CMSC 451 - Algorithm Design
Lecture 9 - DP: LCS and Edit Distance

Strings - Used in document processing & computational genomics

This lecture - Dynamic Programming (DP) algorithms for two string processing problems:
- Longest common subsequence (LCS)
- Edit Distance

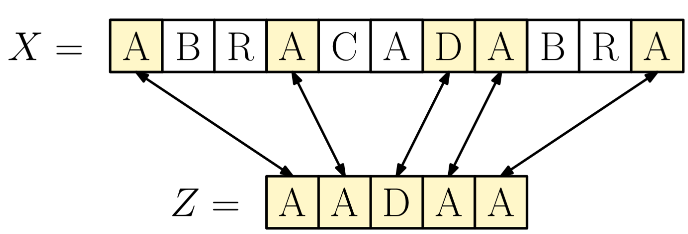
Notation - X is a string $= \langle x_1, \ldots, x_m \rangle$ over some alphabet Σ.
e.g. $\Sigma = \{a, b, c, \ldots, z\}$, $\Sigma = \{A, C, G, T\}$

$|X|$ = length of X

X_i = prefix $\langle x_1, \ldots, x_i \rangle$ $X_0 = \langle \rangle$

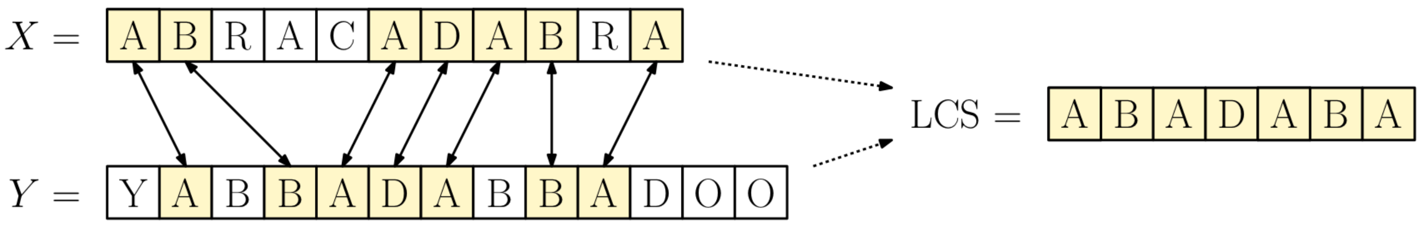
A string $Z = \langle z_1, \ldots, z_k \rangle$ is a subsequence of X if Z's characters appear in order in X.

E.g.



Given strings X + Y, their longest common subsequence (LCS) is a max length string that a subsequence of both

Example:



Note: The LCS is not unique

$$LCS(\langle ABC\rangle, \langle BAC\rangle) = \langle AC\rangle \text{ or } \langle BC\rangle$$

DP Formulation for LCS:

- Decompose into subproblems (recursive)
- Priniciple of optimality will apply
  (subproblems should be solved optimally)

Define: For $0 \le i \le m$, $0 \le j \le n$:

$lcs(i,j)$ = length of LCS for prefixes $X_i = \langle x_1 \dots x_i\rangle$ + $Y_j = \langle y_1 \dots y_j\rangle$

Eg. $X_5 = \langle ABRAC\rangle$ $Y_6 = \langle YABBAD\rangle$

$lcs(5,6) = 3$ $(\langle ABA\rangle)$

Basis: If $i = 0$ or $j = 0$ (empty string)
then LCS is empty

$$\boxed{\Rightarrow \ lcs(i,0) = lcs(0,j) = 0}$$

Last characters match: $x_i = y_j$

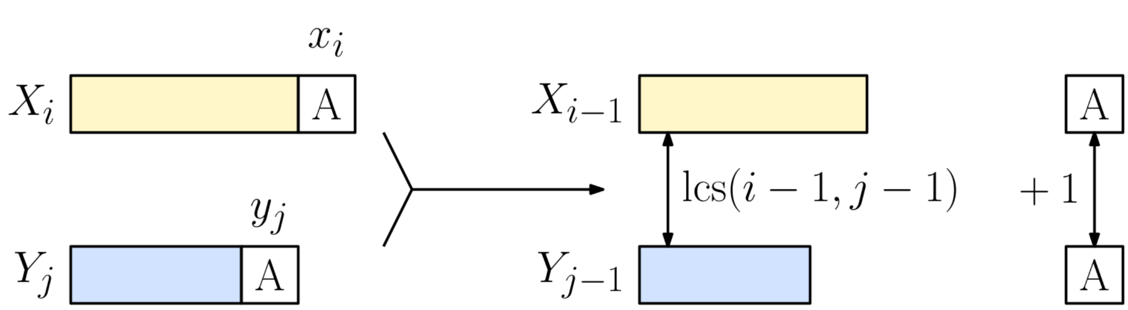
(Suppose $x_i = y_j = $ 'A')

Claim: LCS also ends in 'A'

Proof: Obvious. If not, we could extend it by appending on 'A'.

