Problem 1 (LCS of k sequences)

Efe Sencan

(a) In the lecture, we have have calculated the length of the LCS for only 2 sequences. In order to modify the algorithm for finding the LCS of k sequences: Firstly, we have to extend the checking condition of i,j variables to i,j,..k variables (which are defined as the index of the last character of the corresponding strings (it can not be negative, also these variables are 1 indexed)) whether they are equal to zero. If any of them is equal to zero this means there will not be any LCS from that position since the corresponding string would not be exist. The second if condition checks whether the last character of the strings are all equal to each other. If so, we simply add one to our result and recursively call the LCS function for k many (it was two) strings. The third condition was also extended for the k many strings such that, we first decrease the i by one while j,..k remains same. For the second parameter, we decrease j by one while i...k (except j) remains same and we apply this methdology for the k many strings.

Recursive formulation of LCS of k sequences

$$|LCS(X_i, Y_j, ...K_t)| =$$

$$\begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \text{ or} ...k = 0 \\ \mathit{1} + LCS(X_{i-1}, Y_{j-1}, ...K_{t-1}) & x_i = y_j = ... = k_t \\ \max\{LCS(X_{i-1}, Y_j, ...K_t), ..., LCS(X_i, Y_j, ...K_{t-1}),\} & otherwise \end{cases}$$

Top-Down Dynamic Approach for LCS of k sequences

For the dynamic approach of the LCS of k sequences, we have to create a table of k dimensions instead of 2, and initialize the values in the table to NIL (which means we have not yet calculated that value). The remaining parts of the dynamic approach problem would same with the one which we have covered in the lecture but instead, we use the recursive definition of the LCS for the k sequences.

Algorithm 1 Top-Down Dynamic Approach for LCS of k sequences

```
procedure LCS(x,y,...,k,i,j,...,t)
    if C[i,j,..t] == NIL then
        if x[i] = y[j] = ... = k[t] then
            C[i,j,...,t] = 1 + LCS(x,y,...k,i-1,j-1,...,t-1)
        else
            C[i,j,...,t] = max(LCS(x,y,...k,i-1,j,...t),LCS(x,y,...,i,j,...t-1)))
        end if

```

Problem **Efe Sencan**

(b) **Asymptotic time complexity of naive recursive algorithm:**

Consider the recursion tree of the of the LCS algorithm. There would be k branches (every node will have k children) for a spesific node of a tree, and at each step the only one parameter of the strings (i,j,...k) could be decreased by one in the worst case. Therefore the height of the tree would be n*k where n is the length of the longest word of the strings. Thus we get O(k^{nk}) time complexity.

Asymptotic time complexity of the top-down dynamic algorithm:

Since we create a table of k dimensions in order not to solve the same problem again for the dynamic approach and we will iterative over that table (if we know the value in the cell of a table we can reach it with constant time) then our time complexity would be O(n^k) which is also the cost of the the total time for filling the table. The result is not surprising since the time complexity was O(n^2) for the algorithm in the lecture since we were only dealing with the 2 strings. So the result is expected since we find an polynomial asymptotic time with the dynamic approach instead of exponential.

Problem 2 (Minimum Cardinality Set) **Efe Sencan**

(a) **Recursive Solution:**
There are two cases to consider:
Either we will take the current root in our vertex cover. If so, we will calculate the size of the vertex covers for the children of that root plus 1 (since we include the root). Alternatively, the root will not be part of our vertex cover. If that is the case, we will take all of its children to our vertex cover in order to include the edges from the children to the root. For this reason, we will calculate vertex cover of the grandchild of the root recursively plus the total number of its children.

V(c) = min(1 + sum(V(c) for i in v.children), len(v.children) + sum(V(g) for i in v.children, for j in i.children)

Sub-problems: Since we try to calculate the minimum vertex cover in a tree recursively, our sub-problems would be the size of the minimum vertex cover of sub-tree rooted at v (v is an element of V)

Overlapping Sub-problems: In order to apply the dynamic approach to the problem, we have to verify that our problem contains the overlapping substructure property. Consider the case that, we decided not to take the current root to our vertex cover, so we have to calculate the minimum vertex cover of the children of that root, but that minimum vertex cover of that children will also be calculated when the parent of our current root would not be included in our minimum vertex cover. So we will have many overlapping sub-problems for the problem.

