CSCI 5421 - Assignment 6

Email	neis@umn.edu
Name:	Kelsey Neis

16.1-2

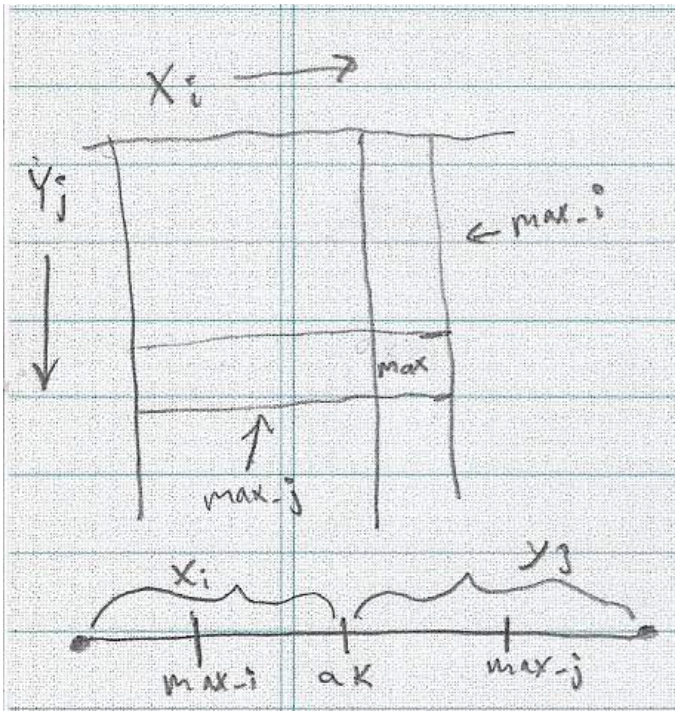
1. Make 2 arrays x = S, y = S-reversed
2. Nested iteration of Y_j → n and X_i → n

   a. Iterate from j - 1 down to 1 to find value of C[i, k] that is largest* and store C[i, k] and k (max-j and index of max-j)

   b. Iterate from i - 1 down to 1 to find value of C[k, j] that is largest* and store C[k, j] and k (max-i and index of max-i)

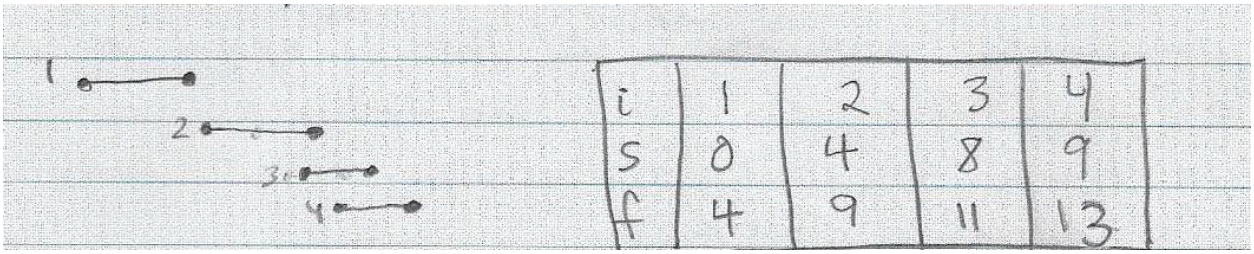
   c. C[i, j].value = max-i + max-j + 1
   C[i, j].max-j-index = max-j-index
   C[i, j].max-i-index = max-i-index
   if C[i, j].value > stored_max_overall, then:
   stored_max_overall = C[i, j].value
3. Trace back through the table, starting at position where stored_max_overall is. That will be the pivot of the solution array. Append max-i elements before the 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.



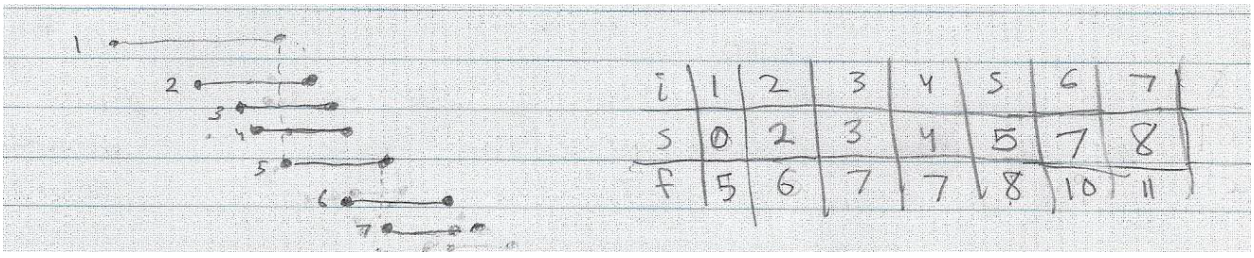
16.1-3

Selecting activity with least duration:



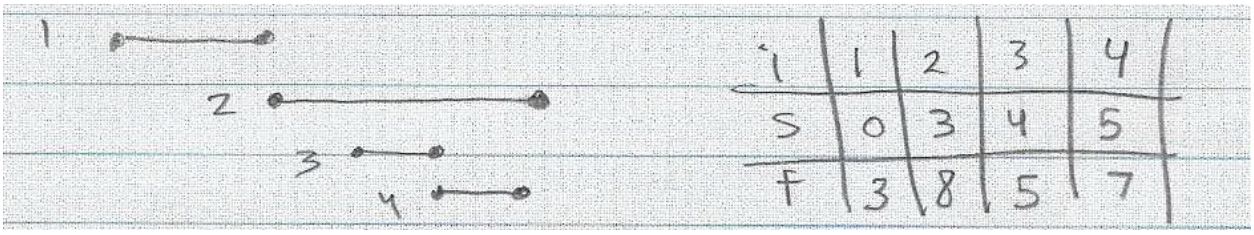
Assuming the first activity is chosen, the next activity would be 3, being the shortest among 2, 3, and 4. Since 3 overlaps with 2 & 4, the number of activities would be 2, which is less than the optimal number of 3.

Selecting activity that overlaps with the fewest other activities:



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, event though 5 overlaps with 4 other activities.

Selecting for earliest start time:

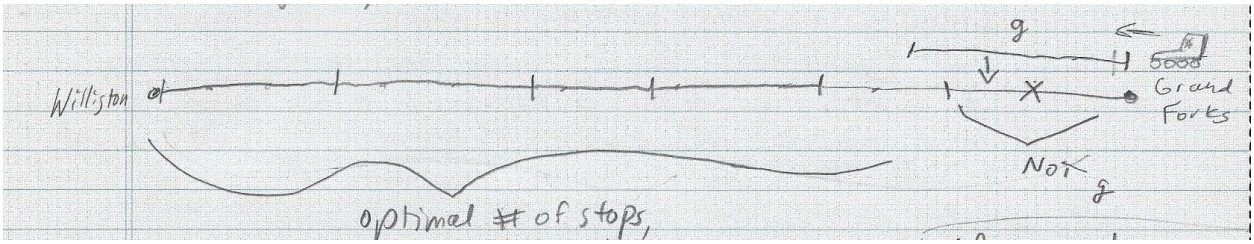


If 1 is the first activity, this method would choose 2 next, leading to only 2 activities where instead choosing 3 and 4 would make for the optimal 3 activities.

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 the greedy choice:



Replacing not-g with g does not break the optimal structure, since g will be of greater length than not-g, 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 steps that are further along than if you had stopped earlier.

Runtime:

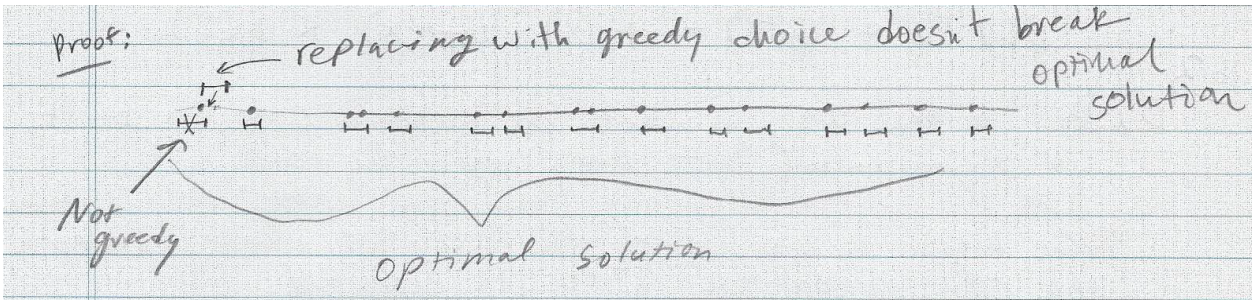
To illustrate the runtime, the pseudocode would be something like this:

```
Greedy-water-stops(m, S): //where m = max dist before running out of water
1  n = S                  //and S = set of distances from stops to Grand Forks
2  A = {a}
3  dist_traveled = 0
4  for t = 2 to n:
5    if S[t] - dist_traveled <= m:
6      t++
7    else
8      dist_traveled = S[t-1]
9      A = AU{a_t-1}
10     t++
```

Each element of S is examined once in lines 4-10, so the runtime is $\theta(n)$.

