Matthew Feng November 7, 2018

6.046 Problem Set 8

Collaborators: *James Lin*

Problem 1

Subproblems. Let $D(i, j)$ denote the minimum number of uses of the machine such that using only the first i boxes, Ben can shrink or grow his boxes in the way described in the problem such that the volumes of those first i boxes sums to j.

These subproblems have optimal substructure because with $D(i, j) = d^*$ optimal, if there exists a more optimal solution d' to any subproblem $D(i-1, j-v_i)$ (where v_i may be any legal volume of box i), then we may simply use that solution instead, achieving a lower value for d^*, violating the notion that d^* was optimal. Thus no such subsolution d' can exist.

Relate. WLOG, order the dimensions of the boxes such that $x_i \leq y_i \leq z_i, \forall i \in [1, n]$. Our subproblems are related by the following recurrence:

$$D(i, j) = \min_{x \in [1, V^{1/3}]} \{D(i-1, j-(x)(x+y_i-x_i)(x+z_i-x_i)) + |x_i - x|\}$$

with base cases

$$\begin{aligned} D(0, 0) &= 0, \\ D(i, 0) &= \textbf{None}, \ \forall i > 0, \\ D(i, j) &= \textbf{None}, \ \forall j < 0, \end{aligned}$$

where the min ignores **None** or outputs **None** if it takes no arguments (i.e. everything it is trying to min over is also **None**).

In other words, $D(i, j)$ looks at all possible ways to modify box i, and picks the method with the minimum uses of the machine, provided such a method exists. We are able to bound the number of modifications we make to box i because we are targeting a specific volume V, and so the smallest dimension does not need to exceed $V^{1/3}$.

DAG. Since the volume j decreases with every recursive call, and i decreases as well, every subproblem only depends on smaller subproblems, and thus the dependency graph is a DAG. This means that the problems can be solved bottom-up in an efficient manner using memoization.

Evaluate. To solve the original problem, we need to compute the value of $D(n, V)$.

Analyze. The running time of this algorithm can be computed using the formula

$$(\text{number of subproblems}) \times (\text{cost per subproblem}).$$

There are $O(nV)$ subproblems, and each subproblem looks at $O(V^{1/3})$ sub-subproblems, and so the total runtime is $O(nV^{4/3})$.

Problem 2

Subproblems. Let us denote the set $C = \{c_i\}$, $\forall i \in [1, n]$ be the set of all colors we may use (i.e. the colors for each of the different Martian colonies), and let $A[1:i]$ (inclusive, one-indexed) be the current coloring of the array, where $A[i] \in C$ denotes the color at index i. Let $DP[1:i]$ represent the optimal coloring, with $DP[i]$ denoting the optimal color at index i.

Let $D(i, c)$ denote the minimum cost needed to create peace on Mars for subarray $A[1:i]$, under the condition that $DP[i] = c$. Each $D(i, c)$ will be a subproblem in our dynamic program.

Now, we need to demonstrate that our subproblems have optimal substructure. Suppose then, that for subproblem $D(i, c)$ with optimal cost $d^* = D(i, c)$, there was some coloring with cost $D'(i-1, c')$ that was lower than $D(i-1, c')$. Then we can simply substitute $D'(i-1, c')$ for $D(i-1, c')$ to obtain a cost $d' < d^*$, which meant that $D(i, c) = d^*$ was never optimal. Thus, the optimal coloring must have optimal subcolorings as well.

Relate. We have the following recurrence relation:

$$D(i, c) = \min_{c' \in C} \left(D(i-1, c') + a\mathbb{1}_{c \neq c'} + b\mathbb{1}_{c \neq A[i]} \right),$$

where $\mathbb{1}_*$ is the indicator random variable for $*$, i.e. 1 if $*$ is true, and 0 otherwise.

We also have the following base cases:

$$D(1, c) = 0, \forall c \in C,$$

i.e. for a single cell, there is no cost for peace.

In other words, we guess that the optimal color of the last cell is c; based on that guess, we want to find the minimum cost of the remaining $i-1$ cells, trying all colors for the $(i-1)$th cell. If the ith and $(i-1)$th cells differ in color, we must build a wall; likewise, if we need to change the color of the ith cell, we need to add that cost as well. We don't need to consider the cost of changing the color of the $(i-1)$th cell, as it will be covered in the recursive call.

DAG. Our recurrence is a directed acyclic graph (which allows for efficient computation) because each subproblem $D(i, c)$ only depends on subproblems with a smaller index $i-1$.

Evaluate. Our original problem becomes finding the value of

$$\min_{c \in C} D(n, c)$$

as the final color may be any of the choices in C.

Analyze. The number of subproblems in our dynamic program is $O(nm)$ since a subproblem exists for each (index, color) pair. Each subproblem requires $O(m)$ time to solve, as it needs to iterate over the m different colors in C. Overall, then, our algorithm has a runtime of (no. of subproblems) $\times$ (cost per subproblem) which equals $O(nm) \times O(m) = O(nm^2)$, as desired.

Problem 3

Subproblems. Let F_v denote the coefficient of fun for employee v, and let $v.c$ denote the set of children of v, i.e. the nodes u whose boss $\pi(u) = v$. Let $D(v)$ denote the maximum sum of the cofficients of fun (we will use the terminology "maximum fun" from now on) for the subtree rooted at v (including v), subject to the constraint that if employee x is invited, none of $y \in x.c$ are invited, and likewise that x is not invited if any of $y \in x.c$ are invited.

Suppose that the our subproblems didn't have optimal substructure. That means that although $D(v) = f_v^*$, the maximum fun possible, that one of its subproblems isn't optimal. That means there is some subtree of v rooted at u such that $D(u) < f_u^*$, where f_u^* is the maximum fun of the subtree rooted at u. But this would be that we could invite the employees whose coefficients constitute the optimal sum f_u^* and have a fun greater than f_v^*, which is a contradiction. Thus, the fun for each of the subproblems must also be optimal for $D(v)$ to be optimal.

Relate. Our subproblems are related by the following recurrence:

$$D(v) = \max \left\{ F_v + \sum_{u \in v.c} \sum_{w \in u.c} D(w), \sum_{u \in v.c} D(u) \right\}$$

with base case $D(v) = F_v$ if v is a leaf.

Essentially, the recurrence states that the maximum fun for the tree rooted at v has two cases: whether v is invited or not. If v is invited, we account for v's fun, and then sum over the maximum fun for each of v's grandchildren (since the children cannot be invited). On the other hand, if v is not invited, then we can invite v's children, so we take the sum of the maximum fun for each of v's children.

DAG. The subproblem dependency graph forms a DAG because the value of $D(v)$ depends only on its children and grandchildren, never an ancestor. Thus, these subproblems are able to be computed efficiently via memoization.

Evaluate. We can find the maximum fun possible by computing $D(b)$, where b represents Gill Bates. We can compute $D(b)$ with either a top-down or a bottom-up approach (i.e. start from leaves vs. start from root).

Analyze. For each of the subproblems $D(v)$, the cost of that subproblem is

$$|v.c| + \sum_{u \in v.c} |u.c|,$$

i.e. the sum of the number of children and number of grandchildren of v. That means the total cost of the algorithm is

$$\sum_{v} (|v.c| + \sum_{u \in v.c} |u.c|).$$

Since each node can only be the child/grandchild of a single other node, both terms of the outer sum is bounded $O(n)$. Thus the total runtime of the algorithm is also $O(n)$.