Longest Common Subsequence

We can use the same method that we did in class, with the subsequence of two arrays, but instead we can have 3 arrays. There should only need to be slight changes for it to work.

What: $S[j,k,l]$ = the length of the longest common subsequence between the three subsequences $(\{a_1, a_2, ..., a_j\}, \{b_1, b_2, ..., b_k\}, \{c_1, c_2, ..., c_l\})$

How: $S[j,k,l] =$

$$\begin{cases} 0 \text{ if } j = 0 \text{ or } k = 0 \text{ or } l = 0 \\ max\{S[j-1,k,l], S[j,k-1,l], S[j,k,l-1]\} \text{ if } j,k,l > 0 \text{ and } a_j = b_k \text{ or } a_j = c_l \text{ or } b_k = c_l \\ \max\{S[j-1,k,l], S[j,k-1,l], S[j,k,l-1], S[j-1,k-1,l-1]+1\} \text{ if } j,k,l > 0 \text{ and } a_j = b_k = c_l \end{cases}$$

Where: $S[p,q,r]$

Psudocode

```
def longest_common_subseq(A:array<int>, B:array<int>, C:array<int>):
    for j=0 to A.len:
        for k=0 to B.len:
            S[j,k,0] = 0
    for j=0 to A.len:
        for l=0 to C.len:
            S[j,0,l] = 0
    for k=0 to B.len:
        for l=0 to C.len:
            S[0,k,l] = 0
    for j = 1 to A.len:
        for k=1 to B.len:
            for l=1 to C.len:
                S[j,k,l] = max{S[j-1, k, l], S[j, k-1, l], S[j, k, l-1]}
                if A[j] == B[k] && B[k] == C[l]:
                    S[j,k,l]=max{S[j,k,l], S[j-1,k-1,l-1]+1}
    return S[A.len, B.len, C.len]
```

Proof

First of all, assuming that the erlier elements of the S array are correct, we know that $S[j,k,l]$ will be the maximum possible value, since the maximum value must be either the maximum of the three smaller subproblems (dropping one element from A,B, or C), or it must be that the current elements match, in which case the maximum is one plus the best solution to the previous maximum at $S[j-1,k-1,l-1]$. Therefore we know that $S[j,k,l]$ is correct given that the previous elements to it in the i j and k dimensions are correct. If we can prove that the first elements are correct, then we can prove the correctness of the entire algorithm. The elements in the first row (elements at index 0 in i j or k), must all be zero, since at this point we are considering all the arrays as empty, therefore the

longest subsequence must be empty. Since this base case is correct and each step which builds on it is correct, the solution is correct.

Running Time

Estimate

The running time is $O(p \cdot q \cdot r)$

Reasoning

```
def longest_common_subseq(A:array<int>, B:array<int>, C:array<int>):
    for j=0 to A.len:# Runs p times -> O(pq)
        for k=0 to B.len:# Runs q times -> O(q)
            S[j,k,0] = 0#O(1)
    for j=0 to A.len:# Runs p times -> O(pr)
        for l=0 to C.len:# Runs r times -> O(r)
            S[j,0,l] = 0#O(1)
    for k=0 to B.len:# Runs q times -> O(qr)
        for l=0 to C.len:# Runs r times -> O(r)
            S[0,k,l] = 0#O(1)
    for j = 1 to A.len:# Runs p times -> O(pqr)
        for k=1 to B.len:# Runs q times -> O(qr)
            for l=1 to C.len:# Runs r times -> O(r)
                S[j,k,l] = max{S[j-1, k, l], S[j, k-1, l], S[j, k, l-
1]}#O(1)
                if A[j] == B[k] && B[k] == C[l]:#O(1)
                    S[j,k,l]=max{S[j,k,l], S[j-1,k-1,l-1]+1}#O(1)
    return S[A.len, B.len, C.len], S#O(1)
```

Since the longest part of the algorithm is $O(p \cdot q \cdot r)$, the algorithm takes $O(pqr)$ time.

Reconstruction

For reconstruction, we also want to use a method similar to the one in we used in class, simply adding a third dimension. We do this by starting at the end ($S[p, q, r]$) and working our way back, by checking if all three dimensions previous elements are equal, then moving diagonally if they are and moving in the direction of the higher element otherwise.

Reconstruction Psudocode

```
def reconstruct_3d_subseq(A:array<int>, B:array<int>, C:array<int>,
S:array<array<array<int>>>):
    j=A.len,k=B.len,l=C.len
    let output:array<int>
    while (j>0 and k>0 and l>0):
        if S[j-1,k,l]==S[j,k-1,l] and S[j-1,k]=S[j,k,l-1]:
            j--,k--,l--
            out.append(A[j])
        else if S[j-1,k,l]>S[j,k-1,l] and S[j-1,k,l] > S[j,k,l-1]:
            j--
        else if A[[j,k-1,l]]>S[j-1,k,l] and S[j,k-1,l] > S[j,k,l-1]:
            k--
        else:
            l--
```

Reconstruction Running Time

Reconstruction Estimate

The running time is $O(p \cdot q \cdot r)$

Reconstruction Reasoning

```
def reconstruct_3d_subseq(A:array<int>, B:array<int>, C:array<int>,
S:array<array<array<int>>>):
    j=A.len,k=B.len,l=C.len # O(n)
    let output:array<int> # O(n)
    while (j>0 and k>0 and l>0): # Runs p+q+r times
        if A[j-1]==B[k-1] == C[l-1]:# Runs p times
            j--,k--,l-- # O(n)
            out.append(A[j]) # O(n)
        else if S[j-1,k,l]>S[j,k-1,l] and S[j-1,k,l] > S[j,k,l-1]:# Runs q
times
            j-- # O(n)
        else if S[j,k-1,l]>S[j-1,k,l] and S[j,k-1,l] > S[j,k,l-1]:# Runs r
times
            k-- # O(n)
        else:
            l-- # O(n)
```

Since the longest part of the algorithm runs $p + q + r$ times, the algorithm takes $O(p + q + r)$ time.