Shortest Subsequence

Problem Description: Gary has two string S and V. Now Gary wants to know the length shortest subsequence in S such that it is not a subsequence in V.

Note: input data will be such so there will always be a solution.

For example:
Input strings:
babab
babba

Output: 3
Explanation: "aab" is the smallest substring in *S* which is not present in *V*.

Naive Solution: Time: O(2^(S.len) + 2^(V.len)), Extra Space: O(2^(S.len) + 2^(V.len))

The naive solution that would come in our mind is to simply use a hashmap to save the 2^(S.len) subsequences of S and 2^(V.len) subsequences of V. Now the longest subsequence with frequency 1 would be our answer.

The Space complexity can be reduced to O(1) by instead of using a hashmap we simply iterate over every possible string of V of length *x* where x is the length of subsequence of S to be compared. We will do this for every possible subsequence of S. Think about the time complexity!

Recursive Approach

When presented two strings S, V and i - ith character in S, there are 3 possibilities:

1. S[i] is not present in V: In this case, the shortest subsequence in S not present in V would simply be S[i]. S[i] is just a single character so the answer would be 1.
2. S[i] is present in V and we are including it in our answer. In that case, our new set of strings to further probe for would be S+1 and V+x+1 - where x is the occurrence.
3. S[i] is present in V and we are not including in our answer. In this case, our new set of strings to further look into would be S+1 and V.

For every character not following case 1 we will check for both 2, 3 and our answer string would have whatever of the two combinations produces the least long substring.

Psuedocode

```
int shortestSequence(S,T, remainSLength, remainTLength){
   if (remainSLength == 0)
      return MAX_INT;

   if (remainTLength == 0)
      return 1;

      if S[0] not in T
               return 1;

      else
                  k=i when T[i] = S[0]

   return min(shortestSequence(S+1, T, remainSLength-1, remainsTLength),
         1 + shortestSequence(S+1, T+k+1, remainsSLength-1, remainsTLength-k-1));
}
```

Dynamic Programming: Time Complexity - O(mn^2), Space Complexity - O(mn)

We can observe that there are many subproblems which are getting solved over and over in our recursive solution. We can use the concept of Dynamic Programming to further decrease the runtime.

We establish the relationship between the subproblems and the problem at hand.
Let dp(i, j) be the length of the shortest subsequence that is in S[1..i] that is not in V[1..j].
Here we have the following relationship: dp(i, j) =
1. If letter S[i] is nowhere to be found in V[1..j], then dp(i,j) = 1.
2. Otherwise, we have two case:
    2.1. S[i] is in the shortest subsequence. We find k, the most immediate index in V[1..j]
where V[k] == S[i]. Then dp(i, j) = 1 + dp(i-1, k-1).
    2.2 S[i] is not in the shortest subsequence. Then dp(i, j) = dp(i-1, j).
We pick whichever is lower.

Psuedocode

```
int solve(string S,string V){

        int n = S.size(), m = V.size() , i, j, prev, dp[n+1][m+1] , next[n+1][m+1];

        for i in range(n){
                prev = -1;
                for j in range(m){
                        if(S[i] == V[j])
                                prev = j;
                        next[i+1][j+1] = prev;
                }
        }

        for i in range(n+1)
                dp[i][0] = 1;

        for i in range(m+1){
                dp[0][i] = INF;
        }

        for i in range(1, n+1){
                for j in range(1, m+1){
                        if(next[i][j] == -1)
                                dp[i][j] = 1;
                        else{
                                dp[i][j] = min(dp[i-1][j],1 + dp[i-1][next[i][j]]);
                        }
                }
        }
        return dp[n][m];
}
```