges to connect all n vertices and the MST must use at most $n-1$ vertices if it is minimum. Thus, if we assume that e_1 is in some MST of G, then the MST uses some sequence of edges including e_1 but excluding one edge $e_i \in E$. However, if we consider the sum of edge weights when we include e_i and exclude e_1, then we can see that $e_1 + \sum(E \setminus \{e_i\}) > e_i + \sum(E \setminus \{e_1\})$ since e_1 is by definition greater than e_i. Therefore, there exists a sequence of $n-1$ edges excluding the highest weight edge e_1 which has a smaller sum of edge weights than a sequence of $n-1$ including largest edge e_1. Thus, the assumption that e_1 is in some MST of G implies that the MST is not minimal, which is a contradiction.

Now, we will prove the other direction: if every cycle in G which contains e has the property that the edge with the highest weight is not e, then e belongs in some MST of G. Assume, for the sake of contradiction, that every cycle in G which contains e has the property that the edge with the highest weight is not e but e does not belong in some MST of G. Suppose we run Kruskal's algorithm on G. By the correctness of Kruskals, we know that it will always output a correct MST. Furthermore, by definition, Kruskal's orders the edges by edge weight and adds an edge to the MST if and only if the edge does not create a cycle in our MST. Since we assumed that every cycle in G which contains e has the property that the edge with the highest weight is not e, it follows that there exists some heavier edge in all the cycles containing e. Thus, this edge would be sorted after e in order of consideration in Kruskals. Thus, if e does not belong in some MST of G, then we know by Kruskal's that adding e would create a cycle in our MST. However, this implies that the heavier edge in a cycle containing e was already considered and added. Thus, we arrive at a contradiction since Kruskal's considers the edges in their sorted ascending order. Therefore, if every cycle in G which contains e has the property that the edge with the highest weight is not e, then e belongs in some MST of G.

Since we have proved that both statements imply the other, we have proved that an edge e of a graph G belongs in some MST of G if and only if the following property holds: for every cycle in G that contains e, the edge with the highest weight is not e. □

(b) **(5 points)** Given an edge e in G, give an algorithm that outputs YES if there exists an MST of G that contains e, and NO otherwise. Your algorithm must have runtime asymptotically faster than the algorithms given in class for finding MSTs. You must describe your algorithm, prove its correctness and state its run time. You need not prove the runtime. (You may use the result from Part (a) even if you did not prove it.)

Algorithm: We will define our algorithm that takes in a weighted graph G and finds if there exists an MST of G that contains edge $e = (u, v)$ as such:

Construct a modified graph G' by removing all edges in G which have a weight greater than or equal to e. Next, run DFS starting from u on G'. If DFS finds a path between u and v in G', return NO. Otherwise, return YES.

Correctness: We will prove that our algorithm outputs NO if and only if there does not exist an MST of G that contains e.

Proof. Let $e = (u, v)$. We will use the invariant that every edge in G' has an edge weight that is less than the edge weight of e.

First, we will prove that if our algorithm outputs NO, there does not exist an MST of G that contains e. By the correctness of DFS, we know that if our algorithm returns YES, then there exists a path between u and v in G'. Furthermore, if there exists a path from u to v in G', then it follows in our definition of G' that there exists a path from u to v in G such that all edges in the path have a weight less than e. Therefore, by our invariant, we must have a cycle in G from u to v where e has the highest weight. Thus, using the iff cycle property of MST's we proved in (a), edge e is not included in any MST of G.

Next, we will prove that if there does not exist an MST of G that contains e, then our algorithm return NO. Since our edge weights are distinct and we remove all edges with weights greater than or equal to the edge weight of e, then it follows that all edges in G' have weights strictly lesser than e. Moreover, if there does not exist an MST of G that contains e, we know by the iff cycle property of MST's in (a) that for some cycle in G containing e, edge e has the highest edge weight in the cycle. Thus, it follows by the previous statements that there must exist a path from vertex u to vertex v in G' if there does not exist an MST of G that contains e. Since we run DFS on G' starting from u and return NO if DFS finds a path from u to v, we know by the correctness of DFS that our algorithm will return NO if there does not exist an MST of G that contains e.

Since we have proved both statements imply one another, we have proved that our algorithm outputs NO if and only if there does not exist an MST of G that contains e. □

Runtime: Let $m = |E|$ and let $n = |V|$. Thus, our runtime for the algorithm is $O(n+m)$.

2. **Maximal independent set in an evolving graph (30 points):**

   Given an undirected graph $G = (V, E)$, we say that a subset $S \subset V$ is an *independent set* if no two vertices in S are connected by an edge. We say that S is a *maximal independent set (MIS)* if S is an independent set and furthermore there is no strict superset T (i.e., a set T with $S \subsetneq T$) which is also an independent set. In this problem we will explore the running time of algorithms computing and maintaining maximal independent sets in graphs with n vertices, m edges and with maximum degree Δ (i.e., every vertex $u \in V$ has at most Δ edges touching it.)

   (a) **(5 points)** Give an algorithm that finds a maximal independent set of G, given G in the adja-

cency list representation. (You must describe your algorithm fully and give a brief explanation of why it is correct. You should state your runtime but you don't need to prove it.)

Algorithm: We will define our algorithm that takes in a graph G and outputs a maximal independent set as such: Initialize an array S indexed by V with $|V|$ entries of 1. Next, iterate through the vertices of G starting at an arbitrary vertex $v_0 \in V$. If $S[v_i] = 1$, then iterate through the edges $(v_i, v_j) \in E$ and set $S[v_j] = 0$. Otherwise, if $S[v_i] = 0$, move on to the next vertex in the iteration. Once all the vertices have been iterated through, return S. If $u \in S$, then $S[u] = 1$. Otherwise, if $u \notin S$, then $S[u] = 0$.

Correctness: We will prove that our algorithm finds a maximal independent set of G.

Proof. To find a maximal independent set of G, we must find a set of vertices S such that no two vertices in S are connected by an edge, and no $v \notin S$ can be added to S without S no longer being an independent set.