Optimal Substructure Property: In order to calculate the minimum vertex cover of the tree, we have to calculate the minimum vertex cover of its children and the grandchild recursively and take their minimum at the end (We apply this for all the nodes). Since we calculate the minimum vertex cover for all the sub-problems, the result that we found at the end would be also optimal. Corollary, optimal solution to the problem contains the optimal solution for the sub-problems.

Algorithm 2 Top-Down Dynamic Approach for the Minimum Cardinality Problem

```
1: procedure LCS(r)
2:     if r− >vc == NIL then
3:         r− >vc = min(1 + sum(V(c) for i in v.children), len(v.children) + sum(V(g) for i in v.children, for j in i.children)
4:     end if
5: end procedure
       =0
```

Problem

(b) **Efe Sencan**

Since we create a table of size V (which is the total number of vertices in a tree), in order not to solve the problem that we have already solve, the total number of unique sub-problems would be $O(V)$. Since finding the cost of a problem that we already solved would be constant by looking up in a table, our total time complexity would be $O(V)$.

(c) The problem would be NP complete if the given input was a graph instead of a tree. In order to prove, we first have to show that vertex cover is in NP (time complexity would be exponential). Thus, given a graph G, we have to check whether it is the minimum cardinality set or not in polynomial time. Our algorithm would be, given a vertex set Vc, for each vertex of Vc if we have to remove all the incident edges of that vertex from our graph. At the end, we should check whether we moved all the edges from the graph G. If that is the case then the given vertex set is the minimum vertex cover of the graph, otherwise it is not. Since, we can implement the code in O($|E| + |V|$), it is polynomial. So the problem is NP class. The second we should do is the reduction step. The clique problem could be reducible to the vertex cover problem. Because there exist a clique of size k in a graph G if and only if there is a vertex cover of size $(|V| - k)$ in G'. Since we can do it in polynomial time, we can show that the problem in NP Complete set.

Problem 3 (Minimum Leaf Spanning Tree Problem) **Efe Sencan**

(a) **Input:** Graph G (v,e).

Output: The spanning tree of the graph G which has the minimum number of leaves(vertices with degree 1).

(b) **Real World Application:** Minimum leaf spanning trees are used in computer database systems in order to execute a query in a database efficiently. In order to determine a set of indexes for a table in the database, that can efficiently process a number of anticipated query types, each query type should reference a corresponding combination of the table's columns. This conditions could be handled by using an equivalent graph which is built based on the combination of the table's columns. A minimum leaf spanning tree for the graph is found and indexes are created for the table based on the minimum leaf spanning tree. Since the leaves of a spanning tree of the equivalent graph will correspond to a set of indexes that can cover the anticipated query types. Thus, minimizing the number of leaves in such a spanning tree will lead to an efficient set of indexes.

(c) **Proof of NP Completeness:**
We need 2 steps in order to prove that the problem is NP complete. Firstly, given a guess (which will be a tree) we should be able to verify whether the given tree is minimum leaf spanning tree or not in polynomial time. In order to do that, given a tree T, we will first find the leaves of that tree. If the number leaf is 1, then it would be minimum leaf spanning tree, if that is not the case and the number of leaves is two then it is also a minimum leaf spanning tree. Otherwise, for every leaves, we are going to check whether that leave has a direct edge to any other leave in the original graph G in order to check whether the tree is minimum leaf spanning tree or not. If so, we can say that the spanning tree of the graph is not a minimum leaf spanning tree. Finding the leaves in a tree could be done in polynomial time, besides that checking a whether any other leaf has a direct edge with the current leave value could be also performed in polynomial time. Therefore, we could prove that our problem is NP.

For the second step, we have to reduce a problem which is in NP complete set,to our problem in polynomial time. Actually, the MLST is equivalent to the Hamiltonian path (which is a NP-complete problem) problem when the number leaves is at most 2 in our spanning tree. Therefore, the Hamiltonian path problem could be reduced to our problem in polynomial time by taking the leave value which is smaller than or equal to 2 . As a result we proved that MLST is a NP-complete problem.