15-451/651 Algorithm Design & Analysis, Spring 2026

Homework #2 Solutions

1. (*, dynamic product maintenance mod arbitrary M)

   (a) Show that given a tree where each node p is associated with a non-negative integer x_p, and a fixed modulus M, the product of each subtree mod M,

   $$\prod_{q \in \textsc{SubTree}(p)} x_q,$$

   for all nodes p, can be computed in $O(n)$ time.

   Solution: Store the product of the subtree mod M as state in node:

   $$DP[i] = \prod_{j : j \in \text{Subtree}(i)} v_j \mod M$$

   $DP[i] = (v[i] \times DP[i.l] \times DP[i.r]) \bmod M$ where l is the left child and r is the right child.

   (b) Using the DP state from the above part, or some other method of your choice, give a data structure that maintains the product of a set of n non-negative integers mod M under modifications in $O(\log n)$ time per update, and only performs multiplications modulo M on non-negative integers of value at most $O(n^{10}M^{10})$.

   Note that M may not be prime, and the intended solution does not use any number theory beyond the fact that $(a \cdot b) \equiv ((a \mod M) \cdot b) \equiv ((a \mod M) \cdot (b \mod M)) \pmod{M}$.

   Solution: Take a complete balanced binary tree with leaves set to $x_1, \ldots, x_n$ in order, and maintain the above value for the product of all leaves in each subtree.

   Each internal node is product[left] * product[right] mod M. We can make modifications in $O(\log n)$ by updating the path to the root, and can query the root which is the product of all the numbers in $O(1)$.

2. (** weighted k-independent set) Give an algorithm that takes a tree with weights on the vertices, returns the maximum weight of a subset of exactly k vertices such that no two vertices in the set are adjacent, in time $O(n^{10}k^{10})$.

   Solution: DP state is $DP[i][k][0]$ is the max weight of a subset of k independent elements in subtree of i without root used, $DP[i][k][1]$ is the max weight where the root is used.

   Transitions:

   Let the children of i be $1 \ldots d$. We want to distribute k between whether i is in the set, and how many from subtree j is in the set, k_j.

   If root is not in the set, then each child can be in or out of the set

   $$DP[i][k][0] = \max_{k_1 + \ldots k_d = k} \sum_{j \in \text{Children}(i)} \max\left\{DP[j][k_j][0], DP[j][k_j][1]\right\}$$

   If root is used, each children must be unused,

   $$DP[i][k][1] = w_i + \max_{k_1 + \ldots k_d = k-1} \sum_{j \in \text{Children}(i)} DP[j][k_j][0]$$

   Base case: $DP[i][0][1] = 0$, $DP[i][1][1] = w_i$.

This DP as given takes time $O(nk^{d_{\max}})$, where $d_{\max}$ is the maximum number of children of a node: the transitions involve d numbers, $k_1 \ldots k_d$, each can be up to k.

To make it faster, define DP states on the children prefix DPpartial$[i][i1][k][0]$ for $0 \le i1 \le |\text{Children}(i)|$ to be the maximum weight in the tree induced by i and its first $i1$ children, using k nodes excluding i. The base case is then

$$\text{DPpartial}[i][i1][k][0] = \begin{cases} 0 & \text{if } k = 0 \\ -\infty & \text{otherwise} \end{cases}$$

and the transition (for the children list Children(i)=$< j_1 \ldots j_d >$) is

$$\text{DPpartial}[i][i1][k][0] = \max_{0 \le k1 \le k} \text{DPpartial}[i][i1-1][k-k1][0] + \max\left\{DP[j_{i1}][k1][0], DP[j_{i1}][k1][1]\right\}$$

for each $i1 > 0$. Once we compute this, we can let $DP[i][k][0] = \text{DPpartial}[i][d][k][0]$ for each k.