We will first prove that our algorithm returns a set of vertices S such that no two vertices in S are connected by an edge. We defined our algorithm such that for all $(v_i, v_j) \in E$ where $S[v_i] = 1$ (meaning v_i is in our MIS), $S[v_j] = 0$. Furthermore, since we only add a vertex v in S if $S[v] = 1$, it follows that our algorithm never adds a vertex that has an edge to a vertex already in the MIS. To illustrate this, consider the first step in our algorithm: when we start with v_0, we add v_0 to our set S and set all $S[v_j] = 0$ where $(v_i, v_j) \in E$. Next, we add the next vertex v_k such that $S[v_k] = 1$. We again set all $S[v_l] = 0$ where $(v_k, v_l) \in E$. Through this process, we can see that we only consider/add vertices which have no edges to vertices in S already, maintaining the independence of S.

Next, we will prove that our algorithm returns a set S such that no $v \notin S$ can be added to S without S no longer being an independent set. Consider, for the sake of contradiction, that there exists a vertex $v_i \notin S$ which can be added to S with S remaining an independent set. By the definition of S, this consider two cases: either $S[v_i] = 1$ or $S[v_i] = 0$. If $S[v_i] = 1$, then v_i would satisfy the if condition in our algorithm and v_i would have already been added to S. If $S[v_i] = 0$, then by the definition of our algorithm, there must exist some $(v, v_i) \in E$ such that $v \in S$. Thus, adding v_i would cause S to no longer be an independent set. since both cases lead to a contradiction, we have proved that our algorithm returns a set S such that no $v \notin S$ can be added to S without S no longer being an independent set. □

Runtime: Let $n = |V|$ and $m = |E|$. The runtime of our algorithm is $O(n + m)$ because we iterate through all the vertices and edges in G.

Now suppose the graph G is changing over time, and we want to maintain a maximal independent set of this graph without having to recompute it from scratch every time G is updated. Concretely, suppose that in each time step, some edge is either added to or deleted from G. Our basic data structure simply maintains a set $S \subseteq V$ in the form of an array indexed by V such that $S[u] = 1$ if

$u \in S$ and 0 otherwise.

(b) **(0 points, not to be turned in)** Prove that the adjacency lists can be maintained with $O(\Delta)$ cost per insertion and deletion. (You may assume you have a solution to this problem in future parts even if you did not solve it.)

(c) **(7 points)** Describe algorithms INSERT(e) and DELETE(e) to maintain S under edge insertions and deletions respectively. Give upper bounds on the runtime of both operations. (Your algorithm and its claimed run times must be correct, but you need not prove these. Note that for full credit your runtimes should depend only on Δ and not on n.)

Algorithm: Let $e = (i, j)$.

We will define algorithm INSERT(e) as such: If $S[i] = 1$ and $S[i] = 1$, for all edges $(i,k) \in E$ check if there exists an edge $(k,d) \in E$ such that $S[d] = 1$. If (k,d) does not exist, set $S[k] = 1$. Finally, remove i from S. In all other cases, do nothing.

We will define algorithm DELETE(e) as such: If $j \in S$ and $i \notin S$, check if there exists an edge $(i,d) \in E$ such that $S[d] = 1$. If not, set $S[i] = 1$. By similar logic, if $i \in S$ and $j \notin S$, check if there exists an edge $(j,d) \in E$ such that $S[d] = 1$. If not, set $S[j] = 1$. In all other cases, do nothing.

Runtime: For INSERT(e), we have a worst case runtime where for all edges $(i,k) \in E$, we must check all edges $(k,d) \in E$. Thus, if the maximum degree is Δ, then we must check Δ edges from i and for each of those edges we check another Δ edges. Therefore, INSERT(e) has a upper bound runtime of $O(\Delta^2)$.

For DELETE(e), we must check all the edges $(i,d) \in E$. Thus, if the maximum degree is Δ, then we check at worst Δ edges from i. Therefore, DELETE(e) has a upper bound runtime of $O(\Delta)$.

(d) **(3 points)** For every integer $\Delta > 0$ describe an example (i.e., a graph, an MIS, and an edge to be inserted) such that the number of queries to S for inserting an edge asymptotically match your upper bound from Part 2c.

Consider a graph defined as the union between a singular vertex with no edges and a tree of height one. This tree has 1 root r and r has Δ children. Furthermore, let each of the children c_i of r is connected to all $\Delta - 1$ other children in the tree.

We define our MIS such that it contains just the root of the tree r and the one unconnected vertex v. Let us try to insert edge (v,r), where we choose r to remove from the MIS. Thus, by our definition of INSERT(e), for all Δ children of r, we must check each of the $\Delta - 1$ edges to the other children in the tree. In addition, since none of the children were originally in the MIS (since they were connected to r in the MIS), we must iterate through *all* $\Delta - 1$ edges for each of the Δ children of r to verify that the

child is no longer connected by an edge to the MIS. Hence, we must perform $\Delta(\Delta-1)$ operations which is $\Theta(\Delta^2)$ queries to insert the edge (v, r_i). Note that this example holds for every $\Delta > 0$ since $\Delta > 0$ guarantees the existence of the tree (and not just a singular node r).

To speed up the runtimes from Part 2c, suppose we decide to additionally maintain an array A indexed by V, such that for every $u \in V$, $A[u]$ counts the number of neighbors of u that are in S. (So $A[u] = |\{v \in V | v \in S, (u,v) \in E\}|$.)

(e) **(5 points)** Give algorithms A-INSERT and A-DELETE that maintains both S and A under edge insertions and deletions. (While any correct polynomial-time algorithm will get you full points, needlessly inefficient algorithms will lose points in the next part!)

Algorithm: Let $e = (i, j)$.

We will define algorithm A-INSERT(e) as such: If $S[j] = 1$ and $S[i] = 0$, increment $A[i]$ by 1. By similar logic, if $S[i] = 1$ and $S[j] = 0$, increment $A[j]$ by 1. If $S[i] = 1$ and $S[j] = 1$, for all edges $(i,k) \in E$, decrement $A[k]$ by 1. If this leaves $A[k] = 0$, then for all edges $(k,l) \in E$, increment $A[l]$ by 1. Then, set $S[k] = 1$. Finally, set $S[i] = 0$. In all other cases, do nothing.

We will define algorithm A-DELETE(e) as such: If $S[j] = 1$ and $S[i] = 0$, decrement $A[i]$ by 1. If this decrement leaves $A[i] = 0$, for all edges $(i,k) \in E$, increment $A[k]$ by 1. By similar logic, if $S[i] = 1$ and $S[j] = 0$, decrement $A[j]$ by 1. If this decrement leaves $A[j] = 0$, for all edges $(j,k) \in E$, increment $A[k]$ by 1. In all other cases, do nothing.

