5 ~~0~~/10

CS 1510
Algorithm Design

Dynamic Programming

Problems 9, 11, and 13

Due Wednesday September 17, 2014

Buck Young and Rob Brown

49/75

Problem 9

Input: Consists of n keys $K_1, ..., K_n$ with $K_1 < K_2 < ... K_n$ and associated probabilities $p_1, ..., p_n$.

Output: AVL tree for these keys that minimizes the expected depth of a key.

Algorithm: This algorithm is very similar to the MWBST with strengthening due to the depth requirements of the AVL tree constraints. It uses three tables including an optimal probability (PROB), max depth (DEPTH), and an optimal root (ROOT) of the subtrees. The algorithm is detailed and explained below:

```
1: // Initialize three tables and an empty acceptable root list
2: for i = 1 to $n$ do
3:     PROB[ i ][ i ] = $p_i$
4:     PROB[ i ][ i - 1 ] = PROB[ i + 1 ][ i ] = 0
5:     DEPTH[ i ][ i ] = 1
6:     DEPTH[ i ][ i - 1 ] = DEPTH[ i + 1 ][ i ] = 0
7:     ROOT[ i ][ i ] = $K_i$
8:     acceptableRoots = {}
9:
10: // Iterate through to build tables from bottom to top and left to right
11: for a = $n$ - 1 to 1 do
12:     for b = a + 1 to $n$ do
13:
14:         // Find acceptable roots based on AVL depth constraints
15:         for $\alpha$ = a to b do
16:             if abs(DEPTH[ a ][ $\alpha$ - 1 ] - DEPTH[ $\alpha$ + 1 ][ b ]) $\leq$ 1 then
17:                 acceptableRoots.add( $\alpha$ )
18:
19:         // Find the MWBST of the acceptable roots using the PROB table
20:         PROB[ a ][ b ] = $\min_{\forall \alpha \in acceptableRoots}$( PROB[ a ][ $\alpha$ - 1 ] + PROB[ $\alpha$ + 1 ][ b ] ) + $\sum_{k=a}^{b} p_k$
21:
22:         // Update other tables based on the optimal root
23:         R = the chosen optimal $\alpha$ from the min function on $L20$
24:         ROOT[ a ][ b ] = $K_R$
25:         DEPTH[ a ][ b ] = max( DEPTH[ a ][ R - 1 ], DEPTH[ R + 1 ][ b ] ) + 1
26:
27:         // Clear acceptable root list
28:         acceptableRoots = {}
```

2

how do you know acceptable roots is non empty? This might not find the answer.

$L2 - L8$: Initialize three arrays for keeping track of information about the subtrees from a to b: an optimal probability PROB, a reported max depth DEPTH, and the optimal root ROOT. Also, create an empty list for storing the acceptable roots (which are a subset of all nodes from a to b based on the AVL balancing constraints).

$L11 - L12$: Iterate through the arrays from the bottom up and from left to right.

$L15 - L17$: Create a list of acceptable roots based on the balancing factor constraints of an AVL tree. The absolute value of the total depth of the left subtree minus the total depth of the right subtree is bounded by 1 (as the balancing factor can have a value of -1, 0, or 1). All possible roots which adhere to this constraint are added to the acceptableRoots list.

$L20$: Update the PROB array based on the minimum sum of the left subtree and the right subtree for each possible acceptable root. This min function should also set R on $L23$ (which is the optimal root α chosen by min). Add in the summation of all probabilities from a to b. This is very similar to the MWBST algorithm which we covered in class. The main difference is that min functions only on a subset of values from a to b (the possible acceptable roots). Min still returns the "weight" (or in this case, "probability") for the tree a to b. Additionally, min remembers which choice it made for the optimal root and uses that information to set R from $L23$.

$L24 - 28$: This is where we update the ROOT and DEPTH array based on the chosen optimal root α. We consider the maximum depth of each subtree and add one for the new optimal root α. Finally, we clear the acceptableRoots list so that it is empty for the next iteration.

Problem 11

a)

Consider the values in the last row between columns 0 and L. From these entries, pick the one with the highest value. Let this be at index $(w,\ lvl)$. Consider the values at $T[w][lvl-1]$ and $T[w-w_{lvl}][lvl-1]$. If $T[w-w_{lvl}][lvl-1] = T[w][lvl] - v_{lvl}$, then add v_{lvl} to your output and move to index $(w - w_{lvl},\ lvl-1)$. If instead you found that $T[w-w_{lvl}][lvl-1] \neq T[w][lvl] - v_{lvl}$, then you will find $T[w][lvl-1] = T[w][lvl]$ and you should move to $T[w][lvl-1]$ without adding anything to your output set. Continue this process until $T[w][lvl] = 0$.

