- CS301 Homework 4 -

Problem 1:

Alternative 1: Since this is an optimal solution for longest common subsequence, there is randomness when returning the result. The length of the common subsequence must be the same but the returned sequence may depend on the program. So, this alternative solution may work in many cases but there is always a failure possibility. Therefore this solution (alternative 1) is much faster than the other solution, but there is a failure possibility.

a-

If we want to increase the number of strings (more than 2) to find the LCS of them, we can use 3 strings LCS and and an array to hold the common subsequence extra to our recursive formula.

1- Let's look at our LCS function for 2 strings with dynamic programming. In this approach, we have a memoization matrix (2D) to store to number of common subsequence found so far and we are comparing each possible subsequence of that 2 strings. This approach would take O(mn) where m,n denotes the lengths of the words. The algorithms are the same with LCS and LCS with top-down dynamic programming, both call the subproblems and try to build up to the real problem but with dynamic programming we are holding a memoization matrix (2D) and speed up the process.
The recursive formulation of LCS for 2 strings, where c denotes the memoization matrix

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise.} \end{cases}$$

2- LCS function for 3 strings with or without dynamic programming is no difference from LCS for 2 strings. This time instead of calling 2 strings subproblems we will call 3 strings subproblem. For dynamic programming we could use a 3D matrix as our memoization matrix. Each dimension denotes a string. So the recursive formulation will be:

$$c[i][j][k] = \begin{cases} 1 + c[i-1][j-1][k-1] & if\ (X[i] = Y[i] = Z[i]), \\ \max(c[i-1][j][k], c[i][j-1][k], c[i][j][k-1]) & otherwise \end{cases}$$

Where c is again memoization matrix and X,Y,Z are the strings and i,j,k are the dimensions.

3- So, we have the 3-LCS function to find the LCS of 3 functions. The main k-LCS function will get an array of strings and compare them with each other to find the LCS of k many strings. To do that, we can create a for loop and iterate over the array and compare them 3 by 3. The algorithm will follow these steps;
   a. Call the 3-LCS function for the first 3 words, using a for loop.

b. Put the result into the arrays first index.
c. Call the 3-LCS function for the arrays first index as the first input and take remaining 2 words.
d. Repeat the steps b,c untill the arrays is completely iterated or the LCS is a NULL string (no common subsequence)
e. Return the first index of the array

```
def lcsOf3(X, Y, Z, m, n, o):

    L = [[[0 for i in range(o+1)] for j in range(n+1)]
         for k in range(m+1)]
    for i in range(m+1):
        for j in range(n+1):
            for k in range(o+1):
                if (i == 0 or j == 0 or k == 0):
                    L[i][j][k] = 0

                elif (X[i-1] == Y[j-1] and
                      X[i-1] == Z[k-1]):
                    L[i][j][k] = L[i-1][j-1][k-1] + 1

                else:
                    L[i][j][k] = max(max(L[i-1][j][k],
                    L[i][j-1][k]),
                                     L[i][j][k-1])
    return L[m][n][o]
```

3-LCS function without traceback to the subsequence and the basic idea for to solve the k-LCS problem.

```
def k-lcs (S):
    int i=1
    while (i<S.length())
    do  S[0] = 3-LCS(S[0],S[i],S[i+1])
        if(S[0].length() == 0)
            return 0
        else
            i+2
    return S[0]
```

b-

The time complexity of the algorithm will depend on the longest 3 strings in our strings array. The for loop will iterate over the list but since it increases by 2 each time the time complexity for the for loop would be O(k/2) where k denotes the number of the strings. O(k) from the iterating the list and for each iteration we have 3 for loops to compute the longest common subsequence. In the end the time complexity will be O(k ($x_1x_2x_3 + x_1x_4x_5 + \dots + x_1x_{k-1}x_k$)) where x_i denotes the i^{th} strings' length. Overall the time complexity is O($kx_ix_jx_l$) where x_i,x_j,x_l are the longest 3 strings in the array.