Runtime: We assume that indexing and changing values in both A and S takes constant $O(1)$ time. Thus, our worst case scenario is as such: $i \in S$ and $j \in S$ with the degree of i being Δ. For each neighbor c of i, there are Δ neighbors of c and $A[c] = 1$ (that is, the only edge connecting c to a vertex in S is (i,c)) such that when our algorithm decrements $A[c]$ by 1, we must also update the Δ neighbors of c. Therefore, for each Δ children of i, we must perform Δ more updates so our worst case runtime for A-INSERT(e) is $O(\Delta^2)$.

For A-DELETE(e), we operate under the same assumption that that indexing and changing values in both A and S takes constant $O(1)$ time. Thus, our worst case scenario is as such: without loss of generality, we have $i \in S$, $j \notin S$, j has a degree of Δ, and $A[j] = 1$ (that is, the only edge connecting j to a vertex in S is (i,j)). Then, when we decrement $A[j]$ by 1, we have $A[j] = 0$ and we must also increment $A[c_i]$ by 1 for every neighbor $c_i, i \in \{1, \ldots, \Delta\}$ of j. Therefore, our worst case runtime for DELETE(e) is $O(\Delta)$.

(f) **(10 points)** Assume that initially the graph G is empty (no edges), and S consists of all vertices in V. Give an amortized analysis proof that after T updates to G, the total runtime of the

operations is at most $O(\Delta \cdot T)$. (Hint: Consider a charging scheme that charges the runtime of adding a vertex to S to the vertex itself. You should be careful to pay this charge when the same vertex is deleted from S!)

Proof. Per the Hint, we will begin by considering the runtime of adding a vertex to our MIS and deleting a vertex from our MIS (note that this is not the same as adding a edge or deleting an edge from G). We define the function ADD(v) and REMOVE(v), which maintain the previously defined structures A and S, as such:

ADD(v): Set $S[v] = 1$. Then, for all $(v, u) \in E$, increment $A[v]$ by 1. This takes $O(\Delta)$ runtime.
DELETE(v): Set $S[v] = 0$. Then, for all $(v, u) \in E$, check if $A[u] = 1$, in which case we decrement $A[u]$ by 1 and increment $A[v]$ by 1 . If $A[v] = 1$, call ADD(v). We reiterate that DELETE(v) simply maintains A and S after removing a vertex from the MIS. We do not perform the additional $O(\Delta)$ loops for each u since this is runtime is covered by the processes in ADD(v). Thus, DELETE(v) has a runtime of $O(\Delta)$.

Next, let us express our previously defined functions A-INSERT(e) and A-DELETE(e) in terms of how many vertices they add or delete to the MIS, which we can express in terms of ADD(v) and DELETE(v). Using our definitions in part (e), for A-INSERT(e), we see that we only make changes to the MIS if the edge we add is between two vertices already in the MIS. In this case, A-INSERT(e) then arbitrarily deletes one of the vertex endpoints of the edge and adds at most Δ vertices to the MIS if the vertex we deleted had Δ neighbors which only had one edge connecting them to the MIS. Thus, for any A-INSERT(e) call, we have at most 1 DELETE(v) and Δ ADD(v) calls. For A-DELETE(e), we see that we only make changes to the MIS if we delete an edge which is the only edge connecting a vertex to the MIS. In this case, we add the vertex to the MIS. Thus, for any A-DELETE(e) call, we make at most 1 ADD(v) call.

Thus, in our T operations, let us make j A-INSERT calls and k A-DELETE calls. Furthermore, let us make m ADD calls and n DELETE calls within these T operations. Since we begin with all vertices of G already in the MIS, then we know that we must delete at least as many vertices from our MIS as we add them (each add vertex call must be paired with a delete vertex call). Thus, $n \geq m$. In addition, since we at most make one DELETE call per A-INSERT call, we have $n \leq j$. Therefore, since $T = j + k$, we have $n \leq T - k$. It follows that $m \leq n \leq T - k$. Then, our runtime can be expressed as $O(a\Delta + b\Delta)$ since ADD(v) and DELETE(v) covers all possible operations and they each have a runtime of $O(\Delta)$. Then, using the inequalities above, we have the runtime of T operations is $O(a\Delta + b\Delta) = O(2b\Delta) = O(2(T - k)\Delta) = O(T\Delta)$. Therefore, we have proved that the total runtime of T operations is at most $O(\Delta \cdot T)$. □

3. **Sorta sorting with heaps (24 points)** 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.

Algorithm: We define our algorithm which takes in k sorted lists with n total elements and merges them into one sorted list as such:

Initialize a min heap H and a list S. For each of the k lists, insert(H, list_k, $\text{list}_k[0]$), which inserts the k-th list into the heap with the value of its first element. While H is not empty, deletemin(H), which returns one of the k lists. Next, remove the element at index 0 from the list and append it to the end of S. Then, if the list still has elements, insert the list back into the heap as an object with the new value of list[0]. Finally, return S.

Correctness: We will prove that after the i-th step in our algorithm (ie. the i-th deletemin of our algorithm), the first i elements in S are sorted (this implies that after the n-th deletemin, or step, all n total elements are sorted in S).

Proof. We will use the invariant that each of the k lists is already sorted in ascending order.

Our base case is when $i = 1$. Since we only call deletemin(H) once and add the resulting element to the sorted list, S only has 1 element which is by definition sorted. Thus, our base case is proved correct. Note that this element is the globally smallest element. This is because smallest element (by correctness of deletemin) out of the first elements in every list (which are the smallest elements in their lists).

In our inductive step, we assume the inductive hypothesis that after the first $i-th$ deletemin, S contains i sorted elements. Thus, we wish to show that given the inductive hypothesis, then S will contain $i+1$ sorted elements after the $i+1-th$ deletemin.

Let j be the element added to S by the $i+1-th$ deletemin and k be the element added to S by the $i-th$ deletemin. Let us assume, for the sake of contradiction, that adding j to S will result in S being not sorted. Then, by our assumption that the i elements of S are sorted, it follows j must be less than at least the element k (since k is the largest of the sorted elements).

