CS 3340 Assignment 3
250757782
Linsheng Ding

1)
Show the longest common subsequence table, L, for the following two strings:
X = "skullandbones"
Y = "lullabybabies".
What is a longest common subsequence between these strings?

Longest common subsequence table:
Highlighted blocks show a bridge/diagonals

		s	**k**	**u**	**l**	**l**	**a**	**n**	**d**	**b**	**o**	**n**	**e**	**s**
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
l	0	0	0	0	1	1	1	1	1	1	1	1	1	1
u	0	0	0	1	1	1	1	1	1	1	1	1	1	1
l	0	0	0	1	2	2	2	2	2	2	2	2	2	2
l	0	0	0	1	2	3	3	3	3	3	3	3	3	3
a	0	0	0	1	2	3	4	4	4	4	4	4	4	4
b	0	0	0	1	2	3	4	4	4	5	5	5	5	5
y	0	0	0	1	2	3	4	4	4	5	5	5	5	5
b	0	0	0	1	2	3	4	4	4	5	5	5	5	5
a	0	0	0	1	2	3	4	4	4	5	5	5	5	5
b	0	0	0	1	2	3	4	4	4	5	5	5	5	5
i	0	0	0	1	2	3	4	4	4	5	5	5	5	5
e	0	0	0	1	2	3	4	4	4	5	5	5	6	6
s	0	1	1	1	2	3	4	4	4	5	5	5	6	7

Longest common subsequence based on table : **ullabes**

2)

Algorithm: ReconstructLCS(table, firstString, secondString, row, col)

Description: Reconstruct an LCS from a LCS table
Input: 5 parameters - table stores the lcs table, firstString and secString stores the two different strings, row, col are indexes and used to traverse matrix.
Output: The longest common subsequence string as a recursive print method

```
ReconstructLCS(table, firstString, secString, row, col):
 if table[row][col] == 0
     return
 if firstString[i] == secString[j]
     ReconstructLCS(table, firstString, secString, row - 1, col - 1)
     print(firstString[i])
 elseif table[row - 1][col] > table[row][col - 1]
     ReconstructLCS(table, firstString, secString, row - 1, col)
 else
     ReconstructLCS(table, firstString, secString, row, col - 1)
```

<u>**Time Complexity:**</u>
This will have a worst case scenario of O(n+m) time where n is length of first string and m is length of second string as it will traverse the lcs matrix of these 2 strings but only in a diagonal fashion or move around until it can traverse in a diagonal fashion until reaching a part of the table where there are 0s and thus never reach O(n*m) time complexity.

3) If you put the weight of the sack as n, and each bid i corresponds to an item of weight w_i and value v_i then you have a knapsack problem.This is a fractional version if bidders are willing to accept partial lots. If each bidder i is unwilling to accept fewer than w_i items then this is a 0-1 problem.

4) We can solve the word wrapping problem in a dynamic programming approach:

Given we have n words, we can construct a n * n matrix or a matrix of size n^2, then we will grab M or the max length of characters that can be on one line. We also need to be able to know the length of each word from 0 to n - 1, which we can calculate by moving from the first word to the nth word and storing the length of each word into a array which we can later use.

For the matrix, we will fill in the result from the cost function C(i,j) where i is the row and j is the column and calculate the the number of empty spaces to the power of 3 from fitting in the ith word to the jth word onto one line, if it is not possible to fill in the ith word to the jth word onto one line, we will put infinity into the ith row and jth column of the matrix. We keep doing this until we fill the whole matrix.

As well, we now initialize two arrays with the same length of n with the first array storing minimum cost and the second array storing the final result. Starting from n - 1 and moving to 0 of the first array, we have two indexs for the array for row n-1 and column n-1 that refers to row and column n-1, n-1 of the matrix and if this element is not infinity, we put it into our minimum cost array at index n-1 and keep doing this for index of the array decremented by 1 each time, if we input the minimum cost into the array, we also input the number of words that can be put onto one line, so for example, if index n-1 of the minimum cost array is 4 and the element in the same index in the other array is 5, we are saying that word 4 to 5 can fit on one line. However if the element in the matrix is infinity, we decrement until the row and column is not infinity or we split the two indexes in the minimum cost array to find the cost based on first index and second index decremented by 1 added to our previously calculated minimum cost and keep doing this until we reach the current index of the minimum cost array. We find the minimum of this and store it into our minimum cost array and indicate the number of words that can be fit onto one line in the results array. We keep doing this until we fill up both arrays.

After this is done, we then move from 0 to n -1 on the results array, index 0 of results array will have how many lines can be fit onto the first line and then we move onto the index that is referenced by the element in index 0 which lets say is 2 which means we look at index 2 of the array and add new words onto the next line and keep doing this until we have run out of words.

After getting the words into each line, we can calculate the cube of the number of whitespace of each line using our cost function and then sum that to get the minimum cost.

Because the biggest operation here is moving through the matrix and affects our program the most, thus the time complexity is O(n^2).

5)
We can modify the Knuth-Morris-Pratt Algorithm to help find the longest prefix string between two strings as it runs in O(n+m) time. Basically when finding each matching prefix, we keep track of the length and store the length and location of the longest prefix at each turn of the next table used in the Knuth-Morris-Pratt algorithm or update the length and location if there is a new longest prefix with a longer length than the one stored. However at the end, if we find that both strings are the same or that the first string contains the second string, we just return the found string as this is the longest prefix.

