Massachusetts Institute of Technology — Handout 23
6.046J/18.410J: Introduction to Algorithms — April 07, 2002
Professors Michel Goemans and Piotr Indyk

Problem Set 4 Solutions

Problem 4-1. Planning a Party

This problem may be solved in linear time using dynamic programming. Let $c[x]$ denote the conviviality of employee x. We wish to select a subset of employees S maximizing $\sum_{x \in S} c[x]$, such that if $x \in S$, then $parent[x] \notin S$.

In order to use dynamic programming, we must be able compute the optimal solution of our problem in terms of optimal solutions to smaller subproblems of the same form. These optimal solutions to subproblems will be the following: let $M(x)$ denote the maximum possible conviviality sum if one were to invite only employees from the subtree rooted at employee x, such that x is invited. Similarly let $M'(x)$ denote the maximum conviviality sum for x's subtree if x is not invited. We can express $M(x)$ and $M'(x)$ recursively in terms of optimal solutions to "smaller" subproblems as follows:

$$\begin{aligned} M(x) &= c[x] + \sum_{y:parent[y]=x} M'(y) \\ M'(x) &= \sum_{y:parent[y]=x} \max\{M(y), M'(y)\} \end{aligned}$$

The first equation states that the optimal way to select employees (including x) from x's subtree is to optimally select employees from the subtree of each child y of x such that y is not invited. The second equation expresses the fact that the optimal way to select employees (not including x) from x's subtree is to optimally select employees from subtrees of children y of x, where it is of no consequence whether or not y is selected. The following pseudocode shows how we can compute $M(x)$ and $M'(x)$ for every employee x using a single recursive transversal of the company tree in $O(n)$ time:

```
Solve (x)
M(x) ← c[x]
M'(x) ← 0
y ← left-child[x]
while y ≠ NIL
    Solve(y)
    M(x) ← M(x) + M'(y)
    M'(x) ← M'(x) + max{M(y), M'(y)}
    y ← right-sibling[y]
```

The algorithm will be started by calling SOLVE(p), where p denotes the company president. Upon termination the optimal conviviality sum for the entire company will be given by $\max\{M(p), M'(p)\}$. How does one determine the set of employees to invite which achieves this maximum conviviality sum? This may be done in $O(n)$ time with another tree transversal, by calling INVITE(p), illustrated by the following pseudocode:

```
INVITE (x)
if M(x) > M'(x)
   then Invite x to the party
        for all grandchildren y of x
           INVITE(y)
   else for all children y of x
           INVITE(y)
```

The reasoning behind this reconstruction algorithm is the following: if $M(p) > M'(p)$, then the optimal solution must involve inviting the company president to the party. If this is the case, we then cannot invite any of the employees reporting immediately to the president, so we proceed to optimally invite employees which are grandchildren of the president in the company tree. If $M(p) \leq M'(p)$, then we do not need to invite the president, and we then proceed to optimally invite employees from the subtrees of employees directly reporting to the president. Continuing this process recursively, we will be able to produce a set of employees to invite whose conviviality sum is equal to that of the optimal solution, $\max\{M(p), M'(p)\}$.

Problem 4-2. Class Photo

To check if the row of students is unbalanced, we need to look at $O(n)$ windows of k consecutive students, and for each window we need to compare the maximum height within the window to the minimum height within the window. We will store the students in the window in a red-black tree (any balanced binary search tree will work), keyed by height. Initially, we must build a binary search tree on the first k students, which requires $O(k \log k)$ time.

We proceed to examine each k-length window in sequence, starting with students $1 \ldots k$. For each window location, we compare the maximum and minimum values in the binary search tree – this takes $O(\log k)$ time. We then shift the window over by one student, which requires the deletion of one student from the tree and the insertion of one new student into the tree – this also takes $O(\log k)$ time. Since we must example $O(n)$ window locations, and we spend $O(\log k)$ time in each location, the total running time will be $O(n \log k)$. Note that the time required to initially construct the tree, $O(k \log k)$, is dominated by $O(n \log k)$.

