4/5

CS 1510
Algorithm Design

Dynamic Programming

Problems 4 and 5

Due Friday September 12, 2014

Buck Young and Rob Brown

36/53

Problem 4

Input: Two strings, $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$

Output: The minimum cost steps to convert A to B according (where the cost of deletion is 3, the cost of insertion is 4, and the cost of replacement is 5.

Algorithm:

```
//set boundries (ie, base cases from recursion)
DIFF[m + 1][n + 1]
for 0 ≤ a ≤ m do
    DIFF[a][0] = 3 * a //if we get here, the "recersion" is done and we have no option but deletion
end for
for 0 ≤ b ≤ n do
    DIFF[0][b] = 4 * a //if we get here, the "recursion" is done and we have no option but insertion
end for

//iterate in place of recursive call
for 1 ≤ a ≤ m do
    for 1 ≤ b ≤ n do
        if A[a] == B[b] then
            DIFF[a][b] = DIFF[a − 1][b − 1] + 0 //characters match, do nothing
        else
            DIFF[a][b] = min(DIFF[a − 1][b] + 3, DIFF[a][b − 1] + 4, DIFF[a − 1][b − 1] + 5)
        end if
    end for
end for
```

Need to explain algorithm better, e.g. what is Diff?

			a	b	c	a	b	c
		0	1	2	3	4	5	6
	0	0	3	6	9	12	15	18
a	1	4	0	3	6	9	12	15
b	2	8	4	0	3	6	9	12
a	3	12	8	4	5	3	6	9
c	4	16	12	8	4	7	10	6
a	5	20	16	12	8	4	7	10
b	6	24	20	16	12	8	4	7

To reconstruct:
start at (m, n).
Take min value from $(m-1, n)$, $(m, n-1)$, and $(m-1, n-1)$.

If $m-1$: delete a_m from A
If $n-1$: insert b_n into A at $m+1$
If $(m-1, n-1)$: replace a_m with a_n if not equal

Continue by moving to the index chosen above.

EXAMPLE: move left. delete ↓ disagree. Insert $b_3 = a$ ↓

A = abcabC ⇒ A = abcab

⇒ A = abacab ✓

Problem 5

Here are the generated tables for the given problem. Scratch-work for the non-trivial calculations are attached. A hand-drawn picture of the sample trace (explained below) for the optimal solution is also included with the scratch-work.

Table 1: Optimal Access Times

a \ b	1	2	3	4	5
1	0.5	0.6	0.85	1.4	2.15
2	-	0.05	0.2	0.55	1.05
3	-	-	0.1	0.4	0.9
4	-	-	-	0.2	0.65
5	-	-	-	-	0.25

Table 2: Optimal Roots

a \ b	1	2	3	4	5
1	K_1	K_1	K_1	K_1	K_1
2	-	K_2	K_3	K_4	K_4
3	-	-	K_3	K_4	K_4
4	-	-	-	K_4	K_5
5	-	-	-	-	K_5

Example reconstruction of optimal tree:

1) You want to find the optimal root for all nodes K_1 ... K_5, so check the Roots Table at a=1 and b=5. The optimal root is K_1 from the table. Add K_1. At this point, the left-hand side of node K_1 is null (because this is nothing less-than K_1 in the given problem) and the right-hand side is the optimal subtree for nodes K_2 ... K_5.

2) Continue in this manner. You want the optimal root for K_2 to K_5, so check the Roots Table at a=2 and b=5. The optimal root is K_4. Add K_4. The left-hand side of the node K_4 is the optimal subtree of nodes K_2 ... K_3 and the right-hand side is from K_5 ... K_5.

3) Check the table for (2, 3), get the node K_3. Add K_3 to the left-hand side of K_4. The left-hand side of our new K_3 node is the optimal subtree of K_2 ... K_2. Check the table for (5,5) – obviously it is node K_5. Add K_5 to the right-hand side of K_4. There are no subtrees for node K_5.

4) Add K_2. Verify that your answer is optimal by adding up the access times * depth and checking Table 1 at a=1 and b=5.

From these instructions, we have the optimal subtree.

Please check the attached sheet for a drawn picture representing the above process.

non-trivial Computations for the Optimal Access Times Table

a = 3
b = 5

$$\min[(\varnothing + .65),\ (.1 + .25),\ (.4 + \varnothing)] + \sum_{k=a}^{b} w_k$$

(* root = 4)

$$= .35 + .1 + .2 + .25 = .9$$

---

a = 2
b = 4

$$\min[(\varnothing + .4),\ (.05 + .2),\ (.2 + \varnothing)] + \sum_{k=a}^{b} w_k$$

(root = 4 *)

$$= .2 + .05 + .1 + .2 = .55$$

---

a = 1
b = 3

$$\min[(\varnothing + .2),\ (.5 + .1),\ (.6 + \varnothing)] + \sum_{k=a}^{b} w_k$$

(* root = 1)

$$= .2 + .5 + .05 + .1 = .85$$

---

a = 2
b = 5

$$\min[(\varnothing + .9),\ (.05 + .65),\ (.2 + .25),\ (.55 + \varnothing)] + \sum_{k=a}^{b} w_k$$

(* root = 4)

$$= .45 + .05 + .1 + .2 + .25 = 1.05$$

$a=1$
$b=4$

$$\min[(\emptyset + .55)^{*\text{root}=1},\ (.5 + .4),\ (.6 + .2),\ (.85 + \emptyset)] + \sum_{k=a}^{b} w_k$$

$$= .55 + .5 + .05 + .1 + .2 = 1.4$$

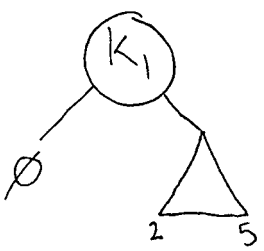
$a=1$
$b=5$

$$\min[(\emptyset + 1.05)^{*\text{root}=1},\ (.5 + .9),\ (.6 + .65),\ (.85 + .25),\ (1.4 + \emptyset)] + \sum_{k=a}^{b} w_k$$

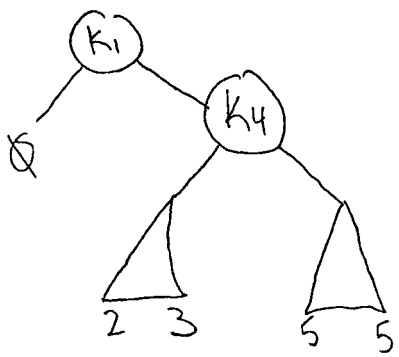
$$= 1.05 + .5 + .05 + .1 + .2 + .25 = 2.15$$

