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.

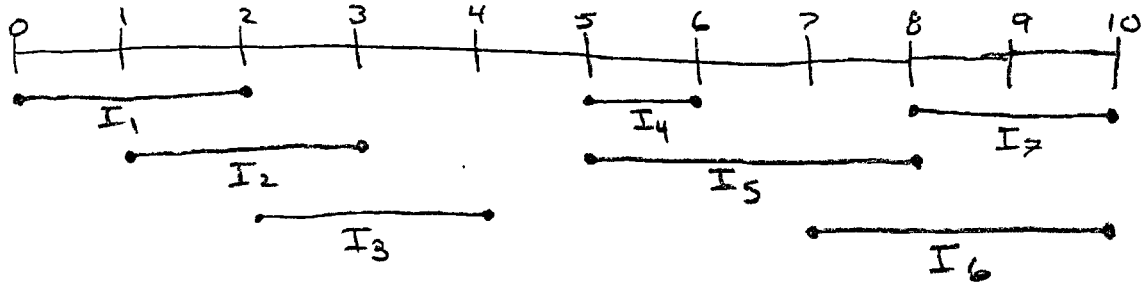
Obviously, this algorithm takes a few liberties with how it assigns values to the arrays – however these sort of details would best be saved for the actual implantation of the algorithm.

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.

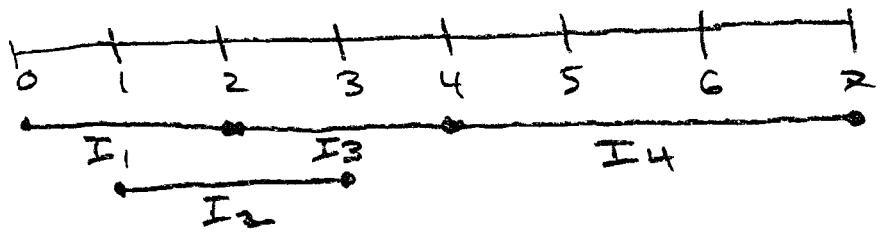
$L7$: The left-hand child, choose the max between the current cell and the cell below. Update the cell below.

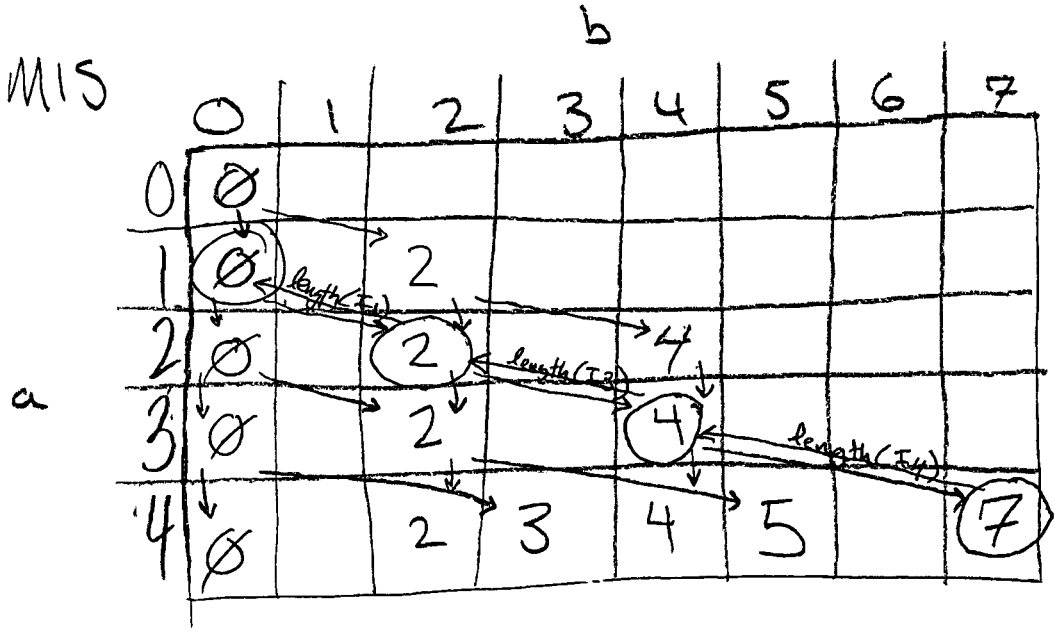
$L10 - L11$: L10 is the implementation of the first pruning rule (no overlaps allowed). L11 is the implementation of the second pruning rule.