We will consider the two possible cases for the $i-th$ and $i+1-th$ deletemin: when k was the element which came before j in its list, and when k was the element at index 0 in another list while j was the element at index 0 in its own list. Note that these are the only two possible cases since our the value of each element in the heap is the element at index 0. For the first case, our assumption that $j < k$ is a contradiction since j comes after k in the same list and our invariant dictates that all lists are sorted. For the second case, if $j < k$ while j and k are at index 0 of their respective lists, then our $i-th$ deletemin (by its correctness from lecture notes) would have added j to S instead of k, a contradiction. Thus, since both cases lead to a contradiction, we have

proved that j must be greater than the greatest element k of the i elements in S. It then follows that after our algorithm adds j to S after the $i+1-th$ deletemin, the $i+1$ elements in S remain sorted.

By induction, we conclude that if S is sorted with i elements after the i-th deletemin, then our algorithm maintains the sorted order of S after the $i+1$-th deletemin. This implies that after the final, or n-th deletemin, all n elements from the k sorted lists will be correctly sorted in S. □

Runtime: We operate under the assumption that removing from the beginning and inserting into the end of any list is O(1).

Constructing the min heap requires $O(k \log k)$ runtime since we must insert all k lists into the heap. In addition, since we continually remove an element from its list until all lists are empty (which means our heap is empty), it follows that our algorithm performs n deletemins. Furthermore, since we also insert each list back into the heap with the exception of when each list has only 1 element left, we do $n-k$ insertions back into the heap. Thus, our algorithm has runtime of $O(n \log k + (n-k) \log k + k \log k)$ which is $O(n \log k)$.

(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.

Algorithm: We define our algorithm which takes in a list of n numbers that is k-close to sorted and outputs the sorted list:

Initialize a min heap H and a list S. Remove the first k elements from the inputted k-close list, inserting them into H (calling insert). Then, while H is not empty, deletemin(H) and append the resulting element to the end of S. If there is still elements in the inputted k-close list, remove element at index 0 from the k-close list and insert it into the heap. Finally, return S.

Correctness: We will prove that after each step i of our algorithm (ie. after the $i-th$ deletemin call), our algorithm appends the $i-th$ smallest element to S (this implies that after the $n-th$ deletemin, or step, all n elements are sorted correctly in S).

Proof. We will use the invariant that each element in the unsorted list is at most k positions away from its correct place in the sorted order of the list. Furthermore, we will prove the correctness with strong induction.

Our base case is when $i = 1$, or the first step of our algorithm. Thus, by our al-

gorithm construction, our heap is populated with the first k elements of the k-close list. Since we know that any element in the k-close list is less than k positions from its actual place in the sorted order, it follows that the element belonging to the first sorted position is appended to the heap. That is, the global minimum element is within the first k elements of the k-close list in the heap. Thus, the deletemin() in the first step must return the global minimum element, which our algorithm then appends to S. Thus, after the 1st deletemin, or step, the smallest element is added to S: our base case is proved true.

In our inductive step, we assume the following inductive hypothesis: after all steps $k = 1, ..., i$ in our algorithm, the $k-th$ smallest element is appended to S. We wish to show that given the inductive hypothesis, the $i+1-th$ smallest element is appended to S after the $i+1-th$ step of our algorithm. First, consider the "frame" of the k-sorted array which our heap has covered after step i. After step 1, we have the first $k+1$ elements in the heap. Moreover, by our inductive hypothesis, we also removed the first smallest element in the array from the heap. For every subsequent step i, by the definition of our algorithm, we remove the minimum element from the heap and insert the next element to be covered into our heap (until there are no further elements left). Therefore, after i steps, we have inserted the next i elements into our heap, meaning our heap has "covered" the first $k+i$ elements (note that $k+i$ is upperbounded by the length of the input list). Furthermore, we know by our inductive hypothesis that after the step i, the i smallest elements of the unsorted list have been removed from the k-close list and appended to S. Therefore, it follows that the $i+1-th$ smallest element will be the new global minimum of the k-close list (less than all remaining elements left in the list). In addition, by our invariant, we are guaranteed that the $i+1-th$ smallest element will be located within the first $k+i$ elements of the k-close list. Thus, by the preceding statements, we know that the deletemin call in the $i+1-th$ step will return the $i+1-th$ smallest element of the list and append it to the end of S.

By strong induction, we conclude that if after all steps $k = 1, ..., i$ in our algorithm, the $k-th$ smallest element is appended to S, then the $i+1-th$ smallest element is appended to S after the $i+1-th$ step. This implies that after the $n-th$ step in our algorithm, all n elements are sorted correctly in S. □

Runtime:

Proof. Let n be the number of elements in the inputted k-close list. Constructing the min heap takes runtime $O(k \log k)$ since we must insert the first k elements of the unsorted list into the heap. In addition, since our algorithm contains at most k elements at any given moment (at most since if there are no more elements left to cover in the unsorted array, then the heap will gradually deletemin until it is empty), then each insert or deletemin takes $O(\log k)$ time (by lecture notes 5). Furthermore, we only deletemin and insert one element per step of the algorithm, until all n elements of the unsorted array have been inserted and removed from the heap (that is, we stop

only when we process through all n elements of the unsorted list). Therefore, our algorithm takes n steps, each of which take $O(2\log k)$ time. Hence, our algorithm has a runtime of $O(2n\log k + k\log k)$. However, since k must be less than n by definition, then our runtime is $O(n\log k)$. □

4. **d-heaps (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.)

5. **Suboptimality of greedy algorithm for set cover (10 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 $k = 3$.)