SAMPLE TRACE OF $a=1$ $b=5$

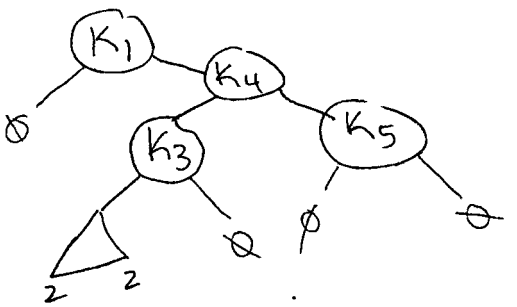
① Get root (1,5) from Root Table



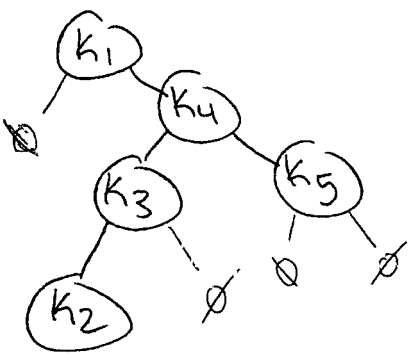
② Get root (2,5)



③ Get root (2,3) and (5,5)



④ Get root (2,2)



$1 \times .5$
$2 \times .2$
$3 \times .1 + 3 \times .25$
$4 \times .05$

2.15 ✓

8/12

CS 1510
Algorithm Design

Dynamic Programming

Problems 6, 7, and 8

Due Monday September 15, 2014

Buck Young and Rob Brown

44/65

67.7%

Problem 6

Input: An array of vertexes $P = [n_1, n_2, ..., n_N]$ such that n_1 connect with n_2, n_2 connects with n_3, and so on until n_N connects with n_1 to form a polygon. Note that each n_i is an ordered pair of (x, y) coordinates in Cartesian space.

Output: The triangulation of P into $N - 2$ triangles such that the sum of the perimeters of the triangles is minimized. Or, equivalently, that the sum of the cuts to form these triangles is minimized.

Algorithm:

```
1:  CUT[N][N];
2:  for 0 ≤ i ≤ N do
3:      for 0 ≤ j ≤ N do
4:          if num_nodes_from(P[i], P[j]) == 3 then
5:              CUT[i][j] = 0
6:          else if num_nodes_from(P[i], P[j]) < 3 then
7:              CUT[i][j] = ∞
8:
9:  for 0 ≤ i ≤ N do
10:     for N ≤ j ≤ 0 do
11:         if CUT[i][j] == ∞ or CUT[i][j] == 0 then
12:             continue
13:         e'_i = norm(P[(i − 1)%N], P[j])
14:         e'_j = norm(P[i], P[(j + 1)%N])
15:         CUT[i][j] = min(CUT[(i − 1)%N][j] + e'_i, CUT[i][(j + 1)%N] + e'_j)
```

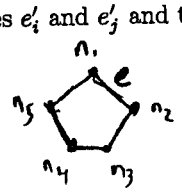
6

At a high level, each (i, j) index pair represents a subset of the original polygon, containing only the nodes between P[i] and P[j] (the excluded nodes have been removed to form a "cut" for triangulation).

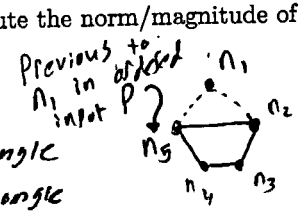
On lines 5 and 7 we see our two base-cases. Both of these depend on the theoretical (but easily implementable) function "$num\_nodes\_from(n_1, n_2)$". This function would find the number of nodes between (and including) n_1 and n_2 in P. The base-case at line 5 is equivalent to the final step of recursion, where the polygon has been reduced to 3 nodes/edges, and cannot be cut any further. Line 7 refers to illogical states where the polygon is no longer closed (ie, $nodes \leq 2$).

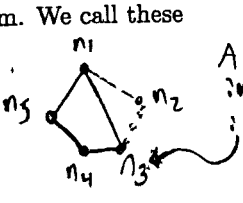
The two lengths on 13 and 14 are the lengths of the cuts that would be made if we excluded node P[i] and P[j], respectively. The only meaningful values in our table are when i and j are ± 1 index from each other. Since our input is ordered so that the i^{th} node is connected to the $(i+1)^{th}$ node (and vice-versa), we can think of P[i] and P[j] as members of an edge in this case. This edge can be contained in two triangles: one where P[i] is excluded and P[j] is connected to the node preceding P[i] in P, and one where P[j] is excluded and P[i] is connected with the node following P[j] in P. See diagram below. These two pairs of nodes, again, can be considered to be edges or vectors, and we can compute the norm/magnitude of them. We call these two values e'_i and e'_j and they are the length of our cut.

Pretty confusing



e = {n1, n2}
exclude n1 => one triangle
exclude n2 => other triangle



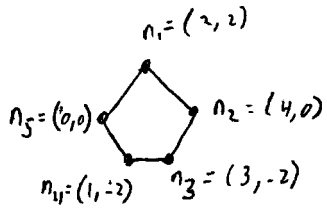


After n2 in ordered input P

On line 15 we then choose our preferred cut to be the minimum between e'_i and all of its prior cuts and e'_j and all of its prior cuts.

Finding the solution using the table In general, to find our triangulation, we start at the index $(i,\ j) = (N-1,\ 0)$ of CUT. From here, we calculate $e'_i = norm(P[i-1][[j])$ and $e'_j = norm(P[i][j+1])$ (the lengths of the cut edges). For one of these two norm values, we will find $CUT[i][j] - n_i = CUT[i-1][j]$ or $CUT[i][j] - n_j = CUT[i][j+1]$. For the matching case, add the corresponding cut (e'_i or e'_j), move to that index $(i,\ j)$, and continue until $CUT[i][j] = 0$.

As an example of reconstruction, consider the polygon and table below. This can be interpreted as looking at the polygon with nodes ranging from n_1 to n_5 (ie, the whole polygon). Per the solution instructions above, consider $CUT[N-1][0] = CUT[4][0] = 7.606$. We calculate $e'_i = norm(P[3],\ P[0]) = norm(n_4,\ n_1) = 4.1231$ and $e'_j = norm(P[4],\ P[1]) = norm(n_5,\ n_2) = 4$. Now $P[4][0] - e'_j = P[4][1]$. So we add e'_j to our cut list, increment j and move to $(i,\ j) = (4,\ 1)$. Again, we calculate e'_j and e'_i, and find $e'_j = norm(n_5,\ n_3) = 3.606$ and $e'_i = norm(n_4,\ n_2) = 3.606$. So our last cut is arbitrary and we choose e'_j. Thus we have $OUTPUT = \{(n_5,\ n_2), (n_5,\ n_3)\}$ and our polygon is segmented as shown below.



		n_1	n_2	n_3 (i)	n_4	n_5 (j)
		0	1	2	3	4
n_1	0	∞	∞	0	3.606	7.606
n_2	1	∞	∞	∞	0	3.606
n_3	2	∞	∞	∞	∞	0
n_4	3	∞	0	∞	∞	∞
n_5	4	∞	∞	0	∞	∞

