# CS1510 Dynamic Programming Problems 23 & 26, Reduction Problem 1

Rebecca Negley, Sean Myers

October 12, 2011

23. Consider the problem where the input is a collection of $n$ train trips with Germany. For the $i$th trip $T_i$, you are given the date $d_i$ and the non-discounted fare $f_i$ for that trip. The German railway systems sells a Bahncard for $B$ Marks that entitles you to a 50% fare reduction on all travel within Germany within $L$ days of purchase. The problem is to determine when to buy a bahncard to minimize the total cost of your travel. Give a polynomial time algorithm for this problem. The running time should be independent of $B$ and $L$.

    Solution: To start off this problem, we want to create a tree! Each level is deciding: do we add this trip to the "discount trips" node we are creating? Each node will have to hold the total accumulated fare of the non-discounted price. The leafs will contain the solution, which is the max fare possible will be the one we want to reduce. We need to keep track of which trip we start at, for it is the key to most of the pruning of this tree.

    The first tree rule (and pruning rule), is if the first trip's date $d_i$ and the date of the trip at the level we are on $d_j$ is such that $d_j - d_i > L$, prune it. Before, our tree had nodes that weren't possibly discount trips, for to be a discount trip you need to have only trips within the bounds of $L$. This makes sure that all leaves at the bottom are only plausible solutions.

    The other pruning rule is: If two trips on the same level have the same starting node, and have already been pruned by the first rule, keep the one with the most fare. Implicitly, if two nodes at a branch start with the same node, the only possible way for this to happen is if one has an additional trip on it. To prove that this pruning is wise, let's say we have two nodes $m$ and $n$. $m$ has trips $t_1, ...t_k...t_j$ whilst $n$ has every trip as $m$ except $t_k$, which means $fare(n) < fare(m)$. Let's say $n$ is the node that is the best choice for a discount. That means if there are some trips after: $fare(n) + f_{j+1}... + f_{j+l}$, then it is possible for $m$ to have $fare(m) + f_{j+1}... + f_{j+l}$, and since $m$ has a higher fare than $n$, it is the maximum fare, making it the best solution.

    Now that we have our pruning rules, let's start building an algorithm. First, we need our array. We know that there are n levels. We also know that there are n dates which with we can start, since each trip can have a different date. This leaves us with an array of: train[n,n].In each index, we will have maximum fare.

    First we initialize the array all the fares to 0, since we are looking to find the max fare of the trips. Then, we travel level by level, using the current level and adding to each. Let's assume the dates are in days from the first trip. For example, date[1] should be 0, whilst date[2] has a value of 4, meaning it is 4 days after trip 1.

    If the date of the next level minus that of the current trip we are looking at is greater than $L$, then don't do anything. Otherwise, it will be train[lvl+1, i] = max(train[lvl+1,i], train[lvl,i]+fare[lvl+1]) which means that either the current node has a larger, better fare, or adding the current fare to it is better (refer to second pruning rule).

The answer will be in the nth level of train array, and will be the max value of that row. If $best/2 < B$, do not buy the Bahncard. What this means is that the max fare isn't worth buying if the price of the Bahncard is higher than the total savings. Otherwise, buy the best ticket using the index as so date[best].

The final algorithm looks something like this:

```
initialize train[n,n] = 0

for(lvl = 0 to n):
    for(sDate = 1 to lvl):
        train[lvl+1, sDate] = max(train[lvl+1, sDate], train[lvl, sDate])
        if(date[lvl+1]-date[sDate] ≤ L):
          train[lvl+1, sDate] = max(train[lvl+1,sDate], train[lvl, sDate]+fare[lvl+1])
```

Just an addendum, the second for loop ends at lvl, because we know that we can't start adding fares of trips that haven't even started.

26. Give an algorithm that takes a positive integer $n$ as input and computes the number of possible orderings of $n$ objects under the relations $<$ and $=$. You algorithm should run in time polynomial in $n$.

Solution: Consider an $n$ level tree. At each level, an object is added to the ordering at each node. Each node has a variable number of children. Say a node has $p$ unique elements in its ordering. (Example: $a < b = c < d$ would have three unique elements.) Then, the new element can be equal to any one of the $p$ elements, less than all of them, greater than all of them, or in between two of them. If the new element is equal to a previous element, the child node will have the same number of unique elements as the parent node. Otherwise, the child must have one more unique element. Therefore, a node with $p$ unique elements will have $p$ children with the same number of unique elements and $1+1+(p-1) = p+1$ children with one more unique element.

