Introduction to Algorithms **Solution Set 2**
CS 4820, Spring 2018

(1)

(1.a) *(5 points)*
Consider the scheduling problem in Section 4.2 of the textbook. Suppose the goal is to minimize the sum of the latenesses of requests. Show that for this objective function, the earliest-deadline-first algorithm does not always find an optimal schedule.

(1.b) *(10 points)*
Again consider the scheduling problem in Section 4.2 of the textbook. Suppose every request has a positive weight w_i and the goal is to minimize the weighted sum of latenesses, $\sum_i w_i \ell_i$.

Give an efficient algorithm for the special case that all deadlines are equal to the time the resource becomes available (i.e., $d_i = s$ for all $i \in \{1, \dots, n\}$).

Solution:

Part a. The following is one idea to construct a counterexample: if the goal is to minimize the maximum lateness, it doesn't matter how many jobs are late. On the other hand, if the goal is to minimize the sum of latenesses, then schedules with a small number of late jobs are preferable.

Here is a counterexample that follows this idea: The starting time for the resource is $s = 0$. We have one request with length k and deadline k. In addition, we have k requests of length 1 and deadline $k + 1$.

In the earliest-deadline-first schedule, the request of length k is scheduled first and the other requests are scheduled after it. The latenesses of the short reqeusts are $0, 1, \dots, k - 1$. Therefore, the sum of latenesses is $\frac{1}{2}k(k - 1)$.

At the same time, if we schedule the short requests first and the long request after them, the total lateness is k (the short requests finish before their deadline and the long request has lateness k).

It follows that for $k > 3$, the schedule computed by the earliest-deadline-first algorithm is not optimal (and the gap grows quite large as $k \to \infty$).

Part b. Denote the processing time of request i by t_i. The algorithm schedules the the requests in decreasing order of w_i/t_i.

Running time: Computing the ratio for each job requires $O(n)$ time, and sorting them requires $O(n \log n)$ time.

Correctness: The proof of correctness uses an exchange argument, similar to the proof of correctness of the scheduling algorithm in Section 4.2 of Kleinberg & Tardos. If you have a schedule with two consecutive requests i, j (in that order) and you swap the order of i and j, the net change can be broken down as follows.

- Request i is delayed by t_j, which increases the weighted sum of latenesses by $w_i t_j$.
- Request j finishes t_i time steps earlier, which decreases the weighted sum of latenesses by $w_j t_i$.
- All other requests at the same time as they did before the swap.

Thus the net change in the weighted sum of latenesses is

$$w_i t_j - w_j t_i = \left(\frac{w_i}{t_i} - \frac{w_j}{t_j} \right) t_i t_j. \tag{1}$$

This implies the following facts.

1. In any optimal schedule, there can't be two consecutive jobs i, j such that $(w_i/t_i) - (w_j/t_j)$ is negative. If there were, one could decrease the weighted sum of latenesses by swapping the order of i and j.
2. Therefore, in every optimal schedule, the jobs are sorted in order of decreasing w_i/t_i.
3. Swapping the order of two consecutive jobs i, j such that $w_i/t_i = w_j/t_j$ has no effect on the weighted sum of latenesses.
4. Therefore, for every ordering of the jobs such that the ratios w_i/t_i form a non-increasing sequence, the cost is equal to that of an optimal schedule.

(2) *(10 points) Suppose we are given a tree, T, and a set of paths in T. Two paths are called* compatible *if they have no vertices in common. Design an algorithm to select a maximum cardinality subset of the paths, such that every two paths in the subset are compatible.*

Solution: Choose an arbitrary node of T, call it the root, and denote it by r. For every other node u, let $d(u)$ denote the length (number of edges) of the (unique) path from r to u in T. We will call this quantity the *depth* of u. For any path P, let $d(P)$ denote the minimum of $d(u)$ over the nodes in P, and let $u(P)$ denote the node of P that is closest to r; in other words, $u(P)$ is chosen such that $d(u(P)) = d(P)$.