16.2-5

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



16.2-7

Proposed Greedy Choice: pair the highest b element with the highest a element.

Prove Greedy Choice Property holds:

1. Suppose there is a most profitable solution which has a set of a and b that does not contain the pair g, which is the highest a paired with the highest b
2. Then, replace the non-greedy choice with the greedy choice. In this case, this would involve the following:

   a. $P = \prod_i^n a^b \setminus g$

   b. $g = a_{max}^{b_{max}}$

c. Somewhere in P, we have: $a_{notmax}^{b_{max}} * a_{max}^{b_{notmax}}$

d. Set $P' = P$, replacing $notg$ with g, we get $a_{notmax}^{b_{notmax}} * a_{max}^{b_{max}}$ somewhere in P'

e. The increase of $a_{notmax}^{b_{max}}$ to $a_{max}^{b_{max}}$ will exceed the loss created by changing $a_{max}^{b_{notmax}}$ to $a_{notmax}^{b_{notmax}}$, since the higher exponent carries a higher weight than the lower exponent. So, $P' > P$

3. Since replacing $notg$ with g results in a higher P, the original assumption that P was optimal is contradicted. Therefore, the optimal solution contains g

Runtime:

Using Mergesort to sort A and B in ascending order, the runtime becomes the runtime for Mergesort, since it is the most expensive operation in the algorithm, which is $\theta(nlgn)$. The rest of the algorithm is linear in n because it is simply then a matter of multiplying the elements of A with the elements of B.

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.

1. Suppose there is an optimal Huffman code that is not represented by a full binary tree.
2. Then, modify it to be a full binary tree by removing leaves which do not have siblings, so that their parents are the leaves instead.
3. Since the leaves did not have siblings, this change would not break the Huffman code, i.e. it would remain prefix-free.
4. However, the change *would* result in a more optimal Huffman code, because some codenames would be reduced to smaller bitstrings. This contradicts the original assumption that the code was optimal when there wasn't a full binary tree.
5. Therefore, an optimal Huffman code must be represented by a full binary tree.

16.3-6

```
treeStructure(g, i, d):
1    code_map = []
2    for i= 0 to g.length:
3        if g[i]['l'] == null:
4            code = g[i]['dir']
5            j = i
6            while j > 0:
7                j = g[j]['parent']
8                if j > 0:
9                    code = g[j]['dir'] + code
10           code_map.add({ 'code': code, 'num': i})
11       else:
12           g[g[i]['l']]['parent'] = i # set parent
13           g[g[i]['l']]['dir'] = 'l' # set l
14           g[g[i]['r']]['parent'] = i # set parent
15           g[g[i]['r']]['dir'] = 'r' # set r
16   return code_map
```

The outer for loop will execute 2n - 1 times, as there are that many nodes. The while loop will execute nlgn times, since the code traverses up the tree via the parents, which corresponds to the height, lgn. This is done for each letter to get the corresponding code.

16.3-7

```
HuffmanTernary(C):
1  n = |C|
2  Q = C
3  if (2n - ceil(n/2) - 1) % 3 > 0:
4    allocate new node m
5    m.freq = 0
6    Insert(Q, m)
7    n++
8  for i = 1 to n - 1:
9    allocate a new node z
10   z.left = x = EXTRACT-MIN(Q)
11   z.right = y = EXTRACT-MIN(Q)
12   z.middle = m = EXTRACT-MIN(Q)
13   z.freq = x.freq + y.freq + m.freq
14   INSERT(Q)
15 return EXTRACT-MIN(Q)
```

In order to make the code optimal, the solution must create a perfect tree. To make a perfect ternary tree, with each node having 3 children, the number of

nodes below the root must be divisible by 3. The total number of nodes is floor(3n/2), since for every leaf, we're adding a node, but this is shared with two other leaves (or nodes), so subtracting the root, you get floor(3n/2) - 1. So, in lines 3-7, if the result is not divisible by 3, a placeholder node must be created with a frequency of 0. This ensures that the placeholder will be at the bottom of the tree, and will not break the optimal structure.

Proof:

1. The proof is similar to the proof for the binary variant:
2. Say a tree T is optimal for C
3. T' = C' = C \ {x, y, m}U{z}
4. f(z) = f(x) + f(y) + f(m)
5. B(T) = B'(T) + f(z)
6. Assume T' is not optimal
7. Change T' to make optimal for C' and call it ~T'
8. B(~ T') = B(~T') + f(z)
9. B(T') > B(~T')
10. B(T) > B(~T)
11. This leads to a contradiction, because the original assumption was that B(T) was optimal. So the assumption that the substructure is not optimal is false.