Alternative 2: Alternatively, this solution will always return correct result for k-LCS problem but the running time is much higher than the alternative 1.

a-

Since we saw how to implement 3-LCS function in the alternative 1, we can implement the k-LCS function in the same way. The logic is the same but instead of 3D matrix we need to create a k dimensinonal matrix to hold the memoization. The recursive formula will be increased version of the 3-LCS recursive formula.

$$c[i][j][k]\dots = \begin{cases} 1 + c[i-1][j-1][k-1]\dots & if\ (X[i] = Y[i] = Z[i]\dots), \\ \max(c[i-1][j][k], c[i][j-1][k], c[i][j][k-1]\dots) & otherwise \end{cases}$$

b-

The time complexity will be $O(x_1x_2x_3x_4x_5\ldots)$ where x_i denotes the length of the strings. So, we can easily see that it will be exponential -kind a like $O(n^k)$- depending on the length of the strings. The space complexity will be the same with the time complexity.

Problem 2:

a-

This problem is called "Minimum Vertex Cover of a Tree" and it can be solved by using dynamic programming approach in O(V) time -V denotes the number of vertices in the tree-.

Computational Problem:

Input:

- A tree T(V,E) where V is the set of vertices and E is the set edges between parents and children.

Output:

- A set X where the elements of X covers all the edges and the number of the elements are optimally minimal.

Subproblems:

- To find the whole tree's minimal cover we need to call its children's minimal cover and calculate the result. So, the find a solution for our problem we can use subproblems results.

Optimal Substructure:

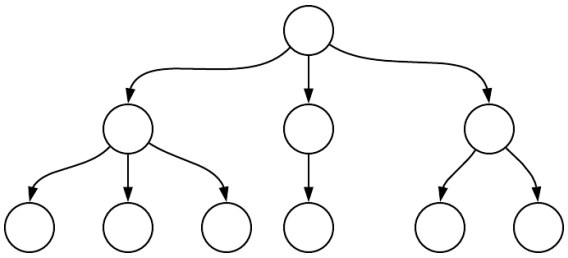
- The algorithm we use for the smaller problem is the same with the real problem. To calculate the minimum vertex cover for tree T, we can simply call the same recursive function for root's children. Finally, adding up the results, we find the real problems result.

Recursive Solution:

- To solve this problem, we need to check 2 things. Including the root for our minimal cover or not. This is valid for all subproblems.
- So, in the end the recursive formula would look like this, Minimum Vertex Cover (MVC) for a given tree's root.

$$MVC(root) = \min(\ 1 + \sum MVC(root.children)\ ,\ \#children + \sum MVC(root.grandchildren)\ )$$

The end case of the recursion is when there is no child for the given root.



For example, for the given tree, to calculate the minimal cover, we first include the root to our vertex cover and calculate, then exclude the root and calculate. The minimum of these two will give us the answer. If we include the root we **most likely** don't have to include the children, but it is not a certain thing, so we must compute the MVC for roots children as well. However, if we exclude the root we must add the #children to our vertex cover since there are edges between root and its children. We sum the results coming from the children to find the roots MVC.

Memoization & Overlapping Substructure:

- Since we are calculating some vertices' MVC multiple times (overlapping substructure), we can use memoization technique in order to reduce the time complexity of our algorithm. In this case, we can store the MVC of each vertex in a variable called count as an attribute to vertices.

Construction of Solution:

- All in all, to calculate MVC for a tree, we can use the recursion given above and with dynamic programming we can make it faster. With the help of the properties such as subproblem, optimal substructure, memoization… we can solve this problem. The recursive formula given above is the answer to the minimum vertex cover.

Pseudocode:

```
MVC(root):
    if(root.count != -1):
        return root.count
    else if (root->NULL):
        return 0
    else if (root->left == NULL && root->right == NULL):
        return 0
    else:
        int include = 1 + MVC(root.children)

        int exclude = 0
        if(root->left != NULL):
            exclude = 1 + MVC(root->left.children)
        if(root->right != NULL):
            exclude = 1 + MVC(root->right.children)

        root.count = min(exclude,include)

        return root.count
```

!!! In the given pseudocode MVC(root.children) denotes the sum of all MVC for the children of the root.

b-

Without using dynamic programming, the time complexity would be ambiguous since we don't know how many children each root would have. However, in the dynamic programming approach if we computed the MVC for any root, we write it to the roots count variable. So, if we already calculated the MVC for that root we can easily return the value in O(1) time. In the end, the time complexity will be the number of vertices we have because when we calculated all the vertices' counts we will just return the value for it. Therefore, the time complexity will be O(V) and the space complexity with the memoization will be O(V) also.

c-

If the problem was a minimum cover for a graph, our algorithm will not give correct result. It would depend on the given root value. Since the root value is indeterministic our algorithm could not find a solution for the problem. Also, if the graph is cyclic there could be some infinite recursions also. Since the given input is indeterministic we can say that this vertex cover problem will become a NP problem. We could only approximate the solution or check the correctness of a given input and a root. Overall, finding vertex cover for a graph with this algorithm is absurd and MVC for a graph is an NP problem.

