CS1510 Dynamic Programming Problems 10 & 11

Rebecca Negley, Sean Myers

September 28, 2011

12. Consider the code for the knapsack program given in the class notes.

    (a) Explain how one can actually find the highest valued subset of objects, subject to the weight constraint, from the Value table computed by this code.

    Solution: Look at the last row and last column calculated first. A nice recursive way to add to the knapsack is take the node you are at (the first one you input was the last row and last column), and if the row above you is the same as the current node, you do not add the current row's denomination, and traverse the table up one row. If the value above the current row is different, then you added the value at this node, which means you traverse [currentK-1][currentS-w(k)] and add K to your set of knapsack values. You do this until you reach [0,0]

    (b) Explain how to solve the Knapsack problem using only $O(L)$ memory / space and $O(nL)$ time. You need only find the value and weight of the optimal solution, not the actual collection of objects.

    Solution: To do this, instead of having an array of size [n][L], have an array of size [2][L]. The only thing in the original code that would need to be changed is every time you reference k, you must mod it by 2. The only information that is needed in calculation is the current computed row and the row we are computing next. Though this removes the informative information about the table, like backtracking to find the original set of objects, it gives us more memory in return. The code looks like this:

    ```
    Value[0,0]=0
    For S=1 to L do Sum[0, S]= -1

    For k=1 to n do
        For S = 0 to L do
            Value[k%2, S] = Max( Value[(k-1)%2, S], Value[(k-1)%2, S - w(k)] + v(k))
    ```

14. Give an algorithm for the following problem whose running time is polynomial in $n + L$.
    Input: positive integers $v_1, ..., v_n$ with

$$L = \sum_{i=1}^{n} v_i$$

    Output: A solution (if one exists) to

$$\sum_{i=1}^{n} (-1)^{x_i} * v_i = 0$$

    where each x_i is either 0 or 1.

Solution: This problem is creatively the same exact problem as the subset sum problem. For if there is a solution to where the sum of positive and negative values are 0, then that means it would take half of L being positive and half of L being multiplied by a negative for the solution to be 0. This is equivalent to taking the sum solution of $L/2$. If that solution exists, then any value not taken in the solution, would be multiplied by negative x_i in this solution. Hence, the solution is as follows:

Pruning Rules:

(a) Eliminate the subtree rooted at any subset with sum greater than $L/2$.

(b) If there are two subsets S and T at the same depth, with the same aggregate sum, then one can arbitrarily select one of the two nodes and eliminate the subtree rooted at that node.

This makes it so there are most L/2+1 nodes left unpruned at any level. So the algorithm will run in n + L polynomial time. The iterative algorithm is copied straight from the notes for subset sum:

```
Sum[0,0]=True
For S=1 to L/2 do Sum[0, S]= False
For k=1 to n do
    For S = 0 to L/2 do
        Sum[k, S] = Sum[k-1, S] or Sum[k-1, S - x(k)]
```

The solution is then found by working backwards from the table created.

15. Give an algorithm for the following problem whose running time is polynomial in n+log$L$:
Input: positive integers $v_1, \ldots, v_n$ and L.
Output: A solution (if one exists) to $(\sum_{i=1}^{n} x_i * v_i) \bmod n = L \bmod n$ where each x_i is either 0 or 1. Here x mod y means the remainder when x is divided by y

Solution: We will construct a dynamic programming algorithm by the pruning method. First, think of an n-level binary tree, where, for each node in level i with value p, the two children of that node in level $i+1$ have values $(p + 0 * v_i) \bmod n$ and $(p + 1 * v_i) \bmod n$. We will prune this tree to reduce the complexity. If two nodes in the same level have the same value (or sum), prune one of them. Then, there will only be at most n nodes at each level in the tree. Since there are n levels, we can construct an $n \times n$ matrix $ModSum$ to represent the tree. Each row will represent a level, and each column will represent a value (sum). Then, each location in $ModSum$ will have either a 0 or a 1 to indicate whether the sum indicated by the column value can be achieved by the level indicated by the row value. Observe that, if there is some node with value L at row j, there will also be a leaf node (in the bottom row of the tree/matrix) that also has value L because we can chose $x_{j+1}, \ldots, x_n$ to all be 0. Then, to find the solution, we simply need to look at whether $ModSum[n][L \bmod n] = 1$ and, if it is, work up the matrix to determine the sequence $x_1, \ldots, x_n$. This algorithm would then look like:

```
MODSUM(v_1,...,v_n, L)
    ModSum[n + 1][n]
    ModSum[0][0]=1
    for i from 1 to n
        ModSum[0][i] = 0
    for i from 1 to n
        for j from 0 to n − 1
            if ModSum[i][j] = 1
                ModSum[i + 1][j] = 1
                ModSum[i + 1][(j + v_i) mod n] = 1
```

```
if ModSum[n][L mod n] = 0
    return ∅
else
    {x_1, x_2, ..., x_n} = {0, 0, ..., 0}
    col = L mod n
    for i from n to 1
        if ModSum[i − 1][col] = 1
            x_i = 0
        else
            x_i = 1
            col = (col − v_i) mod n
    return {x_1, ..., x_n}
```

Observere that this algorithm runs in time $O(n^2)$ so it must therefore run in $O((n+\log L)^2)$.