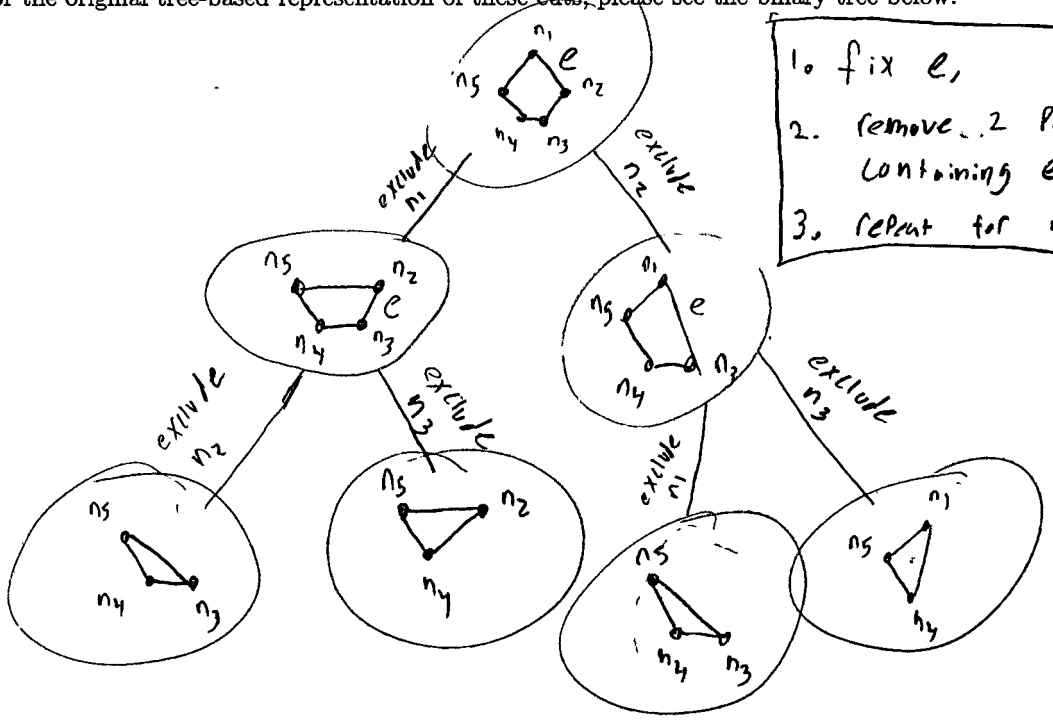
✓ 7.606 - norm(n_5, n_2) = 3.606

✓ 3.606 - norm(n_5, n_3) = 0

P = [n_1 n_2 n_3 n_4 n_5]

output = {e_1, e_2}

For the original tree-based representation of these cuts, please see the binary tree below.



1. fix e_1
2. remove 2 possible triangles containing e
3. repeat for resulting polygons

Problem 7

Input: Sequence S of integers (positive, negative, or zero)

Output: A consecutive sub-sequence with the maximum sum (Max. Consecutive Sum, MCS)

Notes on the Naive Recursive Algorithm: The naive recursive algorithm would be knowing the n^{th} number and the Maximum Consecutive Sum of the first $n-1$ numbers [MCS$(n-1)$]. This is not sufficient information for computing the Maximum Consecutive Sum of the first n numbers [MCS(n)] because we do not know if MCS$(n-1)$ is a subsequence next to the n^{th} number. That is, we do not know if the specific number $n-1$ is included in the MCS of the first $n-1$ numbers. This is an important piece of information if we want to compute MCS(n) because the solution must be *consecutive*. Therefore if we wanted to attempt to include the number n in our solution, the number $n-1$ **must** be contained in the solution. However, we do not have this information with the naive recursive algorithm.

Therefore, when we convert a proper algorithm to the iterative solution, we would be smart to compute both the MCS of the first $n-1$ numbers and also the the MCS of the first $n-1$ numbers with the number $n-1$ included. This way, we can compare MCS$(n-1)$ to MCS_with_n_minus_one_included$(n-1)+n$. Now we know if including the number n would result in a more optimal solution.

Algorithm: Here is the iterative, array-based algorithm – I will explain it below.

```
1: for j=1 to n do
2:     for k=1 to n do
3:         if k == 1 or j == 1 or j > k then
4:             MCS[k][j] = -∞
5:         else
6:             int MCS_with_k_included = Σ_{a=k-j+1}^{k} S_a
7:             MCS[k][j] = max( MCS[k-1][j], MCS[k][j-1], MCS_with_k_included )
```

This algorithm will help us build a table to find the MCS. Let me explain the algorithm line-by-line.

L1&L2: Let j be the maximum length of the subsequence that is being considered. Let k be the maximum index of input S that is being considered. Let n be the number of items in input S. We will start building our table in the upper-left corner and work towards the lower-right corner, going "down" along k rows before changing columns by incrementing j.

L3&L4: This is basically our initialization for the table. We are doing it within the main loop because it is safe to do so. We want the row k=1, the column j=1, and an upper-right portion of the table to all be initialized to negative infinity. This will ensure that those cells are not seriously considered during a call to max() while we are building the table. Further, a significant upper-right portion of the table just doesn't make sense – that is the case when $j > k$. You cannot have a subsequence of length j that is longer than all the numbers up to S_k.

L6: Here we are finding the Maximum Consecutive Sequence which includes the number at S_k. Basically, this summation will add S_{k-1} to S_k if the length j is 2. Similarly, it will add S_{k-2}, S_{k-1}, and S_k together if length j is 3. Et cetera.

L7: Finally, our algorithm will apply a value to the current cell based on the maximum of three numbers. Those numbers are the cell to the left, the cell above, and the MCS with S_k included. It looks to the left to see if there is a better MCS of a smaller length. It looks above to see if there is a better MCS at anytime before S_k. And finally it looks at the MCS with S_k included to determine the benefit of adding S_k to the solution (of length j). This process will continue until we reach MCS[n][n]. From here we will have to determine the solution by tracing through the table.

Finding the solution using the table: First we start at MCS[n][n] and take note of the number there. This value is the sum of our maximum consecutive sequence – but we want to know the sequence. Therefore we must trace left and up until we find the last occurrence of our number. That is, we want the smallest possible length j and smallest possible index k which is "connected" to our maximum sum. This tells us the length of the MCS and the last index of S to include. Using this information, we simply add S_k, $S_k - 1$, ... S_{k-j+1} to the solution and we have our answer! Below is an example input, table building, and finding the solution using the table.

k 1 2 3 4 5 6
S={6, -5, 6, 7, -10, 3}
Solution

		j					
		1	2	3	4	5	6
k	1	-∞	-∞	-∞	-∞	-∞	-∞
	2	-∞	1	-∞	-∞	-∞	-∞
	3	-∞	1	7	-∞	-∞	-∞
	4	-∞	13	13	14	-∞	-∞
	5	-∞	13	13	14	14	-∞
	6	-∞	13	13	14	14	14

If we follow 14 to the left and up the most, we get j=4 and k=4. Now we know that the maximum sum consecutive subsequence is of length 4 and it ends at S_4. Therefore we add those four numbers from the input to our solution and get the optimal sequence: {6, -5, 6, 7}.

Problem 8

Input: Tree T with integer weighted edges (positive, negative, or zero).

Output: The shortest simple path in T, (Shortest Simple Path, SSP).

Algorithm: Here is the iterative, array-based algorithm. Explanation below.