Similarly, for the case where we do take i, define DPpartial$[i][i1][k][1]$ for $0 \le i1 \le |\text{Children}(i)|$ to be the maximum weight in the tree induced by i and its first $i1$ children, using k nodes including i. The base case is

$$\text{DPpartial}[i][i1][k][1] = \begin{cases} w_i & \text{if } k = 1 \\ -\infty & \text{otherwise} \end{cases}$$

and the transition is

$$\text{DPpartial}[i][i1][k][1] = \max_{0 \le k1 \le k} \text{DPpartial}[i][i1-1][k-k1][1] + DP[j_{i1}][k1][0]$$

for each $i1 > 0$, and when done we set $DP[i][k][1] = \text{DPpartial}[i][d][k][1]$.

For running time, the most immediate bound is to look at all dimensions:

(a) there are at most n nodes,

(b) each has at most n children,

(c) the number of things taken is up to k,

(d) the number of values of k_1 is k,

(e) and taken/not taken is 2 states,

so $O(n^2k^2)$.

One can get to $O(nk^2)$ by observing that the total nubmer of children of all nodes is $O(n)$. Another way to see this is to count things in the other direction: each node has at most 1 parent.

UNESSRY tigher bounds

We can actually prove a tighter bound of $O(nk)$...

First note that it's actually bounded by the smaller size of a child subtree:

$$\min\left\{k, \text{Size}\left[p\right]\right\} \cdot \min\left\{k, \text{Size}\left[q_1\right], \text{Size}\left[q_2\right]\right\}.$$

We can actually prove that this total cost in a tree is smaller.

Lemma 1. *In a binary tree with n nodes, for any value k, the total of the value in Equation 2 summed over all nodes is $O(nk)$.*

Proof. We will prove this in two steps, we first handle the case where $\text{SIZE}[p] \le k$. This implies that the size of both children of p are also at most k as well.

Note that whenever $x = x_1 + x_2$ and $x_1 \le x_2$, we get via $x_1 \le x/2 \le x_2$:

$$x_1^2 + x_2^2 + x_1 \cdot x \le x_1^2 + x_2^2 + 2x_1x_2 = (x_1 + x_2)^2 = x^2.$$

for any $y \leq x/2$. So we get that as long as $\text{SIZE}[p] \leq O(k)$, the cost is at most $O(k \cdot \text{SIZE}[p])$. So we can prove by induction on the value x that a node x incurs a cost of at most x^2.

For the case where the size of p is more than k, observe that there are at most $O(n/k)$ nodes with both children having size at most k. So the total cost among such nodes is at least

$$O(n/k) \cdot O\left(k^2\right) = O(nk).$$

Then the remaining case is that one of the children has size $< k$, but p has size more than k. For this case, observe that we can charge a cost of k to all nodes in the smaller subtree. Such nodes are never charged again, because in that subtree there are no more nodes of size more than k. So once again we get a contribution of $O(nk)$. □

On additional trick that can be done to this problem is that we can reduce the total memory usage to $O(\log n \cdot k)$. We always recurse onto the child with bigger size first, and use tail recursion to directly pass up the size k knapsack table. This ensures that the only things we need to keep 'on the stack' are the $O(\log n)$ ancestors with succesisvely doubling sizes.

3. (** pareto optimum points) Give an algorithm that takes a length n array of 2-tuples (a_i, b_i) and computes for **each** i whether there is some $j < i$ such that i is larger than j in both attributes, aka. $a_i > a_j$ and $b_i > b_j$, in a total time of $O(n \log n)$.

   Solution: First, use sort to reduce all key values to the range $1 \ldots n$.

   Build a tree on an array $B[1 \ldots n]$ that supports:

   - Modify $B[i]$
   - Query for prefix min, the minimum of $B[1 \ldots i]$ for some i.

   Loop through the items in increasing order, for each i, query for the minimum b in the tree where $a < a_i$, by querying for the minimum of

   $$B[1 \ldots (a_i - 1)].$$

   If it's less than a_i, then a_i has such a j.

   Then update $B[a_i]$ in the tree with b_i.

   This is $O(n)$ update/query operations, giving a total of $O(n \log n)$.