Observe that we cannot prune this tree. Every node contains a unique ordering. Suppose not. The first level with only one object (and, hence, one ordering) is trivially unique. Let $k$ be the shallowest level with two separate nodes having an identical ordering. Call these nodes $A$ and $B$. By the way we defined each node's children, two children of the same node cannot have an identical ordering. Then, $A$ and $B$ must have different parents. However, the first $(k-1)$ elements must have the same ordering in $A$ and $B$ in order for the first $k$ elements to have the same ordering. This gives that $A$ and $B$ have different (from above) but identical parents, a contradiction of $k$ being the shallowest level of the tree with two separate identical nodes. Hence, each node is unique.

Since we know we do not need to prune and do not need to return the actual orderings, we do not need to keep track of what is actually at each node. We simply need to know the number of nodes with $p$ unique elements at some level $i$ for some integer $p \leq i$. We can keep track of this information in a $n \times n$ table. Each row of the table corresponds to a level of the tree. The columns corresponds to the number of unique elements in an ordering. In each location, we store the number of orderings at the current level with the corresponding number of unique elements. We can generate the information from the current row for the next row. If some row has $m$ orderings corresponding to $q$ unique elements, we add $q * m$ to the $q$th column of the next row and add $(q+1) * m$ to the $(q+1)$st column of the next row. (We do this because each ordering generates $q$ more elements of the same unique length and $(q+1)$ elements of one longer length.) Then, when we fill in our table, we sum all of the values in the last row to get the total number of orderings of $n$ elements. We initialize our table by setting the first location in the first row to 1 and the rest to 0. In code, it looks like this:

NumOrderings($n$)
initialize $n \times n$ matrix nO to 0
nO[1,1]=1
for $i$ from 1 to $(n-1)$ do:
for $j$ from 1 to $(n-1)$ do:
$\text{nO}[i+1, j] += j * nO[i, j]$
$\text{nO}[i+1, j+1] += (j+1) * nO[i, j]$
return $\sum_{j=1}^{n} \text{nO}[n, j]$

This algorithm runs in $O(n^2)$ time.

1. A square matrix $M$ is lower triangular if each entry above the main diagonal is zero. That is, each entry $M_{i,j}$, with $i < j$, is equal to zero. Show that if there is an $O(n^2)$ time algorithm for multiplying two $n$ by $n$ lower triangular matrices, then there is an $O(n^2)$ time algorithm for multiplying two arbitrary $n$ by $n$ matrices.

Solution: Suppose we want to multiply two $n \times n$ matrices $A$ and $B$ (in that order). Construct a $3n \times 3n$ matrix $A'$ such that $A'[2n+i, n+j] = A[i.j]$ for $1 \leq i, j \leq n$ and $A'[r, c] = 0$ elsewhere. Similarly, construct a $3n \times 3n$ matrix $B'$ such that $B'[n+i, j] = B[i, j]$ for $1 \leq i, j \leq n$ and $B'[r, c] = 0$ elsewhere. So we have:

$$A' = \begin{array}{c|c|c} 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & A & 0 \end{array} \text{ and } B' = \begin{array}{c|c|c} 0 & 0 & 0 \\ \hline B & 0 & 0 \\ \hline 0 & 0 & 0 \end{array}.$$

Observe that both $A'$ and $B'$ are then lower triangular. We can construct each in $(3n)^2 = O(n^2)$ time. If there is an $O(n^2)$ algorithm for multiplying two $n$ by $n$ lower trinagular matrices, then we can multiply $A'$ by $B'$ in $O((3n)^2) = O(n^2)$ time. We can see that

$$A'B' = \begin{array}{c|c|c} 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline AB & 0 & 0 \end{array}.$$

To retrieve our desired matrix, $AB$, we can copy the $n^2$ elements from the lower left corner of $A'B'$ into a new $n \times n$ matrix. This also takes $n^2$ time, so the whole algorithm is $O(n^2)$. Here is the above explanation in pseudocode:

MatMult($A$,$B$,$n$)

$A' = \begin{array}{c|c|c} 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & A & 0 \end{array}$ $//O(n^2)$

$B' = \begin{array}{c|c|c} 0 & 0 & 0 \\ \hline B & 0 & 0 \\ \hline 0 & 0 & 0 \end{array}$ $//O(n^2)$

$A'B'$ =LowerDiagMult($A'$,$B'$) $//O(n^2)$
for $i$ from 1 to $n$ do:
for $j$ from 1 to $n$ do:
$AB[i, j] = A'B'[2n+1, j]$ $//O(n^2)$
return $AB$