Note: The input is being interpreted as a set of edges and associated values given sequentially numerated n nodes from 1 to n. Example Input: $\{e_{1,2} = 5, e_{1,3} = 10\}$ would describe a tree with 3 nodes, with the edge from node-1 to node-2 having a weight of 5 and the edge from node-1 to node-3 having a weight of 10.

```
1:  for i=n to 1 do
2:      for j=n to 1 do
3:          if e_{i,j} does not exist or i > j then
4:              SSP[i][j] = ∞
5:          else
6:              int min_in_col = ∞
7:              for a = j to n do
8:                  if SSP[j][a] < min_in_col then
9:                      min_in_col = SSP[j][a]
10:             SSP[i][j] = min( e_{i,j}, (e_{i,j} + min_in_col) )
```

This algorithm will help us build a table to find the SSP. Let me explain the algorithm line-by-line.

*L*1&*L*2: We will build the table from the bottom-right to the upper-left – going "up" along j rows before decrementing the i column. Both i and j represent nodes and $e_{i,j}$ is the edge between node i and node j.

*L*3&*L*4: Here is the initialization of the table, basically. We are performing it in the main loop because it is safe to do so. If the edge does not exist, we will set that cell to infinity. Note that any node "to" itself will get a value of infinity by this definition. Further the upper right half of the table will get infinity values. Therefore, we are basically creating a diagonal of infinities down the center-diagonal of the table and also throwing out the upper right half. Finally, to reiterate, we are also putting infinity in for any edges that do not exist as we build the table.

*L*5: Else if the edge does exist...

*L*1 – *L*9: Here we are simply finding the minimum value in a column. The column we are looking in is where i=j. When we state "for a=j to n", we are simply looking at every value in the column from value j on down (this is an optimization because we know that every value in the column i=j from j on up is infinity). So, we are simply trying to find the minimum number in the column i=j. We do this so we can, essentially, find the SSP of that subtree.

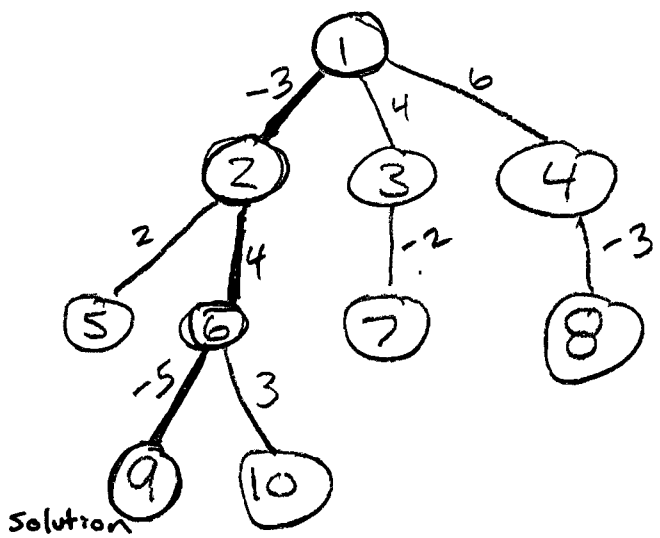
*L*10: Here we finally set a value in the cell. This value should be the minimum between two numbers. Those two numbers are the edge itself and the edge plus min_in_col (which is the SSP of a subtree). By doing this, we can determine if the descendant subtree plus the edge in consideration is worth adding to the solution or if we should just add the considered edge by itself. Next we will discuss how to find the solution

by using the table which we just created.

Finding the solution using the table: This process is relatively simple. First we find the absolute minimum number on the table and remember that number. Then we add that edge ($e_{i,j}$) to our solution set. Now we go to the column i=j and find the minimum number in that column and add $e_{i,j}$ to our solution set. We again goto the column i=j and... etc. etc. We continue in this manner while the absolute minimum number is still less than the sum of the edges in the solution set. As soon as they are equal, we stop.

More simply, we have an absolute minimum number and each edge we add contributes to that minimum value. We continue to add the edges in the solution set until we have reached our minimum value. Below is an example input tree, the created table, and the trace through the table using the methods described.

Tree:



Input: $\{e_{1,2}$=-3, $e_{1,3}$=4, $e_{1,4}$=6, $e_{2,5}$=2, $e_{2,6}$=4, $e_{3,7}$=-2, $e_{4,8}$=-3, $e_{6,9}$=-5, $e_{6,10}$=3$\}$

Table:

j \ i	1	2	3	4	5	6	7	8	9	10
1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	(-4)	∞	∞	∞	∞	∞	∞	∞	∞	∞
3	2	∞	∞	∞	∞	∞	∞	∞	∞	∞
4	3	∞	∞	∞	∞	∞	∞	∞	∞	∞
5	∞	2	∞	∞	∞	∞	∞	∞	∞	∞
6	∞	(-1)	∞	∞	∞	∞	∞	∞	∞	∞
7	∞	∞	-2	∞	∞	∞	∞	∞	∞	∞
8	∞	∞	∞	-3	∞	∞	∞	∞	∞	∞
9	∞	∞	∞	∞	∞	(-5)	∞	∞	∞	∞
10	∞	∞	∞	∞	∞	3	∞	∞	∞	∞

absolute min = −4

solution $\sum e_{1,2}$ = −3, ... absmin < $\sum$ solution set

$e_{2,6}$ = 4, ... absmin < $\sum$ solution set

$e_{6,9}$ = −5 } ... absmin = $\sum$ solution set

∴ Stop!

Solution: $\{e_{1,2}, e_{2,6}, e_{6,9}\}$

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 α = a to b do
16:             if abs(DEPTH[ a ][ α - 1 ] - DEPTH[ α + 1 ][ b ]) ≤ 1 then
17:                 acceptableRoots.add( α )
18:
19:         // Find the MWBST of the acceptable roots using the PROB table
20:         PROB[ a ][ b ] = min_{∀α∈acceptableRoots} ( PROB[ a ][ α - 1 ] + PROB[ α + 1 ][ b ] ) + Σ_{k=a}^{b} p_k
21:
22:         // Update other tables based on the optimal root
23:         R = the chosen optimal α 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 = {}
```

$L2 - L8$: 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	X	7	9	13	-	-	-
4	0	4	6	10	6	8	12	13	9	11	15

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

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

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

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

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) |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.
$L2$: Initialize (0,0) to the value 0 in order to get us started

$L5 - L6$: The outer loop goes level by level and the inner loop goes solution by solution on each level

$L8$: 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.

$L9$: 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.

7/8

CS 1510
Algorithm Design

Dynamic Programming

Problems 10, 14, and 15

Due Friday September 19, 2014

Buck Young and Rob Brown

56/83

Problem 10 (a) Recursive First

Input: Collection A of n intervals over the real line sorted in increasing order of their left endpoint.

Output: Collection C of non-overlapping intervals such that the sum of the lengths of the intervals in C is maximized.

Algorithm: An algorithm for solving this problem in polynomial time is detailed and described below. The hint asked us to consider what two pieces of information are essential to strengthening the inductive hypothesis. The one was obviously the length of the intervals and the other is knowing the interval's start and end (to determine overlaps). The algorithm below defines intervals in the form of $(start, end)$ where interval I_k starts at $I_k(start)$ and ends at $I_k(end)$.

Note: The hint also mentions that we should consider the input in increasing order of their left endpoint ($A_k(start)$). I included this constraint in the definition of the input.

This algorithm iteratively creates two arrays, one which stores the last interval in the considered output collection and another which stores the sum of the lengths of the intervals in the considered output collection. These arrays are labelled INT (for last interval) and LEN (for summed lengths). The indexes are the current interval k from the input collection A and the interval length j. We are essentially looking to put the "best" last interval into the INT array. Best is defined as being as close to length j as possible (without being longer) and having the left-most right-endpoint (end).

