Honor Code Notice. This document is for exclusive use by **Fall 2023 CS3100** students with Professor Hott and Professor Horton at The University of Virginia. Any student who references this document outside of that course during that semester (including any student who retakes the course in a different semester), or who shares this document with another student who is not in that course during that semester, or who in any way makes copies of this document (digital or physical) without consent of Professor Hott or Professor Horton is **guilty of cheating**, and therefore subject to penalty according to the University of Virginia Honor Code.

problem 1 *Scenic Highways*

In this problem we'll describe something similar to solving a problem with Dijkstra's algorithm, and ask you some questions about this new problem and a possible algorithm.

You are taking a driving vacation from town to town until you reach your destination, and you don't care what route you take as long as you maximize the number of scenic highways between towns that are part of your route. You model this as a weighted graph G, where towns are nodes and there is an edge for each route between two towns. An edge has a weight of 1 if the route is a scenic highway and a 0 if it is not.

You need to find the path between two nodes s and t that maximizes the number of scenic highways included in the path. You design a greedy algorithm that builds a tree like Dijkstra's algorithm does, where one node is added to a tree at each step. From the nodes adjacent to the tree-nodes that have already been selected, your algorithm uses this greedy choice:

Choose the node that includes the most scenic highways on the path back to the start node s.

When node t is added to the tree, stop and report that path found from s to t. Reminder (in case it helps you): the greedy choice for Dijkstra's algorithm was:

Choose the node that has the shortest path back to the start node s.

1. What is the key-value stored for each item in the priority-queue? Also, what priority-queue operation must be used when we need to update a key for an item already stored in the priority-queue?

2. Draw one or more graphs to convince yourself that the greedy approach describes above does not work. Then explain in words as best you can the reason this approach fails or a situation that will cause it to fail (i.e., a counterexample).

Solution:

Solution: First, the key-value is the number of scenic highways in the path from s to the node. Second, unlike Dijkstra's, this will be a max-PQueue. So the *Increase-Key* operation must be used when we find a better path to a node in the PQueue. (The SP and MST algorithms we saw in class used Decrease-Key.)

It does not work. This is like a "longest-path" algorithm, and a partial path that starts off badly but is really the best will not be chosen if a sub-optimal path starts off well and has a final edge to the destination t. That final edge will be chosen and the algorithm stops with the wrong answer before we have a chance to pursue the optimal path that started off badly.

Here are the set of weighted edges for a simple graph that shows how this fails:

$$E = \{(s,1,1),(a,t,0),(s,b,0),(b,c,1),(c,t,1)\}$$

The best path is *sbct* with a score of 2, but the algorithm will add a to the tree and then t, with a score of 1.

PROBLEM 2 *As You Wish*

Buttercup has given Westley a set of n tasks $T = t_1, \ldots, t_n$ to complete on the farm. Each task $t_i = (d_i, w_i)$ is associated with a deadline d_i and an estimated amount of time w_i needed to complete the task. To express his undying love to Buttercup, Westley strives to complete all the assigned tasks as early as possible. However, some deadlines might be a bit too demanding, so it may not be possible for him to finish a task by its deadline; some tasks may need extra time and therefore will be completed late. Your goal (inconceivable!) is to help Westley minimize the deadline overruns of any task; if he starts task t_i at time s_i, he will finish at time $f_i = s_i + w_i$. The deadline overrun (or lateness) of tasks—denoted L_i—for t_i is the value

$$L_i = \begin{cases} f_i - d_i & \text{if } f_i > d_i \\ 0 & \text{otherwise} \end{cases}$$

Design a polynomial-time algorithm that computes the optimal order W for Westley to complete Buttercup's tasks so as to minimize the maximum L_i across all tasks. That is, your algorithm should compute W that minimizes

$$\min_{W} \max_{i=1,\ldots,n} L_i$$

In other words, you do not want Westley to complete *any* task *too* late, so you minimize the deadline overrun of the task completed that is most past its deadline.

1. What is your algorithm's greedy choice property?
2. What is the run-time of your algorithm?
3. Consider the following example list of tasks:

   $$T = \{(2,2),(11,1),(8,2),(1,5),(20,4),(4,3),(8,3)\}$$

   List the tasks in the order Westley should complete them. Then argue why there is no better result for the given tasks than the one your algorithm (and greedy choice property) found.

Solution:

Solution: Our algorithm will order the tasks in ascending order by deadline. In particular, Westley will complete the task with the earliest deadline first. This yields an algorithm with running time $\Theta(n \log n)$.

We show that our algorithm is correct using an exchange argument. Assume we have some optimal schedule OPT. We will show that our greedy solution is no worse than OPT.

