CSC373 Fall'20
Midterm 1 Solutions

Note 1: There were two variants of the midterm; let's call them versions A and B. They differed slightly in the details of Q1 and Q2. These differences only manifested in finer details; the key ideas behind the solutions were the same. For each of Q1 and Q2, I'll first describe the difference between the two variants, then provide the solution for version A, and then identify the minor differences in the solution for version B.

Note 2: Please note that the solutions below differ from the kind of solutions expected from you in the actual test in two significant ways.

1. The solutions below are quite detailed. This is to help you understand them without confusion. Your solutions in the actual test can be a lot more succinct.
2. The solutions below are written with the purpose of highlighting the *process* of arriving at them starting from the problem statement. For example, in Q1, I describe *how* to figure out that we need the recursive function to return some additional information. In the actual test, you can directly provide the algorithm and a justification for its correctness without explaining *how* you arrived at it.

Solution to Q1

Variants:

- A: The subgraph is allowed to be empty (empty subgraphs have total node weight 0).
- B: The subgraph is required to be non-empty.

Solution for version A:

For a node v, let $T(v)$ denote the subtree of node v; hence, the entire tree is $T = T(r)$, where r is the root node.

We will design a function, which, when applied on node v, finds the maximum weight of any connected subgraph of $T(v)$ (allowing the empty subgraph). To apply divide-and-conquer, we will call the function recursively on the children of v and use the solutions returned from these calls.

Note that a maximum-weight connected subgraph H of $T(v)$ can have one of two possibilities:

- H doesn't contain v. Then, H must be a maximum-weight connected subgraph of $T(v_i)$ for some child v_i of v. This is because if H contains parts of subtrees of two or more children, then H won't be connected.
- H contains v. Then, for each child v_i of v, H must either include no nodes from $T(v_i)$ or a maximum-weight connected subgraph of $T(v_i)$ *that contains v_i itself.* Once again, the condition of containing v_i itself is necessary and sufficient to make sure that H is connected.

Here, we realize that from the recursive call on a child v_i, we not only need the maximum weight of any connected subgraph of $T(v_i)$ (let's call this $A[v_i]$), but also the maximum weight of any connected subgraph of $T(v_i)$ that includes v_i (let's call this $B[v_i]$). We will compute both quantities in our recursive function simultaneously. Note that $B[v]$ is simply the maximum weight among subgraphs H in the second possibility above. Translating the description of the two possibilities above to code, we get the following algorithm:

Function $f(v)$:

- If v is a leaf, return $(A[v], B[v]) = (\max(0, w_v), w_v)$.
- $(A[v_i], B[v_i]) \leftarrow f(v_i)$ for each child v_i of v.
- $B[v] \leftarrow w_v + \sum_{v_i=\text{child of } v} \max(B[v_i], 0)$.
  // This is the 2nd possibility. The 0 inside max allows not including any node from $T(v_i)$.
- $A[v] \leftarrow \max(\max_{v_i=\text{child of } v} A[v_i], B[v])$.
  // The first term in outer max corresponds to the 1st possibility. $A[v]$ is the max of the two.

The desired solution is obtained by calling $f(r)$ (at root node r) and returning $A[r]$.

Running time: When we call $f(r)$, and let the function make recursive calls, we note that the function is called at each node v exactly once (from the call on its parent node).

The time spent in the call at node v is $O(d(v))$, where $d(v)$ is the degree of node v. Hence, the total running time is $O(n + \sum_v d(v)) = O(n + m) = O(n)$ because a tree has $m = n - 1$ edges.

Changes for version B:

We note that the 2nd possibility already enforces the subgraph H to be non-empty because node v itself is included. Hence, $B[v]$ remains unchanged. We only need to change the definition of $A[v]$ to the maximum weight of any *non-empty* connected subgraph of $T(v)$. Interestingly, this does not change the recurrence relation of $A[v]$ from the last line of function f (because now each $A[v_i]$ will also be forced to ignore the empty subgraph). The only change is in the base case of a leaf v, where we return $(A[v], B[v]) = (w_v, w_v)$.

Solution to Q2

Variants:

- A: The algorithm starts at a, maintains a covered region $[a, t]$, and constantly expands the region more to the right.
- B: The algorithm starts at b, maintains a covered region $[t, b]$, and constantly expands the region more to the left.

Solution for version A:

(a) Let I_r be the interval added by the greedy solution in the r-th iteration of the while loop. Let OPT be an optimal solution that contains intervals $I_1, \ldots, I_k$ for the maximum possible k. If $k = |G|$, then we must have $OPT = G$, so we are done. If $k < |G|$, we derive a contradiction by designing an optimal solution that contains intervals $I_1, \ldots, I_{k+1}$.

