9/12

CS 1510 Algorithm Design

Dynamic Programming

Problems 16, 17, and 19

Due Monday September 21, 2014

Buck Young and Rob Brown

68.4% 65/95

Problem 16

Input: Positive integers $v_1, ..., v_n$

Output: A subset S of the integers such that $\sum_{v_i \in S} v_i^3 = \prod_{v_i \in S} v_i$

Algorithm: The "Subset-Sum-Product" algorithm SSP is defined and detailed below.

Pruning Rule: If two nodes on the same level have the same sum ($\sum$) and same product ($\prod$), prune either.

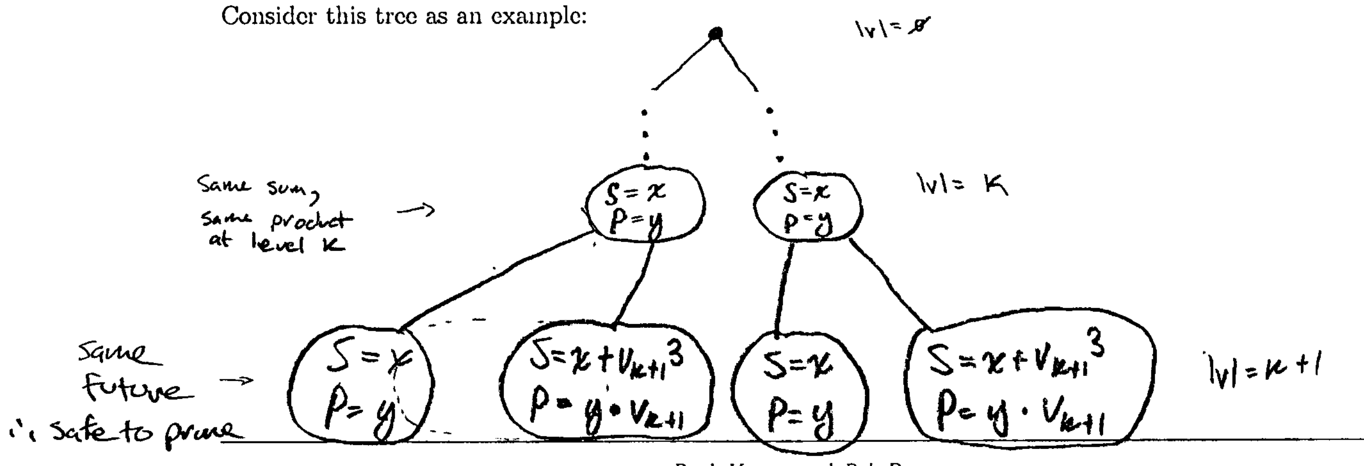
```
SSP[ 0 ][ 0 ][ 0 ] = 1 //initialize
for lvl = 0 to n do
    for S = 0 to L do
        for P = 0 to L do
            if SSP[ lvl ][ S ][ P ] is defined then
                SSP[ lvl + 1 ][ S ][ P ] = 1 //left-child
                SSP[ lvl + 1 ][ S + $v_{lvl+1}^3$ ][ P * $v_{lvl+1}$ ] = 1 //right-child
```

3

Our algorithm cycles through the levels from top to bottom and through all possible sums and products from left to right. By our pruning rule, if two configurations lead to the same sum and the same product on the same level then one is pruned (does not matter which). For the "left-child" in the tree, we are choosing to take the same value as the "parent". For the "right-child" in the tree, we are summing and multiplying in the value of that level according to the given expressions.

Ultimately, we will arrive at a solution when the sum and product indices are equal. At this point, we can backtrace through the array to construct the subset. The backtrace would include subtracting the cube of the value at this level from S and dividing P by the value at this level, and then going up by one level.