Problem 3:

a-

"Minimum Leaf Spanning Tree" problem is a different version of MST problem. MST is a subset of edges of a connected, weighted undirected graph that connects all vertices together without any cycles and the total edge weight should be minimum. However, in MLST problem, instead of looking to weights of the edges, we look into the degrees of the vertices. Degree of a vertex is defined by, the number of edges incident to the vertex in given graph.

MSLT Problem: Given undirected, connected graph G(V,E), the result is a spanning tree T that has the minimum number of leaves in the tree.

Input: An undirected, connected graph G(V,E).

Output: A spanning tree T that has minimum number of leaves.

MSLT-Decision Problem: Given undirected,connected graph G(V,E) and an integer k, decide wheter there is spanning tree that has at most k many leaves in the tree.

Input: An undirected, connected graph G(V,E) and integer k.

Output: YES if there exists a spanning tree that has at most k many leaves, else NO.

b-

Travelling Water in a Central Heater Problem:

Suppose we have a house and we need to design a Central Heater System. While designing the system, we need to span all the rooms in the house to heat and we need to use minimum amount of pipes. In order to span all the rooms and have the minimum amount of pipes, our pipeline system need to have minimum amount of reciever pipes (to get the cold water). So, each our pipeline system somehow needs to be connected to each other most of the times. If a pipeline goes to a room then comeback without visiting another room, it creates a pressure in the pipeline and it is risky for our system. Overall, what we want is a pipeline system that visits the all the rooms and the reciever pipes should be minimal. Reciever pipes are located in the rooms that has only 1 room connected to it.

This is an MLST problem. We can consider each room as a vertex and the degree of the vertices are how many rooms that room can connect to. We need to span all the house since we want each room to be heated when the Central Heater is on. In addition, we don't want unneccesary reciever pipes and its motor beacause it increases the explosion risks. The reciever pipes denotes the leaves of the tree. Since we don't want them, we need to avoid it, therefore the number of leaves should be minimal.

c-

Since MLST problem is a **NP** problem. We need to prove it. The steps are, guess, verification, and proving that verification is polynomial time.

Guess:

For MLST problem, first take 2 indeterministic inputs and a graph G(V,E). A spannig tree T(V,E) and an integer k -which tells us the total leaves of the result of the MLST problem will be at most k.-

Verification (pseodocode) & Running Time:

```
MLST-Decide(G,T,k):
    for each vertex in T:
        v.degree = calculate_degree(v)
    int number_of_leaves = 0
    for each vertex in T:
        if(v.degree = 1):
            number_of_leaves +=1

    if(number_of_leaves <= k):
        return YES
    else:
        return NO
```

The running time of the algorithm will be O(V) for MLST-Decide because we iterate over each vertex.

Also, O(VE) for calculate_degree function since calculating the degree of each vertex would take O(VE).

In the end, the overall algorithm would take polynomial time O(V + VE) which is O(V) where E and V are the number of edges and vertices.

MSLT is an **NP-Complete** problem.

Power of Reduction

- If A is NP-complete, $A \leq_P B$, and $B \in$ NP, then B is NP-complete
- Proof:
  - $C \leq_P A$ for all $C \in$ NP
  - $A \leq_P B$
  - $C \leq_P B$ for all $C \in$ NP

We can use the power of the reduction to prove that MLST is an NP-Complete problem.

In our case, we know that Hamiltonian Path problem is a NP-Complete problem and Hamiltonian Path problem is a subproblem for MLST problem where k is equal to 2.

Thus, we can easily say that:

If Hamiltonian Path problem is NP-Complete problem, Hamiltonian Path $\leq_p$ MLST problem, and we proved that MLST is NP problem, then MLST is NP-Complete.