   Consider the set $X = \{1,2,3,\ldots,2^b\}$ for some integer $b \geq 3$, and consider subsets of X of the form:

   $$M_1 = \{x \in X | x \equiv 1 \mod 3\}$$

   $$M_2 = \{x \in X | x \equiv 2 \mod 3\}$$

   $$M_3 = \{x \in X | x \equiv 0 \mod 3\}$$

   Furthermore, let us define subsets S_i for $i = 0,...,b$ such that each subset covers half of the remaining elements not yet covered by the previous subsets. That is, each subsequent subset $|S_i| = \frac{1}{2^{i+1}} * |Y|$, where we define Y as the current set of non-covered elements. We can concretely express S_i as below, but it is essential to understand the logic behind their construction as the notation becomes convoluted.

   $$S_0 = \{x \in X | 1 \leq x \leq \frac{n}{2}\}$$

   $$S_1 = \{x \in X | 1 + \frac{n}{2} \leq x \leq \frac{3n}{4}\}$$

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

$$S_2 = \{x \in X | 1 + \frac{3n}{4} \leq x \leq \frac{7n}{8}\}$$
$$\dots$$
$$S_b = \{x \in X | 1 + n(1 - 2^{-i}) \leq x \leq n(1 - 2^{-(i+1)}\}$$

Minimum cover: We will show that the minimum cover of X using the above subsets is $k = 3$. For $k = 1$, we can see from the definitions above that none of the subsets $S_i = X$ or $M_k = X$. In fact, for each M_k, we know that $M_1 \cup M_2 \cup M_3 = X$ since each subset M covers one of the 3 possible modulo congruences of $\mod 3$. Therefore, we know that $|M_1| \leq |X|/3$, $|M_3| \leq |X|/3$, and $|M_3| \leq |X|/3$ (since $|X|$ might not be cleanly divisible by 3. Moreover, by our definition of S_i, we are guaranteed that $|S_i| \leq |X|/2$ for any $i = 0, ..., b$. Therefore, there is no possible set cover of $k = 1$.

For $k = 2$, we will use similar arguments. Since $|S_1| \leq |X|/2$ the subsequent $|S_{i+1}|leq|S_i|/2$, it follows that the largest $|S_i| \leq |X|/2$ and the second largest $|S_{i_2}| \leq |S_i|/2$. Furthermore, since all $|M_k| \leq |X|/3$, we cannot have any two subsets which cover all of X.

For $k = 3$, as we have shown, $M_1 \cup M_2 \cup M_3 = X$ by definition of their construction. Therefore, there exists a cover of 3 subsets for X. Thus, the minimum cover of X is $k = 3$.

Greedy Algorithm Cover Size:

Proof. We will use strong induction to prove that after the $i - th$ step in our greedy algorithm (starting at the step 0), our greedy algorithm chooses S_i as the next subset in the cover. Again, we define Y to be the set of elements not yet covered by our algorithm.

Our base case is when $i = 0$. Thus, recall that $|S_0| = |X|/2 \geq M_k$ and $|S_0| = |X|/2 \geq S_i$ for $k = 1, 2, 3$ and $i = 1, ..., b$. Moreover, since we have not added any subsets to our cover yet, Y = X. Therefore, our greedy algorithm chooses S_0 as the first subset.

In our inductive step, we assume the inductive hypothesis that after every step 0 through step i, our greedy algorithm has selected S_0 through S_i to add to our set cover. Note that this is equivalent to our set cover being $\bigcup_{s=0}^{i} S_s$. We wish to show that given this inductive hypothesis, then our greedy algorithm will select S_{i+1} as its next subset for the set cover. Since our cover after step i is $\bigcup_{s=0}^{i} S_s$, it follows from their definitions that we have covered elements $W = \{1, ..., n(1 - 2^{-i-1}\} \subset X$, and $Y = X \setminus W = \{n(1 - 2^{-(i+1)} + 1, n\}$. Therefore, $|Y| = n - (n(1 - 2^{-(i+1)} + 1) + 1$ (inclusive subtraction) which yields $|Y| = n * 2^{-i-1}$. Then, let us define $P = |M_k \cap Y|$ and $Q = |S_{i+1} \cap Y|$. By the definition of our greedy algorithm, the next subset in our set cover will be $\max(P, Q)$. Thus, by an extension of the logic we have explained for the minimum set cover size, we have that $P = |M_k \cap Y| \leq \frac{n*2^{-i-1}}{3}$ and $Q = |S_{i+1} \cap Y| = \frac{n*2^{-i-1}}{2}$. Thus, since we can see that $Q = \frac{n*2^{-i-1}}{2} > \frac{n*2^{-i-1}}{3} \geq P$, then we see that our algorithm will choose Q, or equivalently, choose S_{i+1} as the next subset in the cover. Therefore, we have proven that assuming our greedy algorithm has selected S_0 through S_i to add to our set cover after every step 0 through step i, our greedy algorithm

will then add S_{i+1} after the step $i+1$.

Therefore, since we have shown the base case and inductive step, we conclude that after the $i-th$ step in our greedy algorithm (starting at the step 0), our greedy algorithm chooses S_i as the next subset in the cover. Since we have $i = 0, ..., b$, this means we have $\Omega(b)$ subsets in the set cover returned by the greedy algorithm (we do not concretely know that our set cover size is $b+1$ because |X| is not necessarily a power of 2 However, we have proved that there must be at *minimum*, $b+1$ subsets in our cover. Moreover, if the $b+1$ subsets of form S_i do not cover all of X, then our greedy algorithm will *then* move on to using the sets M_1, M_2, M_3, which by their definition will ensure our algorithm covers all of X. In such cases, the size of our set cover is still lower bounded by $b+1$). However, if we define $|X| = n$, it is clear that we have $2^b = n$, which means $b = \log_2 n$. Thus, we have shown that for the example above, the set cover returned by the greedy algorithm is of size $b = \Omega(\log n)$. □

Generalization: To generalize for different values of k other than $k = 3$, instead of creating 3 subsets M_1, M_2, M_3, we create k subsets $M_1 ... M_k$ defined as:

$$M_1 = \{x \in X | x \equiv 1 \mod k\}$$

$$M_2 = \{x \in X | x \equiv 2 \mod k\}$$

$$\dots$$

$$M_k = \{x \in X | x \equiv 0 \mod k\}$$

However, the creation of our other sets S_i remains the same. In this case, each $|M_k| \leq |X|/k$. Furthermore, since $Q = \frac{n*2^{-i-1}}{2} \geq \frac{n*2^{-i-1}}{k} \geq P$ for all integer values of $k \geq 2$, it follows from our induction proof that the greedy algorithm will still return a set cover of size $\Omega(\log n)$ with this generalized construction for other constant values of k.

6. **Tracking components in an evolving graph (15 points)** 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 distinct 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".

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

Algorithm: We will define our algorithm that takes in a stream of tips and requests for a specified person's squad name as such:

Initialize a Union Find Data Structure (with path compression and union by rank). Next, call Makeset(p_i) and store a unique squad name for every person p_i at Constance High.

For every tip received of form "p_1 is now friends with p_2", update the friendship map by calling Union(p_1, p_2) in our Union Find structure. For every request for the squad name of a person p_i, call Find(p_i) and return the squad name stored in the root parent node of the set containing p_i.

