CS1510 Dynamic Programming Problems 18 & 19

Rebecca Negley, Sean Myers

October 5, 2011

18. The input to this problem is a set of n gems. Each gem has a value in dollars and is either a ruby or an emerald. Let the sum of the values of the gems be L. The problem is to determine if it is possible to partition the gems into two parts P and Q, such that each part has the same value, the number of rubies in P is equal to the number of of rubies in Q, and the number of emeralds in P is equal to the number of emeralds in Q. Note that a partition means that every gem must be in exactly one of P or Q. Your algorithm should run in time polynomial in $n + L$.

    Solution: This problem is almost an equivalent of the sum subset problem, with a few key stipulations, that will be addressed. First, we are going to make a tree, with n levels, where the nth level will have the solution as a leaf. If it is possible to make one partition with value $L/2$, then it is possible to make two with L. Also, if it is possible to make that same partition, with $E/2$ emeralds (E being the max amount of emeralds) and $R/2$ rubies, then it is possible to have 2 partitions with a total of L, R and E. So, in each tree, the only two possible branches will be to either take the current ruby or emerald, or not take it.

    Our partitioning rules are near identical: If any value goes over $L/2$, $E/2$ or $R/2$, then we prune it, for it is not a possible solution. Also, if two values are identical prune one IF they have the same amount of both emeralds and rubies and are on the same level. For if solution $i + k_1 + ... + k_{solution}$ meets those requirements and is a solution, then replacing i with j that has the same emeralds and rubies, will produce the same solution.

    So now that we have the partitioning rules established, let's create the algorithm! First we need an array: partition[lvl, L/2, E/2, P/2], where the values being stored can either be a set (if you don't want to backtrack to find the answer), or a 1, in which case you back track to find the answer. For simplicities sake, let's go with 1. The answer will be a 1 at the last spot partition[lvl, L/2, E/2, R/2], if there is a feasible solution.

    First we set the entire array to 0, except for spot 0,0,0,0 which we set to 1. Then, we run a for loop from 0 to n, on the current level ,*lvl*. Throughout these levels, we will add the two children of each node added or not taken, by building it from the level above. We must run through every 3D array at every level, as to make sure all the branches are filled properly. So we need 3 additional for loops to do so. The pseudocode looks like this:

```
initialize partition[lvl, L/2, E/2, R/2] to all 0's
partition[0,0,0,0] = 1

for(lvl = 0 to n):
   for(value = 0 to L/2):
      for(E = 0 to emeralds.length/2):
         for(R = 0 to rubies.length/2):
            partition[lvl+1, value, E, R] = partition[lvl, value, E, R]
            valueNext = value + Gems[lvl+1].value
```

```
if(valueNext ¿ L/2):
  if(Gems[lvl+1].type == RUBY AND R+1 le rubies.length/2 ):
    partition[lvl+1, value+valueNext, E, R+1] = partition[lvl, value, E, R]
  else if(Gems[lvl+1].type == EMERALD AND E+1 le emeralds.length/2 ):
    partition[lvl+1, value+valueNext, E+1, R] = partition[lvl, value, E, R]
```

19. The input to this problem consists of an ordered list of n words. The length of the ith word is w_i, that is the ith word takes up w_i spaces. (For simplicity assume that there are no spaces between words.) The goal is to break this ordered list of words into lines, this is called a layout. Note that you can not reorder the words. The length of a line is the sum of the lengths of the words on that line. The ideal line lengh is L. No line may be longer than L, although it may be shorter. The penalty for having a line of length K is $L - K$. *The total penalty is the* **maximum** *of the line penalties.* The problem is to find a layout that minimizes the total penalty. Give a polynomial time alorithm for this problem.

Solution: First we will construct a tree with all possible solutions at the leaves. There will be n levels, each corresponding to a word. A node at level i will have two children at level $i + 1$, one with word w_{i+1} on the same linel as w_i and one with w_{i+1} on the next line. Then, the tree will be n levels deep and have 2^n possible solutions at the leaves. To reduce the unnecessary complexity of this tree, we will define two pruning rules:
(1) If a node has a line of length $> L$, prune it because it violates the conditions of the problem and can have no solution.
(2) If two nodes at the same level have last lines of the same length, prune the one with the greater total penalty. If a solution exists in the subtree of the node with the greater penalty, a better solution exists in the subtree of the node with the lesser penalty.

Next, we can use this pruned tree to construct a dynamic programming algorithm. Observe that there are at most L possible lengths for the final line and $n + 1$ levels to the tree (counting the 0 level where we have not yet added any words). We can construct an $(n + 1) \times L$ matrix MPL to store total penalty values at each node in the tree. We initialize $MPL[0, L]$ to 0 and set every other value in the table to $+\infty$. (We can think of $MPL[0, L]$ as $MPL[0, 0]$ because the next word added will automatically go to the next line.) We then iterate down level by level, and each non-infinite penalty gives two possible penalties at the next level. When assigning a penalty to a location in the tree, we take the minimum of the new contender and the value already at the location to get the minimum possible penalty at that location. The solution would be the minimum total penalty in the last line of the matrix (that is, the minimal total penalty when all words have been added). Our code would look like this:

```
minimumPenaltyLayout(w_1, ..., w_n, L)
    for i from 0 to n do:
        for j from 1 to L do:
            MPL[i, j] = +∞
    MPL[0, L] = 0
    for i from 1 to n do:
        for j from 1 to L do:
            if MPL[i, j] < +∞ then:
                if j + w_i ≤ L then:
                    MPL[i + 1, j + w_i] = min{MPL[i + 1, j + w_i], MPL[i, j] − w_i} //add to current line
                MPL[i + 1, w_i] = min{MPL[i + 1, w_i], MPL[i, j] + (L − w_i)} //add to next line
    return min_j MPL[n, j]
```