Parth Ray

Lab 7-1

4. Knapsack with Repeats

   a. Find a counterexample that shows that taking **most valuable item** first will not maximize the value placed in the knapsack.
      If my most valuable item is (50, 3) and the sack has a total capacity of 3, we can only take one and we will get a total value of 50. But say we have another item (30, 1). Here we can take 3 of this other item and get a total value of 90.
   b. Find a counterexample that shows that taking **the smallest item** first will not maximize the value placed in the knapsack
      If my smallest item is (1, 1) and the sack can hold 3, then we have a total value of 3. But if we have another item (50, 3) then if we take with item, we have a total value of 50.
   c. Find a counterexample that shows that taking the **item with the highest value/weight ratio** first will not maximize the value placed in the knapsack
      If the item with the highest value to weight ratio is (10, 2) and we have a total capacity of 3. Then the total value would be 10. Then if we have an item (4, 1) we would get a total value of 12.

A. Let V(n, W) represent the maximum value for a set of n items and a capacity of W where items can be repeated.
B. V_{Knap} (n, W)= max{V(n-1, W), V(n, W-w_n) + v_n}
C. We just need a matrix of size n + 1 x W + 1 and initialize the zero column and zero row with zeros to make algorithm easier. Each entry will represent the maximum value for that n items and w capacity while allowing for repeated items.
D. FillTable()

```
        For i from 0 to n -> V[i][0] = 0
        For i from 0 to w -> V[0][i] = 0
        For i from 1 to n
                For j from 1 to w
                        V[i][j] = max{V(i-1, j), V(i, j-wi) + vi}
        Return V;
```

E. TraceBack()

```
        Set[]
        J = w;
        For i from n to 1
                If V[i][j] == V[i - 1][j]
                        Continue;
                Else
                        While(V[i][j] % vi)
                                Set.append(i)
                                V[i][j] = V[i][j] – vi
                        J = j – wi
                        If j <= 0
```

Break;

Return set[]

F. Filling in the table takes O(m*n) time complexity.

5. Minimum Running Cost

A. Let P(n) be the optimal schedule and minimum cost for running the business for n months.
B. $P(n) = \min\{P(n_c), P(n_j) + M\}$, where M is the travelling cost and j is the city you will travel to in month i and c is the city you are currently in.
C. We will need a table of size n + 1. Each entry will present the minimum cost of running the business for those total months, i.
D. P(0) = 0;

   $P(1) = \min\{P(1_c), P(1_j)\}$;

   For i from 1 to n

   $P(i) = \min\{P(i_c), P(i_j) + M\}$;

   Return P;
E. Sched [];

   Prepend the city at P(n)

   For i from n to 1

   If ($P(i - 1) + i_c == P(i)$) // where c is the city you are in currently

   Prepend city c;

   Else

   Prepend city j;

   //make current city j and j be the old current city

   Return Sched
F. Filling in the table takes O(n) time since you have to go through the list of months only once.

Lab 7-2

2. Longest Common Substring Problem

A. Let LCS(i, j) be the function that represents the longest common substring of two strings of size i and j.
B. LCS(i, j) = if char at i and j match then LCS(i-1, j-1) + 1; else 0
C. We need a table of size i+1 x j+1. Each entry will represent the longest common substring of that location i, j.
D.

   I = length of 1st string + 1

   J = length of 2nd string + 1

   For x from 0 to i

   For y from 0 to j

If x == 0 and y == 0 then
    LCS(x, y) = 0;
Else if chars at x and y match
    LCS(x, y) = LCS(x – 1, y - 1) + 1;
Else
    LCS(x, y) = 0
Return LCS

E.
1. Find max value in matrix.
2. Prepend char of max value in matrix to array
3. Keep prepending char at loc of max [i – 1, j - 1] until you reach a value of zero at the location.
4. Return array

F. Filling in the table takes O(m*n) complexity because we fill out an m x n matrix.

3. **Firestones Profit**:

A. Let MP(n) be the max total profit at for a set of n restaurants.
B. $MP(n) = \max_{i<n \text{ and } (m(n) - m(i)) < k}\{MP(i)\} + p_n$
C. We need a table of size n. Each entry will represent the max profit up to and including the store at i.
D. $MP(1) = p_1$;
For i from 2 to n
    $MP(i) = \max_{j<i \text{ and}(m(i) - m(j))< k}\{MP(j)\} + p_i$;
Return MP;

E.
1. Find max value of MP and place idx into array
2. Profit = max value of MP
3. i= idx of max value of MP
4. Get the restaurant that has the biggest MP and is of at least k distance away from the max and its MP plus p_i is equal to Profit.
5. Place that restaurant into the array
6. Profit = MP restaurant from step 4
7. i = idx of restaurant from step 4
8. Repeat steps 4 – 7 until profit is equal to zero.
9. Return array.

F. Filling in the table takes $O(n^2)$ complexity because we fill out an array of size n once but have to find the max compatible restaurant for each restaurant which takes n steps as well.

4. **Change Making Revisited:**

A. F(k) represents whether we can make change for a value give denominations.
B. F(k) = if $F(k – d_j)$ == T over j where $d_j < k$ then T; else F
Base Case: F(0) = T
C. The table needs to be size k + 1. Each entry represents whether we can make exact change for the value k.
D. F(0) = 0;
For i from 1 to k

If (F(i - d_j) == T over j where d_j < i)
F(i) = T;
Else
F(i) = F;
Return F;

E. Skipped

F. Filling the table takes O(n) complexity because we go through the list of n items once.