Kelsey Meis meis@umn.edu
Section 001
Assignment 4
Due Nov 2

15.4 - ~~2,5~~ ; ~~16.1 - 2,3~~ ; ~~16.2 - 4,5~~ ; ~~16.3 - 2,5~~

15.4 2) Reconstruct-LCS(c, X, Y, i, j)

```
lcs = []
while c[i, j] ≠ 0
    if Xi == Yj
        lcs.add(Xi) // add to beginning of array
        i--
        j--
    else if c[i-1, j] == c[i, j]
        i--
    else
        j--
return lcs
```

The runtime will be m+n because either i or j or both are decremented each iteration.

5) 1. Build an n+1 x n+1 table, populating the bottom half triangle with 0, and a buffer row and column with 0 as well:

	0	X_1	X_2	X_3	X_4
0	0	0	0	0	0
Y_1	0	0			
Y_2	0	0	0		
Y_3	0	0	0	0	
Y_4	0	0	0	0	0

← here, x and y are both the same set, duplicated to allow comparisons

2. Fill in the rest of the table as follows:
   - if $X_i > Y_j$, $c[i,j] = c[i-1, j-1] + 1$
   - else if $c[i-1, j] > c[i, j-1]$
     $c[i,j] = [i-1, j]$
   - else $c[i,j] = c[i, j-1]$

(Note: this is similar to how the table is filled for LCS)

3. -Keep track of the highest number stored in the table.

3. Starting from the lower right corner of the table, work your way back diagonally until the highest number is reached (which was stored in the step 2.

This will be the last number in the optimal solution. Add it to an array.

- Keep going with this process, adding to the array every number that is both the highest number among those with its same table value and does not exceed the last value in the solution array

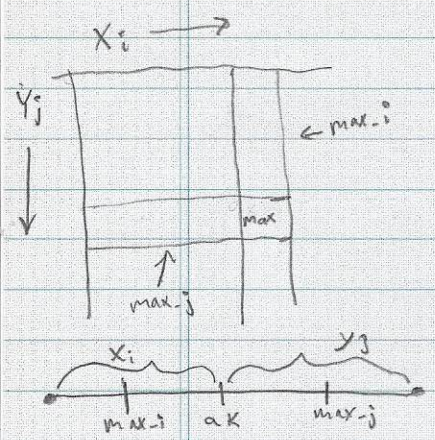
16.1 2) 1 - make 2 arrays $X = S$, $Y = S$-reversed

2. Nested iteration of $Y_j \to n$ and X_i to n

a. - iterate from j-1 down to 1 to find value of $C[i,k]$ that is largest *
→ store $C[i,k]$ and k (max_j and max_j_index)

b. - iterate from i-1 down to 1 to find value of $C[k,j]$ that is largest *
→ store $C[k,j]$ and k (max_i and max_i_index)

c. - $C[i,j].val = max\_i + max\_j + 1$
$c[i,j]$, $max\_j\_index = max\_j\_index$
$C[i,j].max\_i\_index = max\_i\_index$
if $C[i,j].val >$ Stored_max_overall
stored_max_overall $= C[i,j].val$

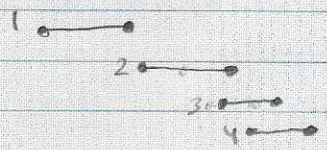


3. Trace back through the table, starting at position where stored_max_overall is. That will be the pivot of the solution array. Append max_i elements before the the pivot and max_j elements after. Follow the max_i and max_j indexes, inserting the subsequent max_i and max_j elements to the left and right of those pivots until the array is of the same length as the stored_max_overall

* for 2-a, the element Y_K must have a starting time greater than Y_j's finishing time in order to "qualify" for max
for 2-b, the element X_K must have a finishing time less than X_i's starting time in order to "qualify" for max

16.1

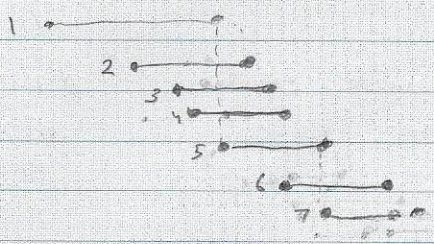
3) - Selecting activity w/ least duration



i	1	2	3	4
s	0	4	8	9
f	4	9	11	13

Assuming the first activity is chosen, the next activity would be 3, being the shortest among 2, 3, and 4. Since 3 overlaps with 2 & 4, the number of activities would be 2, which is less than the optimal number of 3.

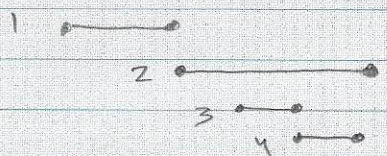
- Selecting activity that overlaps with fewest other activities



i	1	2	3	4	5	6	7
s	0	2	3	4	5	7	8
f	5	6	7	7	8	10	11

Assuming event 1 is selected, 7 would be the qualifying event that has the least overlaps, overlapping with 6 only. If 7 is selected, the number of activities would be 2. Here, selecting 5, then 7 would be optimal, even though 5 overlaps with 4 other activities