As an example, consider the input $values = \{3,4,6,2\}$, $weights = \{4,1,2,3\}$, and $L = 8$. The following table is created.

lvl \ w	0	1	2	3	4	5	6	7	8	9	10
0	0	_	_	_	_	_	_	_	_	_	_
1	0	_	_	_	3	_	_	_	_	_	_
2	0	4	_	_	3	7	_	_	_	_	_
3	0	4	6	10	$\cancel{3}$	7	9	13	_	_	_
4	0	4	6	10	6	8	12	13	9	11	15

$(lvl, w) = (4, 7)$ $v_4 = 2$ $w_4 = 3$ $[13 - 3 \neq 3]$ ✗ $\Rightarrow$ move to $(3, 7)$

$(lvl, w) = (3, 7)$ $v_3 = 6$ $w_3 = 2$ $[13 - 6 = 7]$ ✓ $\Rightarrow$ move to $(2, 5)$ add v_3 to OUTPUT

$(lvl, w) = (2, 5)$ $v_2 = 4$ $w_3 = 1$ $[7 - 4 = 3]$ ✓ $\Rightarrow$ move to $(1, 4)$ add v_2 to OUTPUT

$(lvl, w) = (1, 4)$ $v_1 = 3$ $w_1 = 4$ $[3 - 3 = 0]$ ✓ $\Rightarrow$ move to $(0, 0)$ add v_1 to OUTPUT.

$$\boxed{\text{OUTPUT} = \{v_1, v_2, v_3\}}$$

b)

We note that all rows above our last row are redundant, so our goal is to eliminate them. We also know that if $w > L$ we can stop since we have exceeded our maximum weight, and that if $k > n$ we can stop since we have considered all values in our input. So we have reduced the algorithm we discussed in class (two "for" loops) to be $O(nL)$ time complexity.

Need to be careful how this single row is updated.

since everything above the final row (of length L) is redundant given the desired output (weight/value pairs), can choose to simply over-write the level indexes directly above one another (since the for $lvl < n$ these values are either identical or obsolete).

Problem 13

Input: Positive integers $v_1, ..., v_n$, with $L = \sum_{i=1}^{n} v_i$

Output: A solution (if one exists) to $\sum_{i=1}^{n} (-1)^{x_i} v_i$ where each x_i is either 0 or 1.

Algorithm: Algorithm "Sum To Zero" (STZ) is detailed and explained below.

```
1: // Initialize Table
2: STZ[ 0 ][ 0 ] = 0
3:
4: // Iterate through to build table
5: for a = 1 to $n$ do
6:     for b = 0 to $L$ do
7:         if STZ[ a - 1 ][ b ] is defined then
8:             STZ[ a ][ b ] = STZ[ a - 1 ][ b ] - $v_a$ // Left child (subtract)
9:             STZ[ a ][ b+$v_a$ ] = STZ[ a -1 ][ b ] + $v_a$ // Right child (add)
```

Pruning Rules:
First, we allow pruning to any sums that are over half of L or under half of -L. Put another way, we take the absolute value of the sum and allow pruning to sums greater than half of L:

1) $|\text{sum}| > L/2$

Second, we allow pruning to occur on any sums which are the same on the same level, positive or negative. Put another way, we take the absolute value of the sum and allow pruning to sums that are the same on the same level:

2) |sum| same on same level

Explanation of STZ: The algorithm described above will help us model a binary tree solution as an array using the stated pruning rules.
*L*2: Initialize (0,0) to the value 0 in order to get us started

*L*5 – *L*6: The outer loop goes level by level and the inner loop goes solution by solution on each level

*L*8: This is the left child in the binary tree, thus we are subtracting the value of that level (equates to $x_i = 1$). Basically, if you are building the table, we drop the value from straight above to this cell, subtracting the value of this level.

*L*9: This is the right child in the binary tree, thus we are adding the value of that level (equates to $x_i = 0$). Basically, if you are building the table, we drop the value from 1 level above and over by an amount equal to the value of this level, adding the value of this level.

If there is a solution, we will find the value 0 on level n and can backtrace to determine the path.