The justification for the chosen pruning rule is that at any level, if the sum and the product are the same then anything we add or multiply to those values will be identical at the next level and in the future. Consider this tree as an example:

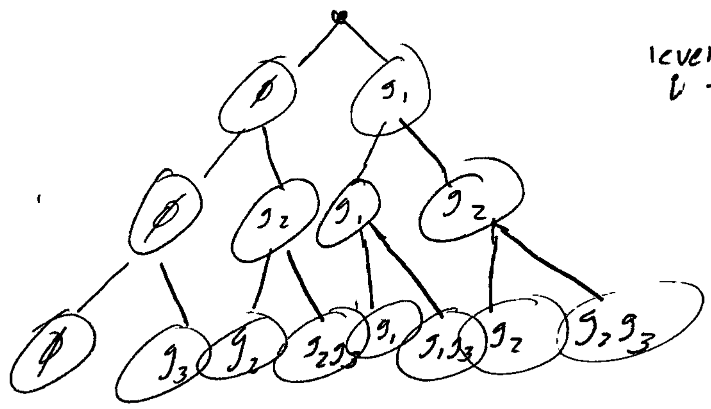

$|v| = \cancel{0}$
Same sum, same product at level K
$S = x$, $P = y$
$S = x$, $P = y$
$|v| = K$
Same future ∴ safe to prune
$S = x$, $P = y$
$S = x + v_{k+1}^3$, $P = y \cdot v_{k+1}$
$S = x$, $P = y$
$S = x + v_{k+1}^3$, $P = y \cdot v_{k+1}$
$|v| = k+1$

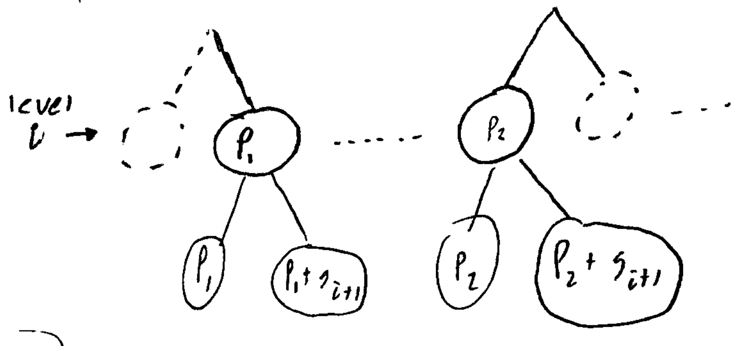
Problem 17

Input: A set $G = \{(t_1, v_1), (t_2, v_2), ..., (t_n, v_n)\}$ of gems with type t equal to either e or r and price p being the integer value of the gem.

Output: A partition of G into two parts P and Q such that such that each part has the same value, the number of rubies in P is equal to the number of rubies in Q, and the number of emeralds in P is equal to the number of emeralds in Q.

To solve this problem, we will enumerate all of the possible compositions for the set P. We can exploit the fact that if some $(t_i, v_i) \notin P$ then it must be the case that $(t_i, v_i) \in Q$ since every gem must be in one of our two partition sets. Consider the tree below enumerating all compositions of P (from which one could derive the composition of Q at any given node).





If $|P_2|_r = |P_1|_r$, $val(P_2) = val(P_3)$ and $|P_2|_e = |P_1|_e$ then it follows that the same will be true for the next level, as g_{i+1} has static value and is either a ruby or emerald.

Purning Rule 1
If it is ever the case that $\sum_{v_i \in node} v_i > \frac{L}{2}$ for some node, we know that there is not enough value in the left-over gems to meet our condition for the value of P equaling the value of Q.

Pruning Rule 2
At a given level of our tree, if two nodes have the same number of rubies, the same number of emeralds, and the same total sum then we can prune one of them arbitrarily. For a graphical explanation of this, see the tree above. Also note that if two compositions for the set P have the same value sum (v_P), number of emeralds (E_P), and the same number of rubies (R_P), then the selection for Q is fixed such that $R_Q = R_{level} - R_P$, $E_Q = E_{level}$, $v_Q = v_{level} - v_p$ (where each "level" variable is the total value, rubies, or emeralds being considered up to that level).