```
1:  for k = 1 to n do
2:      for j = 1 to max(A(end)) do
3:
4:          if length(A_k) > j then
5:              LEN[ k ][ j ] = max( LEN[ k ][ j -1 ], LEN[ k - 1][ j ] )
6:              INT[ k ][ j ] = INT[ k ][ j - 1 ] or INT[ k - 1 ][ j ] depending on what max() chose above
7:          else
8:              new List = {}
9:              for p = 1 to j - 1 do
10:                 sum = length(A_k) + LEN[ k ][ p ]
11:                 if ( sum ≤ j ) and ( INT[ k ][ p ](end) ≤ A_k(start)) then
12:                     List.add(sum)
13:
14:             LEN[ k ][ j ] = max_{0≤x<(List.length)} ( length(A_k), LEN[ k ][ j - 1 ], LEN[ k - 1 ][ j ], List[ x ] )
15:             // Break ties in max by choosing the interval with the left-most right-endpoint (end)
16:             INT[ k ][ j ] = the interval chosen by max() above
```

$L1 - L2$: Build the arrays from top-down and left-to-right. j should traverse a distance equal to the right-most endpoint of all the intervals in the input collection A.

$L4 - L5$: If the length of the currently considered interval is longer than j, then we should set the cell

to the max length of the cell above or to the right. Please consider any out-of-bounds scenarios as returning 0 or NULL.

$L7$: Otherwise, the length of the currently considered interval (A_k) is of length j or shorter.

$L8 - L12$: Create a bucket (*List*) to store some values in. We want to iterate through the values in LEN of the current row and add in the length of the currently considered interval (A_k) in order to sum all of the partial (and possible) output collections. We then check to see if this potential collection is valid before adding the sum of A_k to the *List*. In order for the sum to be valid, it must be of length j or less and it must not have any overlaps between A_k and the other intervals.

$L14 - L16$: Here we consider the maximum value of the length of the interval A_k itself, the max length of the cell above, the max length of the cell to the left, and the max length of all possible collections in *List*. We want to break ties by choosing the interval with the left-most right-endpoint. Finally, we want to assign to the INT array the last interval in the possible output collection chosen by max.

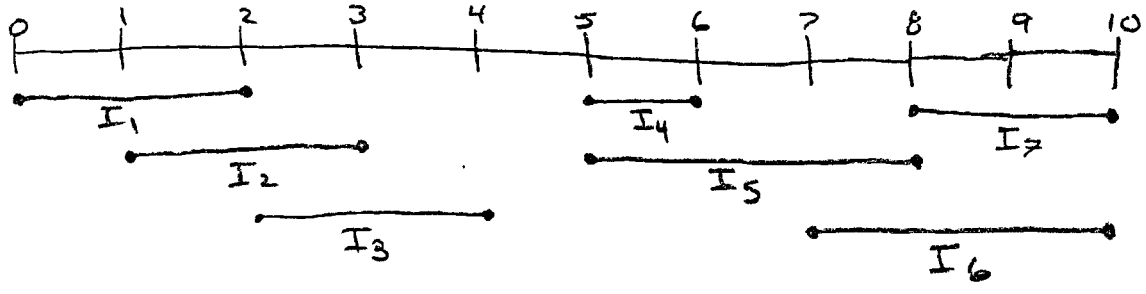
Obviously, this algorithm takes a few liberties with how it assigns values to the arrays – however these sort of details would best be saved for the actual implantation of the algorithm.

Ultimately, this algorithm provides our maximum sum in the last row, last column of the LEN array. From here, we can reference the last row, last column of the INT array to get our last entry in the output collection C. We then follow the intervals from right-to-left in the last column and add all the intervals to C which do not overlap ($A_k(start) \geq A_{k-1...}(end)$).

An example (and example backtrace for retrieving the solution) is provided on the next page:

Problem 10 (a) Example Trace and Solution

Input: $A = \{(0,2), (1,3), (2,4), (5,6), (5,8), (7,10), (8,10)\}$



INT

i \ j	1	2	3	4	5	6	7	8	9	10
1	NULL	A_1	A_1	A_1	A_1	A_1	A_1	A_1	A_1	A_1
2	NULL	A_1	A_1	A_1	A_1	A_1	A_1	A_1	A_1	A_1
3	NULL	A_1	A_1	A_3	A_3	A_3	A_3	A_3	A_3	A_3
4	A_4	A_1	A_1	A_3	A_4	A_4	A_4	A_4	A_4	A_4
5	A_4	A_1	A_5	A_3	A_4	A_4	A_5	A_5	A_5	A_5
6	A_4	A_1	A_5	A_3	A_4	A_4	A_5	A_6	A_6	A_6
7	A_4	A_1	A_5	A_3	A_4	A_4	A_5	A_6	A_7	A_7

output: $C = A_1, A_3, A_5, A_7$

length(C) = 4

LEN

i \ j	1	2	3	4	5	6	7	8	9	10
1	0	2	2	2	2	2	2	2	2	2
2	0	2	2	2	2	2	2	2	2	2
3	0	2	2	4	4	4	4	4	4	4
4	1	2	2	4	5	5	5	5	5	5
5	1	2	3	4	5	5	7	7	7	7
6	1	2	3	4	5	5	7	8	8	8
7	1	2	3	4	5	5	7	8	9	9

Problem 10 (b) Pruning

Input: n intervals $I_1, ..., I_n$ over the real line sorted in increasing order of their left endpoint.

Output: Collection C of non-overlapping intervals such that the sum of the lengths of the intervals in C is maximized.

Note: The hint mentions that we should consider the input in increasing order of their left endpoint. I included this constraint in the definition of the input.

Pruning Rules:
1. Prune any nodes that contain overlapping intervals. These nodes are invalidated by the expectation of the output (and would continue to be invalid in all descendants), thus are safe to prune.
2. If two nodes have the same total length at the same level, prune the one that contains the interval with the right-most endpoint (the interval that ends the latest). Any combination of intervals that end earlier, of the same length, would allow for more options without overlapping in the future – ultimately, anything that could be added to the right-most endpoint interval in the future can be added to the ones we are keeping. Thus they are safe to prune.

Algorithm: Algorithm Max. Interval Sum (MIS) is described below.