Correctness: We will prove that our algorithm maintains a correct mapping of the friendships at Constance and outputs the correct squad name for every request.

Proof. We will begin by proving that given a distinct tip of form "p_1 is now friends with p_2", our algorithm correctly updates the map of squads at Constance. In the first step of our algorithm, we initialized every person p_i at Constance as their own set in our Union Find structure. Furthermore, we call Union(p_1, p_2) if and only if we receive a tip in the above form. Therefore, it follows that two elements p_1 and p_2 are in the same set in our structure if and only if they have a chain of friendships to one another (form a squad). Furthermore, consider when we have two "squads" A and B (two sets in our structure) which contain p_i and p_j, respectively. If we receive a tip "p_i is now friends with p_j", then it follows by the problem description that A and B form one squad, since every member of A and B has a chain of friendships to each other using the new friendship between p_i and p_j. In this case, our algorithm calls Union(p_i,p_j), which replaces set A and set B with their Union. Thus, by the correctness of the Union method from the Union Find Data Structure (Lecture Notes 7), our algorithm maintains a correct mapping of the squads at Constance.

Next, we will prove that our algorithm returns the correct squad name for every request. By the definition of our algorithm, we call Find(p_i) on our Union Find Data Structure for every request for the squad name that contains p_i. As we have previously proved, every unique set in our structure corresponds to a unique squad. Thus, by the correctness of Find(x) (Lecture Notes 7), we know that our algorithm will output the correct name of the squad which contains p_i.

Therefore, we have proven that that our algorithm maintains a correct mapping of the friendships at Constance and outputs the correct squad name for every request. □

Runtime:

Proof. Let a be the number of tips and b be the number of requests. Finally, let n be the number of people at Constance, or equivalently the number of elements in our

Union Find structure. We know from lecture notes 7 that both Union() and Find() takes $O(\log^*(n))$ time (with path compression and union by rank). Thus, since our algorithm calls Union() a times and Find() b times, our algorithm runtime for a tips and b requests is $O((a+b)\log^*(n))$. □

(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).

Algorithm: We define our modified algorithm to detect circular squads as such:

Initialize a Union Find Data Structure (with path compression and union by rank). Next, call Makeset(p_i) and store a unique squad name for every person p_i at Constance High.

Next, we modify the Union(p_i,p_j) method of our structure as follows: let rootI = Find(p_i) and rootJ = Find(p_j). If rootI = rootJ, append the word "circle" to the squad name of p_i and p_j. Then, call Link(rootI,rootJ). Note that this is essentially the original implementation of Union() in Lecture Notes 7 with an additional If check before the Link() call. Finally, we define a naming convention such that when a tip causes two squads to merge into one, if either of the two squad names contains "circle", the resulting squad name also contains "circle".

For every tip received of form "p_1 is now friends with p_2", call our modified Union(p_1, p_2) in our Union Find structure. For every request for the squad name of a person p_i, call Find(p_i) and return the squad name stored in the root parent node of the set containing p_i.

Correctness: We will prove that our modified algorithm returns names that contain the word "circle" for all circular squads (and not for any other squads).

Proof. Our only modification from the algorithm in part (a) is the Union method of the Union Find Data Structure. Thus, by our correctness of our original algorithm in part (a), we know that our modified algorithm will return the correct squad name for all requests, and each distinct set in our Union Find structure corresponds to a distinct squad at Constance High. Finally, our modified naming convention guarantees that once a squad is marked as circular, it will remain marked as circular, even if merging with a non-circular squad.

Now, we will consider the correctness of our modification to Union in marking circular squads. Consider a tip "p_i is now friends with p_j" where p_i and p_j are part

of the same squad. It follows that there already exists a chain of friendships of length > 1 at Constance High between p_i and p_j[1]. Then, if p_i is now friends with p_j per the friendship tip, then there also exists a direct friendship between p_i and p_j, making their squad by definition a circular squad. Since p_i and p_j are part of the same squad, they must already exist in the same set in our Union Find Data Structure at the time of the tip. Thus, it follows that the root of p_i is the same as the root of p_j. Then, the if condition in our modified Union will be satisfied in this tip, and we add "circle" [2] to the name of the squad which contains p_i and p_j. Thus, if and only if we receive a tip "p_i is now friends with p_j" which creates a circular squad, our modified Union marks the squad name with "circle". [1]Per Ed, we know that there will be no duplicate tips. Therefore, if we receive a tip for p_i and p_j and they are part of the same squad, then we know that they have no prior friendship, which means they must be connected by a chain of friendships of length > 1. [2] Per Ed, friendships cannot be broken, so once a squad is circular, the squad will remain circular forever. Thus, we only need to consider adding "circle" to squad names, not removing "circle".

Next, we will prove that if a squad is a circular squad, then our algorithm returns a squad name that contains "circle". If a squad is circular, there exists at least one pair p_i and p_j in the squad who have a direct friendship and also linked by a longer friendship chain. Thus, since we call the Union method for every tip received, it follows that at some point in time, our algorithm called Union(p_i, p_j) such that the root of p_i was the same as the root of p_j. Therefore, the if condition would be satisfied and our algorithm would add "circle" to the squad name. Then, by the correctness of our original algorithm in handling requests, our algorithm will always return a squad name that contains "circle" if the squad is indeed circular.

Now, we will prove that if our algorithm returns a squad name that contains "circle", then the squad is a circular squad. If a request returns a squad name that contains "circle", we know by the definition of our algorithm that we received a friendship tip between p_i and p_j where the root of p_i was the same as the root of p_j. Thus, it follows from the correctness of Find (Lecture Notes 7) that p_i and p_j were already part of the same set in our Union Find Data Structure. Therefore, p_i and p_j also were part of the same squad at Constance prior to the friendship tip. Thus, as we have discussed in paragraph 2, there must have existed a chain of friendships of length > 1 between p_i and p_j before their new friendship, making their squad circular. Hence, by definition of a circular squad, if our algorithm returns a squad name that contains "circle", then the squad is a circular squad.