- **Case 1:** OPT matches greedy. In this case, the greedy algorithm is optimal.
- **Case 2:** OPT differs from greedy. Then, there must exist two consecutive tasks t_i, t_{i+1} in OPT where $d_i \geq d_{i+1}$. In this case we will show that the maximum lateness of the optimal schedule is at least the maximum lateness of the greedy schedule by showing that swapping the order of t_i and t_{i+1} will not increase the maximum lateness. This implies that for any pair of tasks that are completed out of order relative to the greedy solution, we can swap

them without decreasing the quality of the schedule. Effectively, this means that we can take any optimal solution, and repeatedly swap the orders of any two adjacent tasks t_i, t_{i+1} where $d_i \geq d_{i+1}$ until we arrive at the greedy ordering without increasing the maximum lateness.

Let NEW be the solution obtained by swapping the order of t_i, t_{i+1} in OPT. Namely, NEWand OPTare identical except the order of tasks t_i and t_{i+1} is swapped. We will show that the maximum lateness of schedule NEWis no greater than that of OPT.

By design, every task in NEWand OPTother than t_i and t_{i+1} complete at the same time. Next, task t_{i+1} now completes at an earlier time in NEWcompared to OPT(since t_{i+1} is completed before t_i in NEW). Thus, if the maximum lateness of NEWis greater than that of OPT, it must be due to the completion time of task t_i, which now completes *later* than in OPT. Let f'_i be the time NEWcompletes task t_i, and let f_{i+1} be the time OPTcompletes task t_{i+1}. Since OPT and NEW just interchanges the order of tasks t_i and t_{i+1}, we have that $f'_i = f_{i+1}$. Finally, let L'_i denote the lateness of task t_i in NEWand L_{i+1} denote the lateness of task t_{i+1} in OPT. By definition, $L'_i = f'_i - d_i$, and so,

$$L_{i+1} = f_{i+1} - d_{i+1} = f'_i - d_{i+1} \geq f'_i - d_i = L'_i,$$

where we have used the inequality $d_i \geq d_{i+1}$. This means that the maximum lateness in OPTis at least L'_i. Since the only task whose lateness has increased compared to OPTis t_i, and the lateness L'_i of t_i in NEWis at most the maximal lateness of OPT, the maximum lateness in NEWcan be no worse than OPT, which proves the claim.

PROBLEM 3 *Course Scheduling*

The university registrar needs your help in assigning classrooms to courses for the spring semester. You are given a list of n courses, and for each course $1 \leq i \leq n$, you have its start time s_i and end time e_i. Give an $O(n \log n)$ algorithm that finds an assignment of courses to classrooms which minimizes the *total number* of classrooms required. Each classroom can be used for at most one course at any given time. Prove both the correctness and running time of your algorithm.

Solution:

Solution: We use a greedy algorithm. First, sort all of the courses by their start time. We will separately maintain a min-heap `classrooms` for classrooms, ordered according to the end time of their currently-scheduled course (if any). This is initially empty. Then, for each course $c_i = (s_i, e_i)$, in order of their starting time, check if there is a classroom available. Namely, check if `classrooms` is non-empty and if so, whether the end time for the minimum element (i.e., the root node) is less than or equal to s_i.

- If there is an available classroom, pop the classroom from the heap, update its end-time to e_i and re-insert it into `classrooms`. Assign c_i to this classroom.
- Otherwise, if there are no available classrooms, allocate a new classroom, set its end-time to e_i and add it to `classrooms`. Assign c_i to this new classroom.

To argue correctness, optimality, and the running time of the algorithm, observe the following:

- **Correctness:** By construction, the algorithm only assigns a course $c_i = (s_i, e_i)$ to a classroom if the end-time of the previously-scheduled course in the classroom is less than or equal to s_i. Thus, the assignment output by this algorithm is always legal.

- **Optimality:** Let k be the number of classrooms used by our greedy classroom assignment algorithm above. We claim that k is the minimum number of classrooms needed. In particular, we show that there is a point in time where there are exactly k overlapping lectures (in which case, every legal schedule requires at least k classrooms). To see this, suppose the above algorithm allocates the k^{th} classroom when scheduling course $c_i = (s_i, e_i)$. Consider the state at time s_i. At this moment in time, there are exactly $k-1$ allocated classrooms in `classrooms`, and moreover, the course currently assigned to each classroom ends at a time *strictly* greater than s_i (by construction) *and* started before s_i (since the courses are assigned in order of start time). This means that at time s_i, there are at least k classes still in session, so any valid schedule must make use of at least k classrooms.
- **Running time:** The running time of this algorithm is $O(n \log n)$. First, sorting the courses by their start time requires $O(n \log n)$ time. The size of `classrooms` is at most n (every course gets its own classroom), so each update to `classrooms` can be completed in time $O(\log n)$ (namely, both extract-min and insertion can be handled in $O(\log n)$ time). Thus, the overall complexity is $O(n \log n) + O(n \log n) \in O(n \log n)$.

