CS1510 Dynamic Programming Problems 24,& 25, Reduction Problem 2

Rebecca Negley, Sean Myers

October 14, 2011

24. Give a polynomial time algorithm for the following problem. The input consists of a sequence $R = R_0, ..., R_n$ of non negative integers, and an integer k. The number R_i represents the number of users requesting some particular piece of information at time i (say from a www.server. If the server broadcasts this information at some time t, the requests of all the users who requested the information strictly before time t are satisfied. The server can broadcast this information at most k times. The goal is to pick the k times to broadcast in order to minimize the total time (over all requests) that requests/users have to wait in order to have their requests satisfied

    Solution: The first thing we need to do is set up a tree. Each node is going to have k branches. Each branch is a decision of which group that is going to be chosen to send this data out. For example, if there are 3 groups, then the first node can either be in sending group 1, 2, or 3. The leaves are going to be all possible combinations of different groups; some will be possible sending sequences, others not but the solution will be there nevertheless. First, we must prune the non-possible trees.

    The non-possible trees will be where if the level you are currently at is attempting to add a node to group j where group $j + 1$ has a node, R_i in it. Prune any tree that attempts to do that. The reason for this is because if there is a node in group $j + 1$, then that means that group i must have already sent their data. If you want to delay it by adding the next node, then any optimal solution from this branch is impossible since adding that to group $j + 1$, would be less costly.

    Another pruning rule is if the total sum of all the times of all the groups in a current node are less than another on the same branch, take the least whilst pruning the others. If node v has a total service time of t_v which is strictly less than node u's t_u, and u somehow has an optimal solution, such that: $t_u + ... + t_n$, then $t_v + ... + t_n$ is going to be even less that, making it the optimal solution.

    With these pruning rules, we can create an algorithm! First, we must fill an array: server[lvl, k, n]. k is the group. We store the optimal time. Initialize this array to infinity, then set server[0,0, 0] = 0.

    We will go through each level, filling out the tree as best as we can. At each level, we take the current index, and set the index of the next level like so: server[lvl+1, i,n+1] = min(server[lvl+1,i,n+1], server[lvl,i,n+1]+server[lvl,i,n+1] + users[lvl+1]). You are either taking the current thing already there or adding the time that is already on the current level, with itself (because you are multiplying all the current user requests by the depth level), and then adding the current user requests of that level. You do this for every k of that level, then move on to the next level. You also set the choice if you don't choose that group: server[lvl+1, i , n] = min(server[lvl+1, i, n], server[lvl, i , n])

    Now that we have that, now let's show it in pseudocode!:

```
initialize server[n, k, n] = to infinity
server[0,0,0] = 0

for(lvl = 0 to n):
```

```
for(i = 0 to k):
    for(j = 0 to lvl):
        server[lvl+1, i, j+1] = min(server[lvl+1,i,j+1], server[lvl,i,j+1]+server[lvl,i,j+1] + users[lvl+1])
        server[lvl+1, i , j] = min( server[lvl+1, i, j], server[lvl, i , j])
```

To get the answer, you just need to backtrack on the last level and server[lvl][k-1][n] is the correct end node for the last k (since the optimal will include the last end node)

25. Assume that you are given a collection $B_1, ..., B_n$ of boxes. You are told that the weight in kilograms of each box is an integer between 1 and some constant L, inclusive. However you do not know the specific weight of any box, and you do not know the specific value of L. You are also given a pan balance. A pan balance functions in the following manner. You can give the pan balance any two disjoint sub-collections, say S_1 and S_2 of the boxes. Let $|S_1|$ and $|S_2|$ be the cumulative weight of the boxes in S_1 and S_2, respectively. The pan balance then determines whether $|S_1| < |S_2|$, $|S_1| = |S_2|$, or $|S_1| > |S_2|$. You have nothing else at your disposal other than these n boxes and the pan balance. The problem is to determine if one can partition the boxes into two disjoint sub-collections of equal weight. Give an algorithm for this problem that makes at most $O(n^2L)$ uses of the pan balance. For partial credit, find an algorithm where the number of uses is polynomial in n and L.

Solution: We will first construct a tree to give us all possible subsets of the boxes $\{B_1, \dots, B_n\}$. At each level i, every node will have two children, one including B_i and one excluding B_i. Then, all the subsets of boxes are at the leaves of the tree. To solve the above problem, we seek a subset with equal weight to its complement. Since there are 2^n possible subsets, we must prune the tree as follows:

(a) If a node represents a subset with a weight greater than the weight of its complement, prune it. (Indeed, we can only add boxes to our subsets as we traverse down the tree, so no child of this subset can ever equal its complement.)

(b) If two subsets at the same level have the same weight, prune one of them. (The same elements remain in the lower levels of the tree, so we can take the same elements from the complements to give the same weights as we go down the tree.)