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



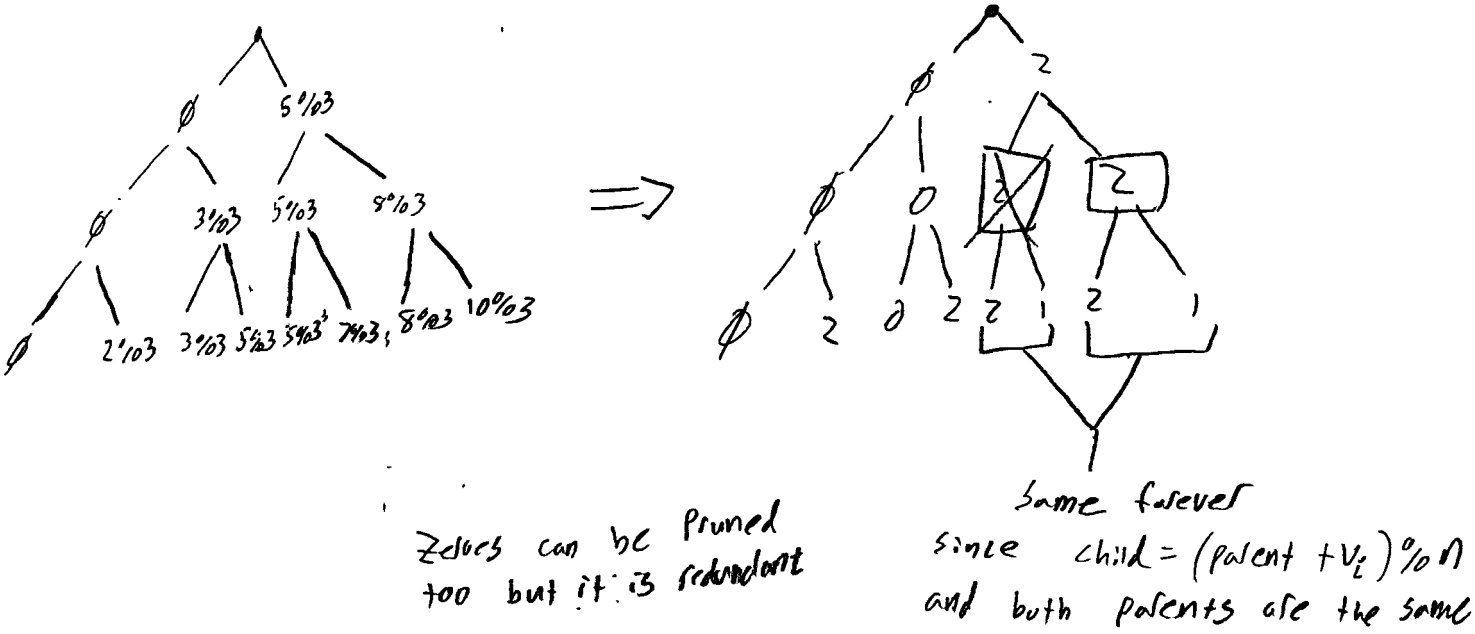


output: $C = I_1, I_3, I_4$

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.

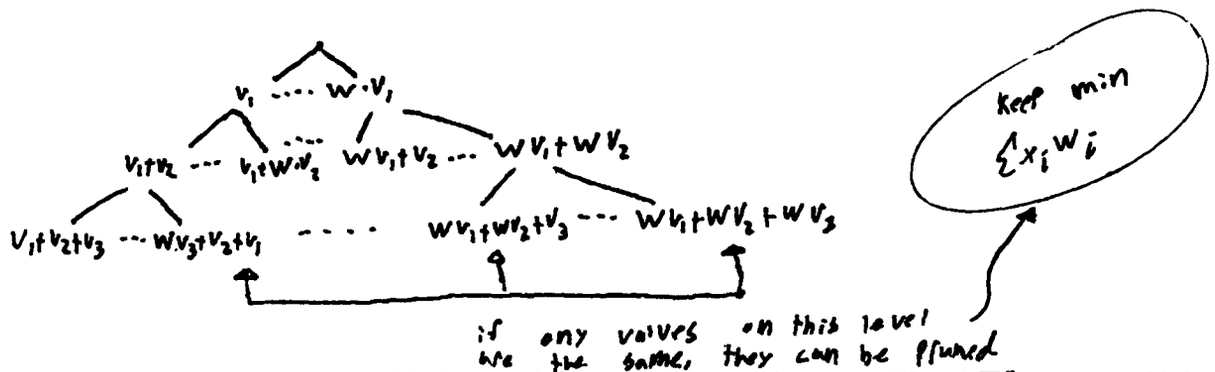


Our pruning rule will be that if any two nodes are equal on a given level, one can be pruned arbitrarily. This follows because if the modulus at the node is the same, the child sums will also be the same.

```
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 $(n+w)$.