```
1:  MIS[ 0 ][ 0 ] = 0
2:
3:  for a = 0 to n do
4:      for b = 0 to I_n(endpoint) do
5:          if MIS[ a ][ b ] is defined then
6:              // Left-hand child
7:              MIS[ a+1 ][ b ] = max( MIS[ a ][ b ], MIS[ a+1 ][ b ] )
8:
9:              // Right-hand child
10:             if no overlap between MIS[ a ][ b ] and I_a then
11:                 MIS[ a+1 ][ b+length(I_{a+1}) ] = max(MIS[ a+1 ][ b+length(I_{a+1}) ], MIS[ a ][ b ]+length(I_{a+1}))
```

Explanation:
$L1$: Initialize the array

$L3 - L4$: Build an array with the levels a and sum lengths b. The entire solution will be contained within the bounds from 1 to the right-most endpoint of the intervals.

$L5$: If the current cell is defined, then "branch" children off of it.

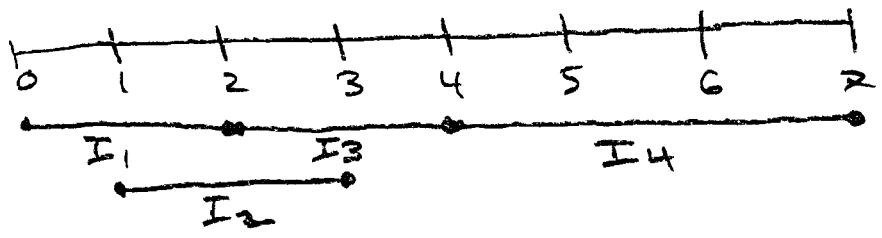
$L7$: The left-hand child, choose the max between the current cell and the cell below. Update the cell below.

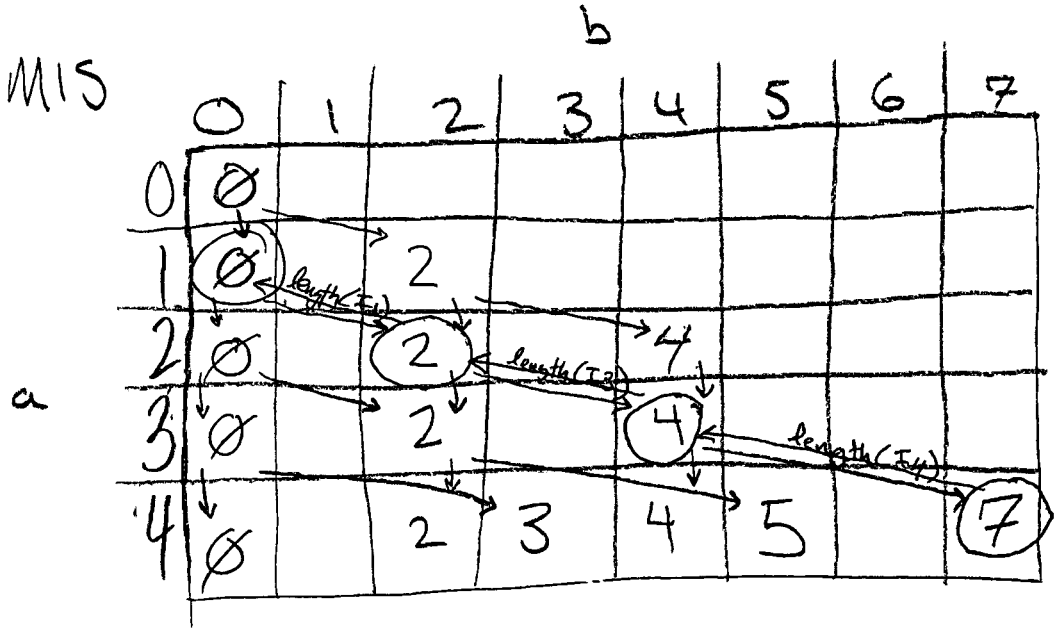
$L10 - L11$: L10 is the implementation of the first pruning rule (no overlaps allowed). L11 is the implementation of the second pruning rule.

At the end, we can easily backtrace this problem to find our solution. An example is provided on the next page.

Problem 10 (b) Example Trace and Solution

Input: $\{(0,2), (1,3), (2,4), (4,7)\}$



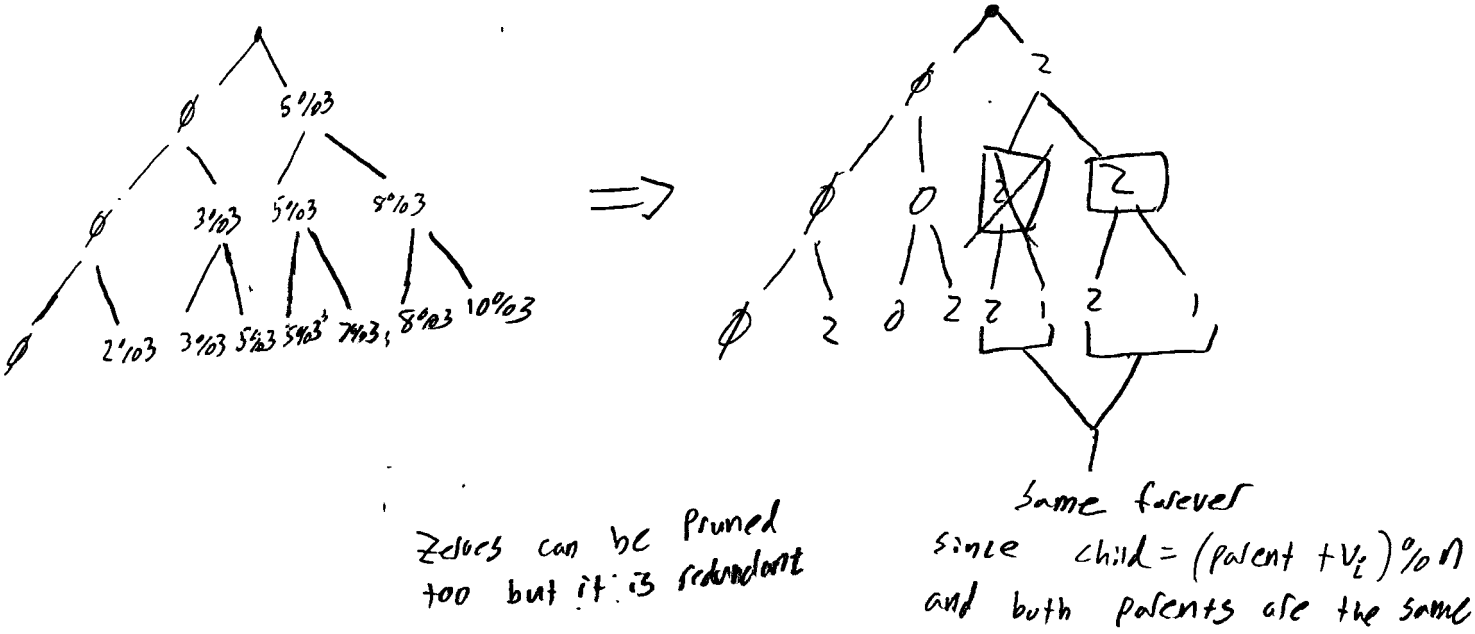


output: $C = I_1, I_3, I_4$

length(C) = 7

Problem 14

First consider an example tree with input $v = \{5, 3, 2\}$. Note that each node can be considered in two ways: a sum to which a module will eventually be applied, and the single value which results from that operation. We construct both trees as follows.

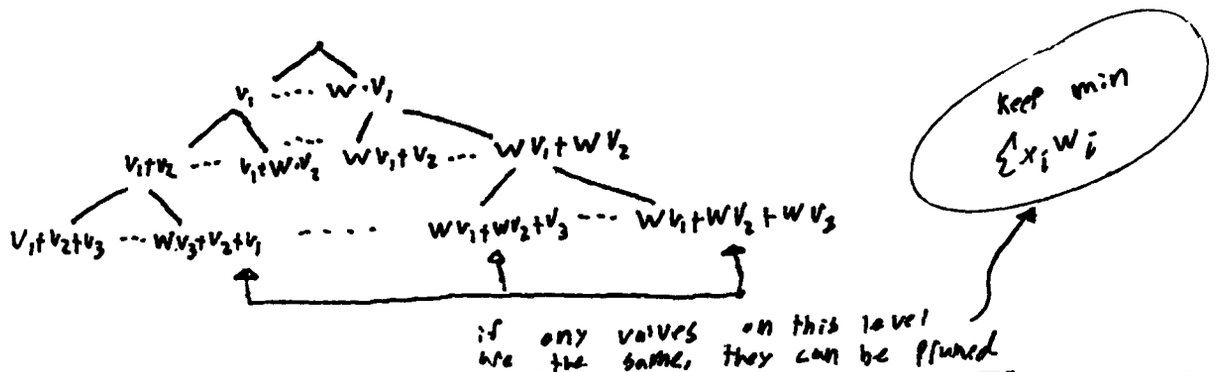


Our pruning rule will be that if any two nodes are equal on a given level, one can be pruned arbitrarily. This follows because if the modulus at the node is the same, the child sums will also be the same.