- selecting for earliest start time

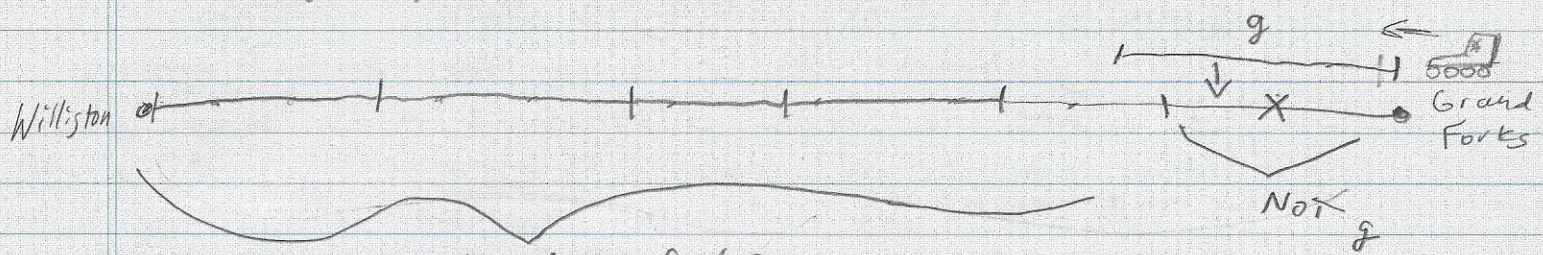


i	1	2	3	4
s	0	3	4	5
f	3	8	5	7

If 1 is the first activity, this method would choose 2 next, leading to only 2 activities where instead choosing 3 and 4 would make for the optimal 3 activities

16.2

4) I would use the greedy choice of whatever stop is furthest away while still being less than or equal to m. To prove this will yield an optimal solution, suppose we have an optimal solution that does not contain this greedy choice:



optimal # of stops, each interval $\geq m$

Replacing NOT g with g does not break the optimal structure, since g will be a greater length than the Not g interval, and the professor will still be able to make it to the next stop in his optimal set of stops. It is also intuitively obvious that if you go further on the first leg, you'll have more possible second stops that are further along than if you had stopped earlier.

If we replace Not g with g, it will not break the optimal structure

- To illustrate the runtime, the pseudocode would be sth. like this:

Runtime

Greedy Water stops(m, S) where m = max dist before running out of water and S = set of stops - integers for dist from Grand Forks

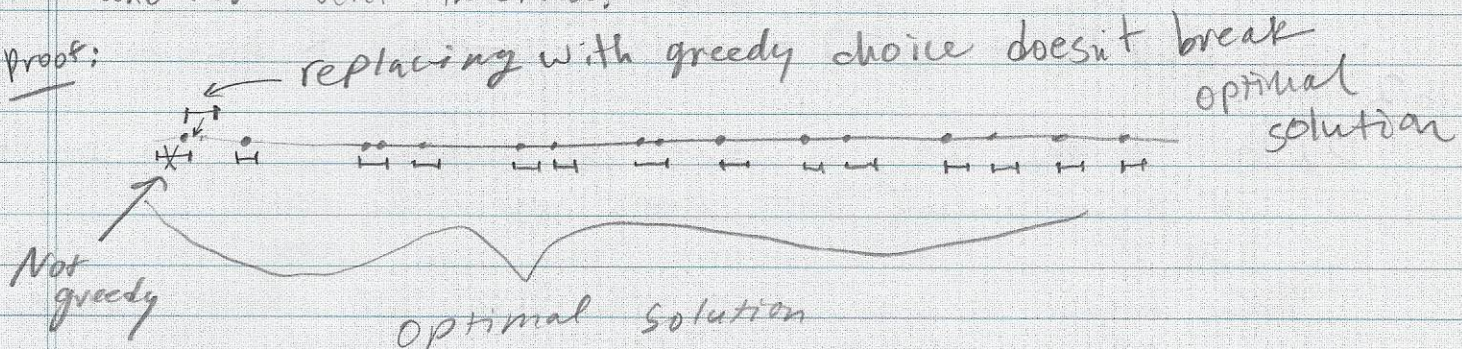
```
n = s.length
A = {a_1}
dist_traveled = 0
for t = 2 to n
    if s[t] - dist_traveled ≤ m
        t++
    else
        dist_traveled = s[t-1]
        A = A ∪ {a_{t-1}}
        t++
```

Each element of S is examined once, so runtime is $\Theta(n)$

16.2

5) Place a unit interval in such a way that its starting point is X_1. Continue to X_2, X_3 to X_n, only placing the unit interval at a point if it won't overlap with another unit interval.

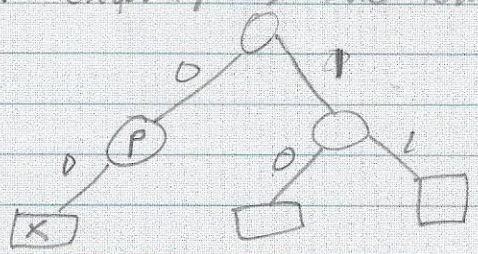
Proof:



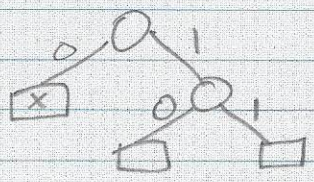
16.3

2) A binary tree that is not a full binary tree can be rearranged to form a full binary tree, and the result will be more optimal

For example, the following incomplete binary tree:



can be rearranged to make the full binary tree:



Since there is no need for a node (p) where the leaf X may be placed directly beneath the root. This would decrease X's depth, reducing the overall cost.

$B(T) = \sum_{c \in C} f(c) d_T(c)$ ⟸ incomplete binary tree

if we set $d_T'(x)$ to be the new depth of x

where $d_T'(x) < d_T(x)$

$B(T') = B(T) - d_T(x) + d_T'(x) < B(T)$

16.3

5) Due to the way in which the Huffman code is built, it is guaranteed that leaves nearer the root will have a greater frequency than the leaves below them. the higher a leaf is in the tree, the lower its depth, and consequently the shorter its codeword.