We will describe and analyze an algorithm that could be called "Deepest Path First". As the algorithm runs, each path a status which is either "active", "selected", or "deleted". maintains an array

Algorithm 1 Deepest Path First

```
1: Input: Tree $T$, paths $P_1, \ldots, P_m$.
2: Compute $d(u)$ for every $u$, using breadth first search.
3: Compute $d(P_i)$ and $u(P_i)$ for $i = 1, \ldots, m$.
4: Mark all paths as active.
5: while there exists an active path do
6:     Let $P_i$ be a path with the maximum $d(P_i)$ among active paths.
7:     Mark $P_i$ as selected.
8:     Mark every active path containing $u(P_i)$ as deleted.
9: end while
10: Output the set of selected paths.
```

Lemma 1. *If P is any path contained in T, and Q is a path from a node of P to a node whose depth is at most $d(P)$, then Q contains $u(P)$.*

Proof. Let a, b denote the endpoints of Q, labeled such that $a \in P$ and $d(b) \leq d(P)$. Let c denote the least common ancestor of a and b in T, when r is treated as the root of T. Since a tree contains a unique path between any two of its nodes, Q must be equal to the path formed by concatenating the paths from a to c and from c to b. Since c is an ancestor of b, we have $d(c) \leq d(b) \leq d(P)$. Any path from a to an ancestor of a at depth less than or equal to $d(P)$ must pass through the ancestor at depth $d(P)$, which is $u(P)$. Hence, $u(P)$ belongs to the path from a to c, which is a subpath of Q. □

Lemma 2. *The Deepest Path First algorithm selects a non-conflicting set of paths.*

Proof. Suppose P_i and P_j are any two paths selected by the algorithm, with P_i being selected earlier. At the end of the iteration in which P_i was selected, P_j must have been active. (Else P_j could not have been selected later.) Due to the Deepest Path First criterion, this implies that $d(P_i) \geq d(P_j)$. Since P_j was active and was not deleted during the iteration in which P_i was selected, it must not contain $u(P_i)$. However, if P_j contained any nodes of P_i, then Lemma 1 would imply that P_j must contain $u(P_i)$, since it also contains a node $u(P_j)$ whose depth is at most $d(P_i)$. Because we know that P_j cannot contain $u(P_i)$, we may conclude that P_j does not contain any nodes of P_i, i.e. P_i and P_j are non-conflicting. $\square$

Lemma 3. *If DPF is the set of paths selected by the Deepest Path First Algorithm, and S is any other non-conflicting subset of the input paths, then $|S| \leq |DPF|$.*

Proof. Let $U = \{u(P_i) \mid P_i \in DPF\}$. The function $P_i \mapsto u(P_i)$ defines a one-to-one correspondence between DPF and U, so $|DPF| = |U|$. For every path $P_j \in S$, let P_i be the path selected by the Deepest Path First algorithm during the iteration in which P_j was either selected or deleted. Then $u(P_i)$ is a node of P_j. This means that every path $P_j \in S$ contains a node in U. No two distinct paths in S contain the same node in U, because the paths in S are non-conflicting. Hence, $|S| \leq |U| = |DPF|$, as claimed. $\square$

Finally we must work out the running time of the DPF algorithm. Let N denote the total length of all the paths $P_1, \ldots, P_m$. (Note that $N \leq mn$, but it is possible that N is much smaller than mn.) The algorithm can be implemented in $O(n + N + m \log m)$ time as follows. The first three lines of the pseudocode take $O(n)$, $O(N)$, and $O(m)$, respectively. In $O(N)$ time we can also run through all the paths and build a reverse index, mapping each node u of T to a linked list of all the indices j such that P_j contains u. In $O(m \log m)$ time we can sort the paths in decreasing order of $d(P_i)$. In each iteration of the main loop it takes $O(1)$ time to select the first path in the list, mark it as selected, and delete it from the list of active paths. Then we run through the list of all paths containing $u(P_i)$ (recall that we precomputed this list before starting the main loop) and delete each one from the list of active paths. This takes $O(1)$ time per deleted path. In total $O(N)$ paths are deleted in the course of executing the algorithm, so the total amount of work devoted to executing the third line of the main loop, summed over all iterations of that loop, is $O(N)$. Finally, outputting the set of selected paths takes $O(m)$.

Summing all of these running time bounds, we get an overall running time bound of

$$O\big(n + N + m + N + m \log m + n + N\big) = O(n + N + m \log m).$$