Algorithm:

```
T[n][n][n][L]
T[0][0][0][0] = 1
for $i = 1$ to $n$ do
    if $t_i$ is $r$ then
        $r_i = 1$
        $e_i = 0$
    else
        $r_i = 0$
        $e_i = 1$
    for $r = 1$ to $n$ do
        for $e = 1$ to $n$ do
            for $v = 0$ to $\frac{L}{2}$ do
                if $T[i][r][e][v]$ is defined then
                    $T[i+1][r][e][v] = 1$
                    if $v + v_i < \frac{L}{2}$ then
                        $T[i+1][r+r_i][e+e_i[v+v_i] = 1$
```

3

To reconstruct P from our table, we must first find some $0 \leq r \leq n$ and $0 \leq r \leq n$ such that $T[n][r][e][\frac{L}{2}]$ is defined (if there are multiple, then there is more than one solution). From this index we step back through the array. At a given level we know a) what type of gem we have (boolean values r_i or e_i, and b) what its value is v_i. We look at $T[i-1][r-r_i][e-e_i][v-v_i]$ and $T[i-1][r][e][v]$. If the the origin of our cell is the former, we add (t_i, v_i) to our set P and continue until $i = 0$. We then construct Q according to the relative compliment $Q = G \setminus P$.

Regarding complexity, we clearly have an algorithm which is $O(\frac{1}{2}n^3 L)$ which we consider to be polynomial in $n + L$ since $(n+L)^4 \geq (n+L)^3 L \geq n^3 L$.

Problem 19

Input: Two sequences $T = t_1, ..., t_n$ and $P = p_1, ..., p_k$ such that $k \leq n$, and a positive integer cost c_i associated with each t_i

Output: A subsequence of T that matches P with maximum aggregate cost. That is, find the sequence $i_1 < ... < i_k$ such that for all j, $1 \leq j \leq k$, we have $t_{i_j} = p_j$ and $\sum_{j=1}^{k} c_{i_j}$ is maximized.

a)

Define $RMAC$ to recursively solve the problem stated above. The input parameters will be the length of T (n) and the length of P (k).

```
1: function RMAC(n, k)
2:     if $n == 0$ or $k == 0$ then
3:         return 0
4:     if $T_n == P_k$ then
5:         return $RMAC(n-1, k-1) + c_n$
6:     return max($RMAC(n-1, k)$, $RMAC(n, k-1)$)
```

Dynamic Programing?

Problem 19 b/c

Algorithm: The "Max Aggregate Cost" algorithm MAC is described below.

Pruning Rules:

1) If the length of the string at any given node is greater than k, prune that node.

2) If the length of the string at any given node equals k but the string does not match P, prune that node.

3) If the string is out of order at any given node, prune that node. (That is, if the first character of the node does not match the first character of P, then no descendants of this node will have the solution -- and so on)

4) If two nodes at the same level have the same length (and all of the above 3 conditions are met), keep the node with the maximum aggregate cost.

```
for j = 1 to n do
    if T[ j ] == P[ 1 ] then
        MAC[ j ][ 1 ] = $C_j$

for lvl = 1 to n do
    for L = 1 to k do
        if MAC[ lvl - 1][ L ] is defined then
            //left-child
            MAC[ lvl ][ L ] =
            max( MAC[ lvl ][ L ], MAC[ lvl - 1 ][ L ] )

            if L + 1 is in bounds and T[ lvl ] == P[ L ] then
                //right-child
                MAC[ lvl ][ L + 1 ] =
                max( MAC[ lvl ][ L + 1 ], MAC[ lvl -1 ][ L ] + $C_{lvl}$ )
```

-2

Example

L →

lvl ↓

	1	2
1	2	
2	2	4
3	7	4
4	7	4
5	7	8

$x_3 + y_5 = \underline{\underline{8}}$

This algorithm will build a table in which we can store the values of the subsequences. At first, we initialize the array's first column with the values of each C_j if the corresponding letter in T is the same as the first letter in P. Then we iterate through and build the next level of the array. For the left child, we are checking against the node above it and the "champ" that is already at that cell. For the right child, we check against the value that is currently there and the value diagonally to the upper-left (added to the value at that level). Along the way, we make a few checks in order to adhere to our pruning rules.

At the end of the day, our solution will be the maximum value of length k. This should be in the lower-right corner. From here we can backtrace to find our solution.

c?