PROBLEM 4 *Ubering in Arendelle*

After all of their adventures and in order to pick up some extra cash, Kristoff has decided to moonlight as the sole Uber driver in Arendelle with the help of his trusty reindeer Sven. They usually work after the large kingdom-wide festivities at the palace and take everyone home after the final dance. Unfortunately, since a reindeer can only carry one person at a time, they must take each guest home and then return to the palace to pick up the next guest.

There are n guests at the party, guests $1, 2, ..., n$. Since it's a small kingdom, Kristoff knows the destinations of each party guest, $d_1, d_2, ..., d_n$ respectively and he knows the distance to each guest's destination. He knows that it will take t_i time to take guest i home and return for the next guest. Some guests, however, are very generous and will leave bigger tips than others; let T_i be the tip Kristoff will receive from guest i when they are safely at home. Assume that guests are willing to wait after the party for Kristoff, and that Kristoff and Sven can take guests home in any order they want. Based on the order they choose to fulfill the Uber requests, let D_i be the time they return from dropping off guest i. Devise a greedy algorithm that helps Kristoff and Sven pick an Uber schedule that minimizes the quantity:

$$\sum_{i=1}^{n} T_i \cdot D_i.$$

In other words, they want to take the large tippers the fastest, but also want to take into consideration the travel time for each guest. Prove the correctness of your algorithm. (Hint: think about a property that is true about an optimal solution.)

Solution:

Solution: Our algorithm will order the uber rides in descending order by $\frac{T_i}{t_i}$, i.e. it will make the ride with the highest tip-for-time $\frac{T_i}{t_i}$ first, and thus runs in $\Theta(n \log n)$ time by using mergesort.

We will show that our algorithm is correct using an exchange argument. Assume we have some optimal schedule OPT. We will show that our greedy solution is no worse than that optimal solution.

Case 1: OPT matches greedy. In this case certainly greedy is optimal

Case 2: OPT differs from greedy (i.e. there are two consecutive Uber rides d_i, d_{i+1} in OPT with $\frac{T_i}{t_i} \leq \frac{T_{i+1}}{t_{i+1}}$. In this case we will show that the cost of the optimal schedule is at least the cost of the greedy schedule by showing that swapping the order of d_i and d_{i_1} will not increase the

cost of the schedule. This implies that for any pair of rides that are out of order relative to the greedy solution, we can swap them without decreasing the quality of the schedule. Effectively, this shows that one can take any optimal solution, and do an insertion/bubble sort to convert it into the greedy solution without increasing the cost of the schedule.

Consider some arbitrary optimal schedule OPT which differs from our greedy solution. Let d_i, d_{i+1} be an arbitrary consecutive pair of rides in OPT s.t. $\frac{T_i}{t_i} \leq \frac{T_{i+1}}{t_{i+1}}$. We will show that the cost of schedule NEW generated by swapping the order of d_i, d_{i+1} is no higher than the cost of OPT.

Note that NEW and OPT differ only in the order that guest rides to d_i and d_{i+1} are made. Let K represent the part of the summation $\sum_{i=1}^{n} T_i \cdot D_i$ which is shared among the two schedules (i.e. the first $i-1$ Uber rides and the last $n-i-2$ Uber rides).

The cost of OPT is given by the expression:

$$K + T_i(D_{i-1} + t_i) + T_{i+1}(D_{i-1} + t_i + t_{i+1})$$

The cost of NEW is given by the expression:

$$K + T_{i+1}(D_{i-1} + t_{i+1}) + T_i(D_{i-1} + t_i + t_{i+1})$$

We wish to show:

$$\begin{aligned}
K + T_i(D_{i-1} + t_i) + T_{i+1}(D_{i-1} + t_i + t_{i+1}) &\geq K + T_{i+1}(D_{i-1} + t_{i+1}) + T_i(D_{i-1} + t_i + t_{i+1}) \\
T_i(D_{i-1} + t_i) + T_{i+1}(D_{i-1} + t_i + t_{i+1}) &\geq T_{i+1}(D_{i-1} + t_{i+1}) + T_i(D_{i-1} + t_i + t_{i+1}) \\
T_iD_{i-1} + T_it_i + T_{i+1}(D_{i-1} + t_i + t_{i+1}) &\geq T_{i+1}(D_{i-1} + t_{i+1}) + T_iD_{i-1} + T_it_i + T_it_{i+1} \\
T_{i+1}(D_{i-1} + t_i + t_{i+1}) &\geq T_{i+1}(D_{i-1} + t_{i+1}) + T_it_{i+1} \\
T_{i+1}D_{i-1} + T_{i+1}t_i + T_{i+1}t_{i+1} &\geq T_{i+1}D_{i-1} + T_{i+1}t_{i+1} + T_it_{i+1} \\
T_{i+1}t_i &\geq T_it_{i+1} \\
\frac{T_{i+1}}{t_{i+1}} &\geq \frac{T_i}{t_i}
\end{aligned}$$