We now must create an algorithm to construct this tree in only $O(n^2L)$ time (or uses of the balance). We will build our dynamic programming algorithm from the tree described above. We will use an $(n+1)$-sized array of lists. Each list will be at most lengh $nL/2$ (the max weight that a subsequence can have without being heavier than its complement). Each item in the list will be a subset of boxes. (We must store the subset since we cannot store the weight.) We initialize the first list in our array (at index 0) to contain only the empty set. Then, we can loop through the levels of our tree and carefully construct each level from the level above. By our construction, each list will be sorted in order of weight. This will be important for creating the next list. The procedure to do so is as follows: Call the subsets from the current level $S_1, \dots, S_k$ for some $k \leq nL/2$. Add S_1 to the list for the next level. Then create two indices $i = 1$ and $j = 2$. If adding the new box (corresponding with the current level) to S_i produces a lesser weight than $|S_j|$, add $S'_i = S_i \cup \{B_{new}\}$ to your list for the next level and increment i. If the weights are equal, add one of $S_i \cup \{B_{new}\}$ and S_j and discard the other (choose arbitrarily), incrementing both i and j. Otherwise, add S_j to your list for the next level and increment j. Since $|S_p \cup \{B_{new}\}| > |S_p|$ for all p, we know $i \leq j$. When j reaches the end of the list for the previous level, continue adding $S_i \cup \{B_{new}\}$ to the list for the new level and incrementing i until $S_i \cup \{B_{new}\}$ weighs more than its complement or i reaches the end of the list for the previous level. This way, we construct a list of all possible subsets at that level that are lighter than their complements sorted from lightest to heaviest. Since duplicate weights are removed, the list is at most length $nL/2$. Since we used the balance once for every subsequence in the list, we use the balance at most $nL/2$ times each level. There are n levels of the tree, so we use the balance $O(n^2L)$ times. To find out if there is a satisfying subsequence, we simply need to look at the last item in the last list. The last item will be heavier than all other subsequences in the last row (which will weigh strictly less than their complements), so any subsequence that equals its complement would be at that location. In code, our algorithm looks like this:

```
EWB($B_1$,...,$B_n$)
    EWB[n+1]
    EWB[0].add($\emptyset$)
    for $lvl$ from 1 to $n$
        EWB[$lvl$].add(EWB[$lvl-1$][1]
        $i = 1$
        $j = 2$
        while $j \leq$EWB[$lvl-1$].length:
            if |EWB[$lvl-1$][$j$]| > |EWB[$lvl-1$][$i$] $\cup \{B_{lvl}\}$|:
                EWB[$lvl$].add(EWB[$lvl-1$][$i$] $\cup \{B_{lvl}\}$)
                $i++$
            else if |EWB[$lvl-1$][$j$]| = |EWB[$lvl-1$][$i$] $\cup \{B_{lvl}\}$|:
                EWB[$lvl$].add(EWB[$lvl-1$][$j$])
                $i++$, $j++$
            else
                EWB[$lvl$].add(EWB[$lvl-1$][$j$])
                $j++$
        while $i \leq$ EWB[$lvl-1$].length AND EWB[$lvl-1$][$i$] $\cup \{B_{lvl}\}$| $\leq$ |$\overline{\text{EWB}[lvl-1][i] \cup \{B_{lvl}\}}$|:
            EWB[$lvl$].add(EWB[$lvl-1$][$i$] $\cup \{B_{lvl}\}$)
            $i++$
    if EWB[$n$][EWB[$n$].length]= $\overline{\text{EWB}[n][\text{EWB}[n].\text{length}]}$:
        return true
    else
        return false
```

26. Show that if there is an $O(n^k)$, $k \geq 2$, time algorithm for inverting a nonsingular n by n matrix C then there is an $O(n^k)$ time algorithm for multiply two arbitrary n by n matrices A and B.
For a square matrix A, A inverse, denoted A^{-1}, is the unique matrix such that $AA^{-1} = I$, where I is the identity matrix with 1's on the main diagonal and 0's every place else. Not that not every square matrix has an inverse, e.g. the all zero matrix.

Solution: If we can reduce multiplying two arbitrary n x n matrices to inverting a matrix, and the most inefficient part of the algorithm is the inversion, then the algorithm will run in at least $O(n^k)$.

If we set up a matrix C, where the matrix of size 3n x 3n, where the matrix looks like: $\begin{vmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{vmatrix}$

(I being identity matrix). The setup of this matrix would take $9 * n^2$, since it is trivial to set up an identity matrix, copy a matrix or 0-pad a matrix like so.

Then the inverse(derivation not shown) would look something like: $\begin{vmatrix} I & -A & A*B \\ 0 & I & -B \\ 0 & 0 & I \end{vmatrix}$

Then all we would need to do is extrapolate the top right node (which takes n^2 amount of time), and we have our matrix multiplication.

The time to convert to the input matrix C is $O(n^2)$, the inversion takes $O(n^k)$ where k must be greater than or equal to 2 and then once we have the output, the time to translate to the desired output is $O(n^2)$. Hence, the algorithm's slowest possible run time is that of the inverse multiplication $O(n^k)$.