Note that $I_1, \ldots, I_k$ already cover the interval $[a, f_{I_k}]$. Now, OPT must contain at least one other interval $[s_i, f_i]$ such that $s_i \leq f_{I_k} < f_i$. Otherwise, points just to the right of f_{I_k} will not be covered. Note that this interval covers f_{I_k}, and since the greedy algorithm chooses the interval with the greatest endpoints among such intervals, we have $f_i \leq f_{I_{k+1}}$. Hence, $I_1 \cup \ldots \cup I_k \cup [s_i, f_i] \subseteq I_1 \cup \ldots \cup I_k \cup I_{k+1}$. Thus, OPT' obtained by replacing the interval $[s_i, f_i]$ from OPT with I_{k+1} is also an optimal solution for covering the entire interval $[a, b]$, which is the desired contradiction.

(b) The algorithm sorts the intervals in a non-descending order of the starting point, i.e., so that $s_1 \leq \ldots \leq s_n$. Then, it makes one pass over the list, maintaining i^*, the interval with the maximum f_i observed so far. When it encounters any $s_i > t$, it adds interval i^* to the solution, sets $t = f_{i^*}$, and resets (uninitializes) i^*.

Let t_r be the value of t at the end of the r-th iteration (with $t_0 = a$). Then, the key insight is this: In the $(r + 1)$-st iteration, we want to consider intervals that cover t_r, i.e., for which $s_i \leq t_r \leq f_i$. However, we only really need to consider those intervals for which $s_i \in (t_{r-1}, t_r]$. Any interval with $s_i \leq t_{r-1}$ will already be considered in a previous iteration, and since we always choose the greatest endpoint to include, it will not cover any additional point. On the other hand, some interval with $s_i \in (t_{r-1}, t_r]$ may not have $f_i \geq t_r$ (i.e. it may not actually cover t_r), but this will get ignored anyway since we will choose the interval with the maximum f_i (which will inevitably have $f_i > t_r$).

The sorting takes $O(n \log n)$ time and then the single pass takes $O(n)$ additional time. Hence, the algorithm runs in $O(n \log n)$ time.

Changes for version B:

The two versions are completely symmetric. To see this, visualize the intervals in front of you on a piece of paper and now imagine going to the opposite side of the paper. The algorithm from version A will now become the algorithm from version B. So if one is optimal, so is the other. In the solution above, you can correspondingly swap a and b, s_i-s and f_i-s, and max and min.

Solution to Q3

(a) Define an array $Q[1 \ldots n]$ where $Q[i]$ denotes the maximum number of pairwise friendly heads that we can chose from houses $1, \ldots, i$ subject to choosing the head of house i. The desired solution is then $\max(Q[1], \ldots, Q[n])$.

(b) The difficulty in writing a recurrence relation that describes $Q[i]$ in terms of $Q[j]$ for $j < i$ is that, while we can check if i is friendly with j, we have no idea what other heads are chosen as part of the solution of $Q[j]$, and therefore, cannot directly check if i is friendly with them too.
At this point, we notice the following non-trivial insight:

- For $k < j < i$, if heads i and j are friendly, and heads j and k are friendly, then heads i and k must be friendly too.

This is because if $|A[i] - A[j]| \leq i - j$ and $|A[j] - A[k]| \leq j - k$ (note: no absolute value on the right hand side because we know $i > j > k$), then by the triangle inequality,

$$|A[i] - A[k]| \leq |A[i] - A[j]| + |A[j] - A[k]| \leq (i - j) + (j - k) = i - k.$$

Once we realize this insight, we notice that for $j < i$, head i can be added to any friendly committee from $\{1, \ldots, j\}$ as long as i and j are friends (because j will already be friends with any head $k < j$ part of the committee, so i will be friends with k too). Hence, we get the following Bellman equation (the max over j returns 0 if there is no $j < i$ who is friendly with i):

$$Q[i] = \begin{cases} 1, & \text{if } i = 1, \\ 1 + \max_{j \in \{1,\ldots,i-1\} \wedge (i,j) \text{ are friendly}} Q[j], & \text{if } i > 1. \end{cases}$$

(c) Since each $Q[i]$ depends on $Q[j]$ for smaller j, we compute the entries in the order $i = 1, 2, \ldots, n$.

(d) Since each entry takes $O(n)$ time to compute (due to taking maximum over j) and $O(1)$ space to store, the time complexity is $O(n^2)$ and the space complexity is $O(n)$.