Problem 1. (30 points)

Suppose you are given a set \( T = \{(s_1, f_1), \ldots, (s_n, f_n)\} \) of \( n \) tasks, where each task \( i \) is defined by the start time \( s_i \) and a finish time \( f_i \). Two tasks \( t_i \) and \( t_j \) are non-conflicting if \( f_i \leq s_j \) or \( f_j \leq s_i \). The activity selection problem asks for the largest number of tasks can be scheduled in a non-conflicting way. The greedy algorithm explained in class considers tasks one by one ordered by the finish time. Ordering by increasing finish time is crucial to prove that this strategy always leads to the optimal solution.

Does the following greedy algorithm compute the optimal solution for activity selection?

1. Compute the number of overlaps for each task
2. Sort the task by the number of overlaps, in increasing order (break ties arbitrarily)
3. Pick the task \( i \) with the smallest number of overlaps, schedule it, and remove from further consideration tasks that are overlapping with \( i \)
4. Repeat step 3 until all tasks are scheduled

Note that the number of overlaps is NOT updated after step 1. If you think the strategy works, prove that the greedy choice property hold. If not, show an example where this strategy gives a suboptimal solution.

Answer: The strategy is not optimal. Consider the following tasks.

The optimal number of tasks is four (boxed), but the algorithm will pick the dashed tasks (the first one in the middle has two conflicts, the two on the sides has 3).

The optimal number of tasks is four (boxed), but the algorithm can pick the dashed tasks (the two on the sides have one conflict so they are picked first).

Problem 2. (35 points)

A server has \( n \) customer waiting to be served. The service time required by each customer is known in advance: it is \( t_i \) minutes for customer \( i \). So if, for example, the customers are served in order of increasing \( i \), then the \( i \)-th customer has to wait \( \sum_{j=1}^{i} t_j \) minutes. We want to minimize the total waiting time:

\[
T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i)
\]
Give a greedy (efficient) algorithm for computing the optimal order in which to process the customers. Prove the correctness of your algorithm by showing greedy choice and optimal substructure.

**Answer:** Our algorithm follows a greedy strategy, by sorting the customers in the increasing order of service times and servicing them in this order. The running time is $O(n \log n)$.

First we prove the correctness directly (instead of using the greedy choice/optimal substructure proof). For any ordering of the customers, let $\sigma(j)$ denote the $j$-th customer in the ordering. Then:

$$T = \sum_{i=1}^{n} \sum_{j=1}^{i-1} t_{\sigma(i)} = \sum_{i=1}^{n} (n - i) t_{\sigma(i)}$$

For any ordering, if $t_{\sigma(i)} > t_{\sigma(j)}$ for $i < j$, then swapping the positions of the two customers gives a better ordering (i.e., the value of $T$ is decreased). Since we can generate all possible orderings by swaps, an ordering which has the property that $t_{\sigma(1)} \leq \ldots \leq t_{\sigma(n)}$ must be the global optimum.

Instead if one decided to prove the greedy choice property and the optimal substructure, here is my solution. For the greedy choice, we need to show that there is at least one optimal solution that contains the greedy choice, i.e., serving first the customer that has the smallest service time. **Proof:** Let $A$ be an optimal ordering where $\sigma(j)$ denote the $j$-th customer in the ordering. If $A$ begins with the greedy choice, i.e., if $\sigma(1) = j$ for the shortest service time $t_j = \min_i \{t_i\}$, we are done. If not, then the first customer in $A$ must have a service time $t_k$ equal to $t_j$ (for some $k \neq j$), otherwise the total waiting time

$$T = \sum_{i=1}^{n} \sum_{j=1}^{i-1} t_{\sigma(i)} = \sum_{i=1}^{n} (n - i) t_{\sigma(i)}$$

would increase, violating the optimality of $A$. If we swap out customer $k$ and swap in customer $j$ in $A$, we can create a new ordering $B$ which begins with the greedy choice, as we wanted.

For the optimal substructure we need to show that if $A$ is an optimal ordering for the set of all the customers $S = \{1, 2, \ldots, n\}$ and $j$ is (one of) the customers with the smallest service time, then $A' = A - \{j\}$ is an optimal ordering for the set of customers $S' = \{1, 2, \ldots, j-1\} \cup \{j+1, j+2, \ldots n\}$. **Proof:** For sake of contradiction, let’s assume that $A'$ is not optimal for $S'$ (while at the same time $A$ is optimal for $S$). That means that we can find another solution $B$ for $S'$ which the total waiting time is shorter. However recall that

$$T = \sum_{i=1}^{n} (n - i) t_{\sigma(i)} = (n - 1) t_{\sigma(1)} + \sum_{i=2}^{n} (n - i) t_{\sigma(i)}$$

If $B$ has a shorter total waiting time for $S'$ that means that $\sum_{i=2}^{n} (n - i) t_{\sigma(i)}$ is smaller. In this case we could add to $B$ the greedy choice $j$ and obtain a better solution than $A$, which creates a contradiction because $A$ is optimal.
**Problem 3.** (35 points)

Let $G = (V, E)$ be a weighted undirected graph. We define the *bottleneck* of a path $p$ in $G$ as the minimum weight of any edge on $p$. We define the maximum bottleneck of any $s,t$-path as the maximum, over all paths $p$ from $s$ to $t$, of the bottleneck of $p$.

Prove or disprove: given any connected, undirected, edge-weighted graph, algorithm Bottleneck below produces a tree $T$ such that the bottleneck of the path $p$ from $s$ to $t$ in $T$ is the maximum bottleneck of any $s,t$-path in the original graph.

**Algorithm Bottleneck** ($G(V,E) : graph$)

1. sort the edges $e_1, e_2, \ldots, e_m$ in order of decreasing cost
2. $T \leftarrow \emptyset$
3. for $i \leftarrow 1, 2, \ldots, m$ do
   - Add $e_i$ to $T$ if doing so does not create a cycle in $T$
4. return $T$

**Answer:** The algorithm is correct. The algorithm produces a spanning tree $T$ of the graph for the same reason that Kruskal’s algorithm does. At each point in time, each edge considered but not added by the algorithm creates a cycle with the edges added so far. This implies that: (*) if you can get from a vertex $u$ to a vertex $w$ using just the edges considered so far, then you get get from $u$ to $w$ using just the edges added so far. Now consider the path $p$ connecting $s$ to $t$ in $T$. Consider the last edge $e_i$ on $p$ added to $T$ by the algorithm. The property (*) above implies that there is no path from $s$ to $t$ using just the edges $\{e_1, e_2, \ldots, e_{i-1}\}$. In other words, every path from $s$ to $t$ uses some edge in $\{e_i, e_{i+1}, \ldots, e_m\}$. Keeping in mind that the edges are considered in order of decreasing weight, this implies that every path from $s$ to $t$ uses an edge of weight at most $w(e_i)$. This implies that every path from $s$ to $t$ has bottleneck at most $w(e_i)$. 