6)
We can modify Kruskal’s algorithm for finding a minimum spanning tree to look for a maximum spanning tree while also using a Union-find algorithm to check for cycles in the graph. Normally you would use a min-heap to sort the edges, but for this case you would use a max-heap to sort the edges into the right order and in this case also the max edge is taken out every time of the graph into the max-heap.

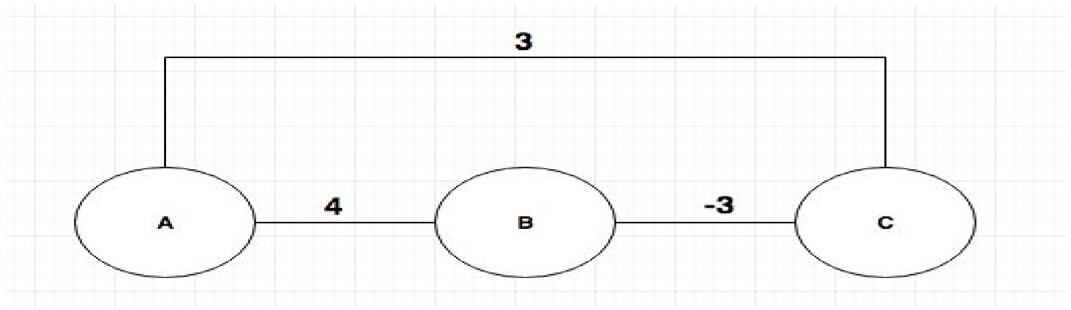
7)
Description
We can presume there are no negative edge weights as there cant really be negative bandwidth with a telephone network. Then we can answer this question by modifying Dijkstra’s algorithm

for finding minimum weight to instead look for the maximum bandwidth weight of a path between 2 centers and instead record the minimum bandwidth weight at each center instead of keeping track of a sum as we do in a Dijkstra's shortest path algorithm, we then output the maximized minimum bandwidth weight until we reach vertex b.

Steps/Algorithm:
We first set infinity as the weight to our starting vertex a, and all vertices to have weights of negative infinity so that we start with the vertex with the largest bandwidth weight which is our starting vertex a. For each vertex starting from a, we find the greatest minimum bandwidth weight which is the edge weight from the current vertex to the next reachable vertex that is greater than the rest of the other edge weights and put that as the recorded bandwidth weight of the reachable vertex. This would be less than infinity for vertex a at the start and assign that bandwidth weight to the next reachable vertex and move to that reachable vertex with maximum bandwidth weight. We then keep doing this until and recording the maximum bandwidth weight at each vertex until we reach vertex b and return the maximum bandwidth of the path between vertex a and vertex b.

Running time:
The running time is similar to a normal Dijkstra's algorithm, meaning that it is O(V log V) where V is the number of vertices in a graph.

8)
Dijkstra's algorithm doesn't work with negative edge weights as it is a greedy algorithm and given a graph such as the example below:



Consider the above graph which consists of vertices A,B and C with edges AB, BC, and AC, with edge weights of w(AB) = 4, w(BC) = -3, w(AC) = 3. Because Dijkstra is a greedy algorithm it would take path AC as 3 is less than 4 or path AC has less edge weight than AB but in reality the shortest path is AB to BC or A to C with the sum of the weights along that path being 1, therefore we have not found the shortest path in a Dijstra's algorithm implementation involving negative edge weights, proving that it does not work with negative edge weights.

9)
Given a weighted directed graph G with negative weights but no negative cycle, the all-pair shortest-path algorithm is still correct as it takes in all the paths between a pair of vertices in G.

Thus the weights for both a pair of nodes that are both connected to each other right next to each other or connected through more than one node will be considered. Thus if there is any negative edge weight it will be included when the all-pairs shortest-path algorithm includes all paths between two nodes in G unlike what is seen in question 8 or dijkstra's algorithm where dijkstra being a greedy algorithm would ignore the shortest path in a graph with negative edge weight.

10)
Description:
We can modify the Floyd-Warshall algorithm to find a cycle with minimum weight in a weighted directed graph G with with no negative cycles. We then use the Floyd-Warshall algorithm to figure out all the shortest paths where you record in a matrix all the paths from node 1 to the last node n in graph G and then go through those shortest paths to find a cycle with the smallest weight.

Steps:
After you run Floyd-Warshall and using all the shortest-paths recorded, you assign one variable called minimum which is set to infinity and thus for each pair of vertices recorded if the distance indicated on the matrix of row u and column v (where u and v are numbers that represent nodes) and the distance of row v and column u is less than the minimum, then assign those two distances summed together to find the new minimum and remember the pair of u and v. When you're done running through all the shortest-paths with the above algorithm you would have found the two nodes with a cycle with minimum weight.

Running time:
Because the largest part of the algorithm deals with the Floyd-Warshall algorithm and the running time of the Floyd-Warshall algorithm is O(n^3) where n is the number of vertices, thus this modified algorithm also runs in O(n^3) time because that is the most time-intensive operation of the whole algorithm and satisfies our question's requirements.