The LCS Algorithm

→ the definition of $L_{j,k}$ satisfies subproblem optimization, for we cannot have an LCS without also having longest common subsequences for the subproblems

→ it also uses subproblem overlap b/c a subproblem solution $L_{j,k}$ can be used in several other problems

→ to turn $L_{j,k}$ into algorithm:

- → create $(n+1) \times (m+1)$ array, defined for $0 \leq j \leq m$ and $0 \leq k \leq m$
- → init all entries to 0, In particular so all entries $L_{j,0}$ and $L_{0,k}$ are zero
- → then iteratively build up values in L until we have $L_{n,m}$

```
def LCS(X,Y):
  n,m = len(X), len(Y)
  L = [[0] * (m+1) for k in range(n+1)]        //(n+1) x (m+1) table
  for j in range(n):
    for k in range(m):
      if X[j] == Y[k]:                         //align this match
        L[j+1][k+1] = L[j][k] + 1
      else:                                    //choose to ignore one char
        L[j+1][k+1] = max(L[j][k+1], L[j+1][k])
  return L
```

→ running time: algorithm uses two nested for loops, outer one iterates n-times and inner loop runs m times. As such, the algorithm runs in $O(n*m)$.

→ previous algorithm only finds length of LCS, following algorithm constructs LCS in $O(n+m)$ time

→ explanation:

- at any position $L_{j,k}$, if $x_j = y_k$, then length is based on common subsequence associated w/ length $L_{j-1,k-1}$ followed by common character x_j
- we can record x_j as part of the sequence, then continue analysis from $L_{j-1,k-1}$
- if $x_j \neq y_k$, we can move to larger of $L_{j,k-1}$ and $L_{j-1,k}$
- continue process until reaching some $L_{j,k} = 0$

→