Thus, we have proved that our modified algorithm returns names that contain the word “circle” for all circular squads (and not for any other squads). □

Runtime:

Proof. Let a be the number of tips and b be the number of requests. Finally, let n be the number of people at Constance, or equivalently number of elements in our

Union Find structure. We know from lecture notes 7 that Find() takes $O(\log^*(n))$ time (with path compression and union by rank). Furthermore, under the assumption that checking the equivalence of two elements and appending "circle" takes constant runtime, our modified Union method still takes $O(\log^*(n))$ since we do not additionally call Find. Thus, since our algorithm calls Union() a times and Find() b times, our algorithm runtime for a tips and b requests is $O((a+b)\log^*(n))$. □

7. **Greedy scheduling (35 points)** Consider the following scheduling problem: we have two machines, and a set of jobs $j_1, j_2, j_3, \ldots, j_n$ that we have to process. To process a job, we need to assign it to one of the two machines; each machine can only process one job at a time. Each job j_i has an associated positive integer running time r_i. The load on the machine is the sum of the running times of the jobs assigned to it. The goal is to minimize the completion time, which is the maximum of the load of the two machines.

   Suppose we adopt a greedy algorithm: for every i, job j_i is assigned to the machine with the minimum load after the first $i-1$ jobs. (Ties can be broken arbitrarily.)

   (a) **(5 points)** For all $n > 3$, give an instance of this problem for which the completion time of the assignment of the greedy algorithm is a factor of 3/2 away from the best possible assignment of jobs.

   Consider a scheduling of jobs $j_1, j_2, j_3, \ldots, j_n$ such that $r_1 = n-2$, $r_2 = r_3 = \ldots = r_{n-1} = 1$, and $r_n = 2(n-2)$. Thus, our total runtime $S = \Sigma_{i=0}^{n} r_i = (n-1)+2(n-1)+(n-1) = 4(n-1)$. Note that at the bare minimum, both machines much process half of the sum of the run times. This is because by their definition, the two loads on the machines must add up to the sum of the total run times (since each job must be assigned to either the load of machine 1 or 2). Thus, without loss of generality, if the first machine has a load $l_1 \leq S/2$, then the second machine has a load $l_2 \geq S/2$ and the completion time will be $l_2 \geq S/2$. Therefore, in this case, the best possible assignment that minimizes the completion time would be assigning an equal load of $2(n-2)$ onto each machine. However, consider the assignment of the greedy algorithm, where we handle any tie by assigning the job to machine 1. At $i = 1$, our algorithm assigns j_1 with $r_1 = n-2$ to machine 1. Then, for $i = 2 \ldots (n-1)$ with $r_2 \ldots r_{n-1} = 1$, our algorithm assigns $j_2 \ldots j_{n-1}$ to machine 2 since machine 1 has a load of $n-2$ already, which will be greater than the load of machine 2 until j_{n-1}. Now, at the time of before the assigning of the last job j_n, it follows that both machines have load $n-2$. Then, since there is a tie, we assign the final job j_n with $r_n = 2(n-2)$ to machine 1. Therefore, such a scheduling of jobs will result in machine 1 has the maximum load of $3(n-2)$, which is a factor of 3/2 away from the best possible assignment of jobs $2(n-2)$ for all $n > 3$.

   (b) **(15 points)** Prove that the greedy algorithm always yields a completion time within a factor of 3/2 of the best possible placement of jobs. (Hint: Think of the best possible placement of jobs. Even for the best placement, the completion time is at least as big as the biggest job, and at least as big as half the sum of the jobs. You may want to use both of these facts.)

Proof. Let us define L_2 and L_1 being the loads of machine 2 and machine 1, respectively, at the termination of the greedy algorithm. In addition, let r_m be the largest runtime in the set of jobs $j_1, j_2, j_3, \ldots, j_n$. Let $S = \Sigma_{i=1}^{n} r_i$ such that it is the total runtime of all the jobs. Finally, let T_i be the ideal completion time of the machines with the best possible assignment of jobs, and let T_g be the completion time of the greedy algorithm. Thus, it follows by definition of ideal completion time that $T_i \leq T_g$.

Even with the best possible placement of jobs, $T_i = \max(L_1, L_2) \geq S/2$ since the machines must at minimum still evenly split the sum of the jobs. Furthermore, $T_i \geq r_m$ since at least one of the machines will need to handle the largest job.

Now, without loss of generality, let $L_2 > L_1$. Thus, by the definition of the problem, the completion time of our greedy algorithm is $T_g = L_2$. Let us define r_k as the runtime of the last job assigned to L_2. Thus, it follows that $L_2 = L_2' + r_k$, where L_2' was the load of machine 2 prior to adding r_k. Since we added r_k to L_2', it follows by definition of our greedy algorithm that $L_2' \leq L_1'$, where L_1' is the load of machine 1 prior to job j_k. Therefore, by the previous statement, the maximum value (upper bound) of L_2' is when $L_2' = L_1'$. This condition itself has an upper bound of $L_2' = L_1' \leq \frac{S-r_k}{2}$, since the maximum value of $L_2' = L_1'$ is achieved when r_k is the last job in the schedule so that the sum of all other jobs prior to job j_k would be maximized. Thus, it follows that $L_2' \leq \frac{S-r_k}{2}$. Using this inequality, we plug this into our definition $L_2 = L_2' + r_k$ to yield $L_2 \leq \frac{S-r_k}{2} + r_k$. Simplifying this, we get $L_2 \leq \frac{S}{2} + \frac{r_k}{2}$. Now, consider the upper bound of r_k. By the definition of a maximum, we are guaranteed that $r_k \leq r_m$. Thus, we have $L_2 \leq \frac{S}{2} + \frac{r_k}{2} \leq \frac{S}{2} + \frac{r_m}{2}$.

Recall that we showed $T_i \geq S/2$ and $T_i \geq r_m$ earlier. Hence, $\frac{T_i}{2} \geq \frac{r_m}{2}$. Therefore, it follows through inequality relationships that $L_2 \leq \frac{S}{2} + \frac{r_k}{2} \leq \frac{S}{2} + \frac{r_m}{2} \leq \frac{3}{2}T_i$, or $L_2 \leq \frac{3}{2}T_i$. However, recall that $L_2 = T_g$ because we assumed w.l.o.g. that $L_2 > L_1$. Then, $L_2 = T_g \leq \frac{3}{2}T_i$. In addition, recall that $T_i \leq T_g$ since by definition of an ideal runtime, no other runtime can be faster. Thus, we have proved that $T_i \leq T_g \leq \frac{3}{2}T_i$. □

(c) **(10 points)** Suppose now instead of 2 machines we have m machines and the completion time is defined as the maximum load over all the m machines. Prove the best upper bound you can on the ratio of the performance of the greedy solution to the optimal solution, as a function of m?