Another way to solve this problem is to use a pair of heaps instead of a balanced binary search tree. One heap will be a "min-heap" which keeps track of the running minimum of the sliding window, and the other will be a "max heap" which keeps track of the running maximum of the sliding window. This approach also yields a running time of $O(n \log k)$.

To solve part (b), we use some more sophisticated techniques to keep track of the minimum and maximum elements within the sliding window. We will proceed to describe a technique which will maintain the running minimum of our window in $O(1)$ amortized time per window position (the same technique will work for maintaining the running maximum). Within our window, we will color elements one of two colors: green or red, and we will maintain a pointer to the current minimum green element and the current minimum red element. Every green element will have a pointer to the smallest green element ahead of it in the window, so if that element were to be removed, one could correctly reset the pointer to the minimum green element in only $O(1)$ time. The minimum value in the window can be computed in $O(1)$ time, by comparing the minimum red and minimum green elements. When we add a new element to the front of the window, we color it red and potentially update the minimum red pointer – this also takes $O(1)$ time. As mentioned before, removing a green element requires only $O(1)$ time. The only difficulty lies with removing a red element. Whenever we attempt to do so, we will first execute a "recoloring" operation, where we scan over all the elements in the window, color them green, and initialize their "minimum green element ahead" pointers. Recoloring takes $O(k)$ time, but one can think of this as spending $O(1)$ time per element, and since every element is recolored only once during its lifetime in the window, we will spend $O(1)$ amortized time per window position.

Problem 4-3. Study Groups

This problem is solved by dynamic programming. We define optimal solutions to our DP subproblems in the following way: let $M(i, j)$ denote the maximum possible number of respected friendships if one considers only students $1 \ldots i$ and attempts to partition just these students into exactly j groups. We can express $M(i, j)$ recursively in terms of optimal solutions to smaller subproblems by noting that in order to optimally partition students $1 \ldots i$ into j groups, we must first optimally partition some prefix $1 \ldots l$ of the first i students into $j-1$ groups, and then add a single group consisting of students $l+1 \ldots i$. We therefore have

$$M(i, j) = \max_{2(j-1) \leq l \leq i-2} \{M(l, j-1) + T(l+1, j)\},$$

where $T(i, j)$ denotes the total number of friendship pairs within the range of students from $i \ldots j$. We can compute $T(i, j)$ for all $1 \leq i < j \leq n$ in $O(n^2)$ time as follows. First, scan over all pairs of friendships and with them build an $n \times n$ matrix F, such that $F(i, j)$ is one if students i and j are friends, and zero otherwise. This requires $O(n^2)$ time, since there are at most $O(n^2)$ friendship pairs to process. We will then compute $T(i, i+1)$ for all i, then $T(i, i+2)$ for all i, then $T(i, i+3)$ for all i, and so on. Each $T(i, j)$ is computed using the following straightforward formula:

$$T(i, j) = F(i, j) + T(i+1, j) + T(i, j-1) - T(i+1, j-1).$$

It will only take $O(1)$ time to evaluate this formula when computing $T(i, j)$, as we will already have computed $T(i+1, j)$, $T(i, j-1)$, and $T(i+1, j-1)$. Overall, it will therefore take $O(n^2)$ time to compute a table of all $T(i, j)$ values. After first computing all $T(i, j)$ values, we finish solving the main problem by computing all of the $M(i, j)$ values in sequence:

```
for j ← 1 to k
    for i ← 1 to n
        Compute M(i, j).
```

The total running time to solve all of these nk subproblems will be $O(n^2k)$, as it takes $O(n)$ time to compute each $M(i, j)$ value. Upon termination, $M(n, k)$ will contain the value of the optimal solution to the problem. One may reconstruct the actual groups within this optimal solution by using the standard technique of maintaining "backpointers", for each (i, j), to the subproblem which was the maximizing subproblem for $M(i, j)$.