```
1: T[][];
2: T[0][0] = 0
3: for 0 ≤ i ≤ n do
4:     for 0 ≤ j ≤ n do
5:         if T[i][j] is defined then
6:             T[i + 1][j] = max(T[i + 1][j], T[i][j])
7:             if (T[i][j] + v_i) mod n ≠ 0 then
8:                 T[i + 1][(j + v_i) mod n] = 1
```

I think there may be re-construction issues regarding the sequence $x_1, x_2, ..., x_n$ that was used due to indexing on the modulus (which results in a very small table). I believe the pruning rules are valid, however, and the existence of a solution can be found by considering the $(L \mod n)^{th}$ column and looking for a 1. Further reconstruction may or may not be possible by considering the level at which each 1 is found.

Problem 15



Each node can have up to W children (since x could range from 0 to W and $\sum_{w=1}^{n} x_i w_i$ could feasibly be $\leq W$ up to that level in the tree).

Clearly, if any of these node's weight sums ($\sum_{w=1}^{n} x_i w_i$) are greater than W, they can be pruned. This is our first pruning rule. Additionally, if at a given level two node's corresponding value sums ($\sum_{w=1}^{n} x_i v_i$) are the same, then you can prune the one with a larger weight sum ($\sum_{w=1}^{n} x_i w_i$). This is our second pruning rule. In the tree this information is not all available, but we can embedded it into our dynamic program by using one (the value sum) as the indexes and another (weight sum) as the mapped value.

```
1: T[][];
2: T[1][1] = 0
3: for 1 ≤ i ≤ n do
4:     for 1 ≤ j ≤ n + W do
5:         for 1 ≤ x ≤ W do
6:             if T[i][j] is defined AND (T[i][j] + w_i x) ≤ W then
7:                 T[i][j + v_i x] = min(T[i + 1][j + v_i x], T[i][j] + w_i)
```

Not quite right, since sum of values could be much larger than W

A short note on efficiency:

$(n+w)^3 \geq n(n+w)^2 \geq n^2 w$ so the given algorithm is polynomial in $(n+w)$.

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.

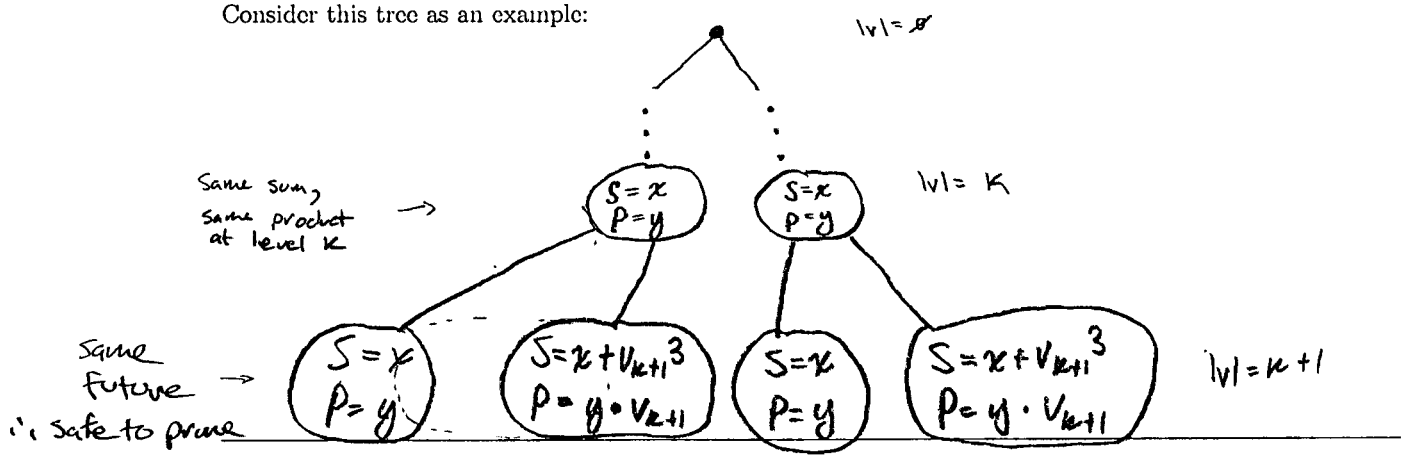
Pruning Rule: If two nodes on the same level have the same sum ($\sum$) and same product ($\prod$), prune either.

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

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:

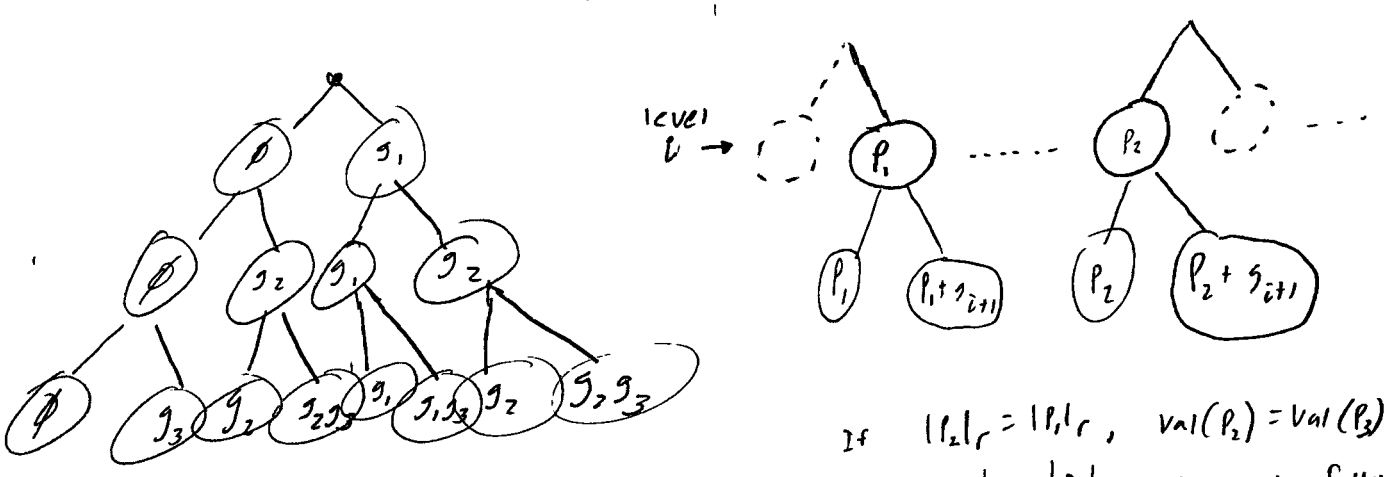


Problem 17

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:

```
1: T[n][n][n][L]
2: T[0][0][0][0] = 1
3: for i = 1 to n do
4:     if t_i is r then
5:         r_i = 1
6:         e_i = 0
7:     else
8:         r_i = 0
9:         e_i = 1
10:    for r = 1 to n do
11:        for e = 1 to n do
12:            for v = 0 to L/2 do
13:                if T[i][r][e][v] is defined then
14:                    T[i+1][r][e][v] = 1
15:                    if v + v_i < L/2 then
16:                        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 $O(\frac{1}{2}n^3L)$ which we consider to be polynomial in $n + L$ since $(n+L)^4 \geq (n+L)^3L \geq n^3L$.

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 Program?

Problem 19 b/c

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

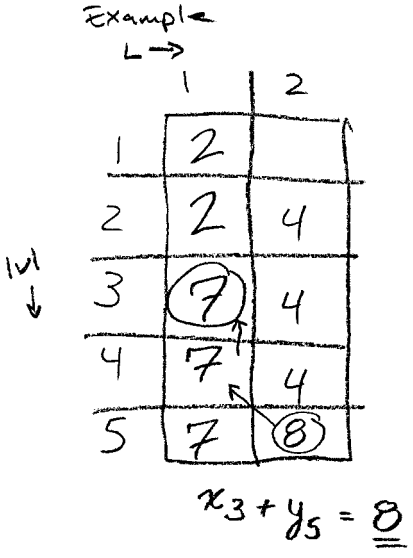
Pruning Rules:
1) If the length of the string at any given node is greater than k, prune that node.

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.

```
1:  for j = 1 to n do
2:      if T[ j ] == P[ 1 ] then
3:          MAC[ j ][ 1 ] = C_j
4:
5:  for lvl = 1 to n do
6:      for L = 1 to k do
7:          if MAC[ lvl - 1][ L ] is defined then
8:              //left-child
9:              MAC[ lvl ][ L ] =
10:             max( MAC[ lvl ][ L ], MAC[ lvl - 1 ][ L ] )
11:
12:             if L + 1 is in bounds and T[ lvl ] == P[ L ] then
13:                 //right-child
14:                 MAC[ lvl ][ L + 1 ] =
15:                 max( MAC[ lvl ][ L + 1 ], MAC[ lvl -1 ][ L ] + C_lvl )
```



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?

2/12

CS 1510
Algorithm Design

Dynamic Programming

Problems 18, 20, and 22

Due Wednesday September 24, 2014

Buck Young and Rob Brown

62.6% 67/107

Problem 18

Input: Ordered list of n words and an ideal line-length L. With the length of the ith word being w_i.

Output: A layout of lines (none of which can extend beyond L) which minimizes the total penalty. Total penalty is defined as the maximum of all line penalties (the line penalty for a line of length K is L - K).

Pruning rules:
1) If any node creates a line longer than L, prune it

2) If any nodes at the same level have the same length (therefore the same words on the line), prune all but the minimum total penalty.

Algorithm:

```
1: // Initialize multi-dimensional array of size n+1 by L+1 to positive infinity
2: A = new array(n+1)(L+1)
3: A[ * ][ * ] = ∞
4:
5: for lvl = 0 to n - 1 do
6:     for t = 1 to L do
7:         // Initialize this levels penalty based on the word by itself on a line
8:         A[ lvl + 1 ][ w_lvl+1 ] = L - w_lvl+1
9:
10:        // Set current cell to be current min in row ("left child")
11:        A[ lvl + 1 ][ t ] = min( A[ lvl + 1 ][ t ], A[ lvl + 1 ][ t - 1 ] )
12:
13:        // Consider if current word was added to the line above ("right child")
14:        if t - w_lvl+1 is in bounds && A[ lvl ][ t - w_lvl ] - w_lvl+1 ≥ 0 then
15:            A[ lvl + 1 ][ t ] = min( A[ lvl + 1 ][ t ], A[ lvl ][ t - w_lvl+1 ] - w_lvl+1 )
```

2

What is A supposed to hold?

The last line may not be the worst.

} You must include all words...

$L2 - L3$: This initializes a multi-dimensional array with indices 0 to n and 0 to L and sets all cells to have a value of positive infinity.

$L5 - L6$: From top to bottom and left to right

$L8$: First, in order to fill out the next line, consider the word for that level. Go to the cell which matches the length of this word and put in the value as if this word were on a line by itself.

$L11$: Now, iterate through each cell in the row and fill in a value based on the minimum between itself and its left neighbor.

$L14 - L15$: If we can complete the next operations, lets do them – note we must be wary of being out of bounds to the left and we also don't want to fill in a value less than 0. At any given cell, we want to consider the minimum between what is already there and what the possible penalty could be if we added

this word to the line above. The if statement checks to make sure we would not extend beyond L (otherwise we would be in the less than zero range).

Backtracing: This will give us a table where our last-line penalty is in the lower rightmost cell. From here, we can backtrace to find our layout which minimizes the total penalty. In order to do this, we head up a level and back by the length of the word. By doing this we can determine if our word was added to the line above or not. If it was not, simply write the word down as the last line. If it was, continue the process until you have a completed line. Once you have a completed line, head up a level and start in the lower-rightmost cell as if this level was the lowest boundary of the array. This lower-rightmost cell's value is the line penalty for the line we are about to construct. Continue with the instructions until you have all the lines in the layout. An example table and backtrace is provided below.

Example Input: "THIS IS A GROUP WORDS", $L = 6$

t

lvl	0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞	∞
1	∞	∞	∞	∞	2	2	2
2	∞	∞	4	4	4	4	(0)
3	∞	5	5	3	3	3	3
4	∞	∞	∞	∞	∞	1	(0)
5	∞	∞	∞	∞	∞	1	(1)

$p_{L1} = 0$ (can add to line above)

$p_{L2} = 0$ (can add to line above)

$p_{L3} = 1$ (cannot add to line above)

Generated Output:
($p = 0$) THIS IS
($p = 0$) A GROUP
($p = 1$) WORDS
($p_{total} = 1$)