Proof. Let us define $L_1, ..., L_m$ as being the loads of machine 1 through machine m at the termination of the greedy algorithm. In addition, let r_m be the largest runtime in the set of jobs $j_1, j_2, j_3, \ldots, j_n$. Let $S = \Sigma_{i=1}^{n} r_i$ such that it is the total runtime of all the jobs. Finally, let T_i be the ideal completion time of the machines with the best possible assignment of jobs, and let T_g be the completion time of the greedy algorithm. Thus, it follows by definition of ideal completion time that $T_i \leq T_g$.

Even with the best possible placement of jobs, $T_i = \max(L_1, L_2, ..., L_m) \geq S/m$

since the machines must at minimum still evenly split the sum of the jobs. Furthermore, $T_i \geq r_m$ since at least one of the machines will need to handle the largest job.

Now, without loss of generality, let $L_i > L_j$ for all $j \in \{1,...,m\} \setminus \{i\}$. Thus, by the definition of the problem, the completion time of our greedy algorithm is $T_g = L_i$. Let us define r_k as the runtime of the last job assigned to L_i. Thus, it follows that $L_i = L'_i + r_k$, where L'_i was the load of machine 2 prior to adding r_k. Since we added r_k to L'_i, it follows by definition of our greedy algorithm that $L'_i \leq L'_j$ for all $j \in \{1,...,m\} \setminus \{i\}$, where L'_j is the load of machine j prior to job j_k. Therefore, by the previous statement, the maximum value (upper bound) of L'_i is when $L'_i = L'_1 = L'_2 = ... = L'_m$. This condition itself has an upper bound of $L'_i = L'_1 = L'_2 = ... = L'_m \leq \frac{S-r_k}{m}$, since the maximum value of $L'_i = L'_1 = L'_2 = ... = L'_m$ is achieved when r_k is the last job in the schedule so that the sum of all other jobs prior to job j_k would be maximized. Thus, it follows that $L'_i \leq \frac{S-r_k}{m}$. Using this inequality, we plug this into our definition $L_i = L'_i + r_k$ to yield $L_i \leq \frac{S-r_k}{m} + r_k$. Simplifying this, we get $L_i \leq \frac{S}{m} + \frac{(m-1)r_k}{m}$. Now, consider the upper bound of r_k. By the definition of a maximum, we are guaranteed that $r_k \leq r_m$. Thus, we have $L_i \leq \frac{S}{m} + \frac{(m-1)r_k}{m} \leq \frac{S}{m} + \frac{(m-1)r_m}{m}$.

Recall that we showed $T_i \geq S/m$ and $T_i \geq r_m$ earlier. Hence, $\frac{(m-1)T_i}{m} \geq \frac{(m-1)r_m}{m}$. Therefore, it follows through inequality relationships that $L_i \leq \frac{S}{m} + \frac{(m-1)r_k}{m} \leq \frac{S}{m} + \frac{(m-1)r_m}{m} \leq \frac{2m-1}{m} T_i$, or $L_i \leq \frac{2m-1}{m} T_i$. However, recall that $L_i = T_g$ because we assumed w.l.o.g. that $L_i > L_j$ for all $j \in \{1,...,m\} \setminus \{i\}$. Then, $L_i = T_g \leq \frac{2m-1}{m} T_i$. In addition, recall that $T_i \leq T_g$ since by definition of an ideal runtime, no other runtime can be faster. Thus, we have proved that $T_i \leq T_g \leq \frac{2m-1}{m} T_i$. □

(d) **(5 points)** Give a family of examples (that is, one for each m – if they are very similar, it will be easier to write down!) where the factor separating the optimal and the greedy solutions is as large as you can make it.

If the factor separating the optimal and the greedy solutions for a certain m is as large as possible, then by our derivation in part (c) we must have that the completion time of the greedy algorithm $T_g = \frac{2m-1}{m} T_i$, where T_i is the best case completion time.

First, if $m = 1$, we only have 1 machine to handle the schedule of jobs. Thus, it follows that there is only one allocation of runtimes possible, meaning that our $T_g = T_i$, which we can verify with our formula $T_g = \frac{2-1}{1} T_i$. Therefore, for $m = 1$, we can use any arbitrary schedule of jobs because there is the only possible factor separating the optimal and greedy solutions is 1.

Now for $m \geq 2$, consider a schedule of $m(m-1)+1$ jobs where $r_1 = r_2 = ... = r_{m(m-1)} = 1$ and $r_{m(m-1)+1} = m$. Thus, our total runtime $S = \Sigma_{i=0}^{m(m-1)+1} r_i = (m(m-1)) + (m) = m^2$. It follows that the best possible assignment of job gives a completion time of $S/m = m$ since at the bare minimum all m machines much equally share the sum of all run

times in their loads. This is because by their definition, the m loads on the machines must add up to the sum of the total run times (since each job must be assigned to the load of one of the m machines). Thus, without loss of generality, if the first machine has a load $l_1 \leq S/m$, then the another machine must compensate with a load $l_i \geq S/m$ and the completion time will be at least $l_i \geq S/m$ since it is the maximum of all the machine loads. Now, let us consider the behavior of our greedy algorithm on our defined job schedule. Since we have $r_1 = r_2 = ... = r_{m(m-1)} = 1$, our greedy algorithm will repeatedly these $m(m-1)$ jobs across all m machines, leaving each machine with an equal load of $(m-1)$ after $j_{m(m-1)}$. Then, once our greedy algorithm reaches the final job $j_{m(m-1)+1}$ with $r_{m(m-1)+1} = m$, it adds $r_{m(m-1)+1}$ arbitrarily to one of the m machines since they carry equal load at that point. Thus, one machine will end with a load of $(m-1)+m = 2m-1$ and the other $m-1$ machines will end with a load of $(m-1)$. Since our completion time is the maximum of all loads, it follows that our completion time of the greedy algorithm for such a job schedule is $2m-1$, which is factor $\frac{2m-1}{m}$ away from the best possible completion time of m. Therefore, this scheduling of jobs will have the factor separating the optimal and greedy solutions as large as the upper bound derived in (c) for all $m \geq 2$.