- Since LCS ends in 'A', we may as well assume it comes from mathing x_i with y_j. (There is no benefit from matching it earlier.)
- Once matched, x_i + y_j are elimated from further consideration.
- We should do our best with remainders

$$X_{i-1} = \langle x_1 \ldots x_{i-1} \rangle + Y_{j-1} = \langle y_1 \ldots y_{j-1} \rangle$$



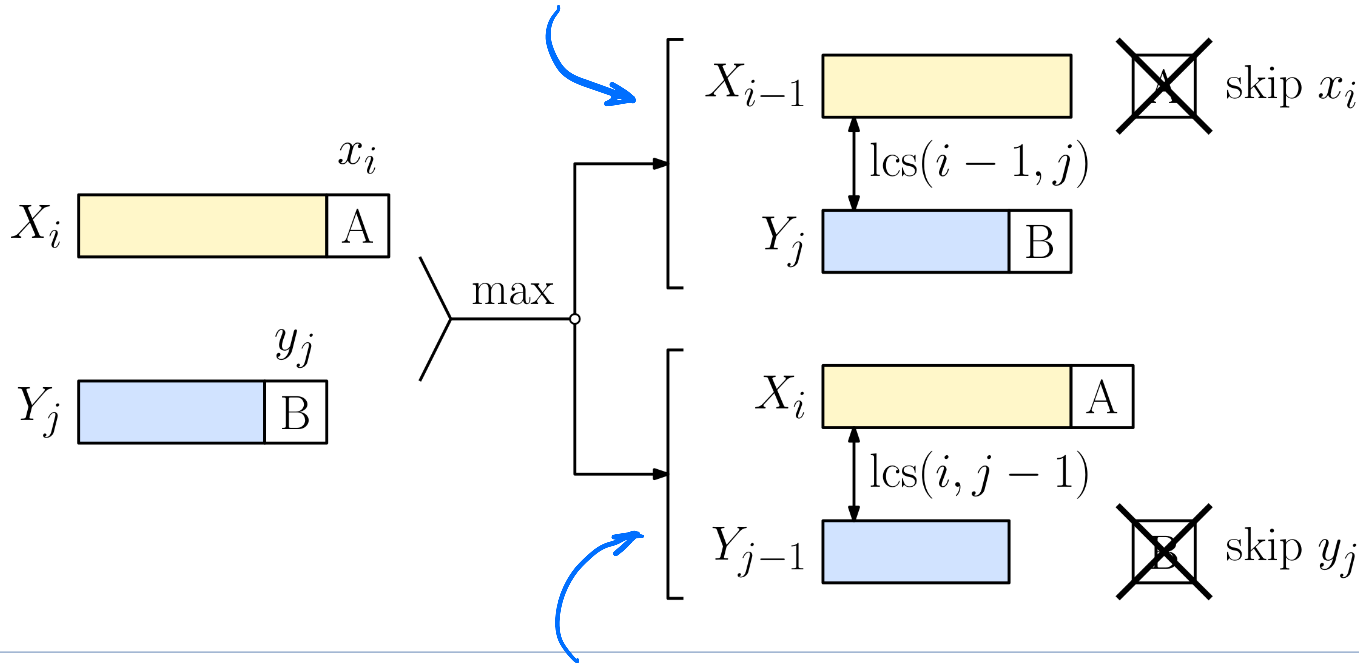
$$\boxed{\Rightarrow \ \text{if } (x_i = y_j) \ lcs(i,j) = lcs(i-1, j-1) + 1}$$

Last characters do not match: $x_i \neq y_j$

- Either x_i or y_j or both are not in LCS.

- x_i is not in LCS
  - we may ignore x_i + continue matching remainder $X_{i-1} = \langle x_1 \dots x_{i-1} \rangle$ with Y_j

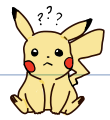
  $\Rightarrow$ $lcs(i-1, j)$



- y_j is not in LCS
  - (symmetrical) ignore y_j + continue matching remainder Y_{j-1} with X_i

  $\Rightarrow$ $lcs(i, j-1)$

- Both x_i + y_j not in LCS
  - This will be handled by above cases.



- But which?

DP Credo: Don't be smart.
Try 'em all. Take the best.

$$\Rightarrow \text{if } (x_i \neq y_j) \quad lcs(i,j) = \max \begin{cases} lcs(i-1, j) \\ lcs(i, j-1) \end{cases}$$

Final DP Formulation:

$$lcs(i,j) = \begin{cases} 0 & \text{if } \min(i,j) = 0 \\ 1 + lcs(i-1, j-1) & \text{if } x_i = y_j \quad (i,j>0) \\ \max \begin{Bmatrix} lcs(i-1, j) \\ lcs(i, j-1) \end{Bmatrix} & \text{if } x_i \neq y_j \quad (i,j>0) \end{cases}$$

- Correctness follows from earlier derivation
- Recursive implementation will take exp. time
- Instead: Build table $lcs[0..m, 0..n]$ through
    - memoization (caching)
    - or - bottom-up

Memoized Implementation (+ Hooks)

- Build table $lcs[i,j]$ recursively
- Add table $H[0..m, 0..n]$ to remind us of decisions made, so we can reconstruct LCS.
- Init: $lcs[i,j] \leftarrow -1$ (undefined)
- Final result: memo-lcs(m,n) $m = |X|$, $n = |Y|$

```
memo-lcs(i,j)                                   // memoized LCS
    if (lcs[i,j] = -1)                          // undefined?
        if (i = 0 or j = 0)                     // basis
            lcs[i,j] ← 0
        else if (x_i = y_j)                     // match?
            lcs[i,j] = 1 + memo-lcs(i-1, j-1)
            H[i,j] = '↖'
        else  // x_i ≠ y_j                      // don't match
            skipX ← memo-lcs(i-1, j)            // lcs if skip x_i
            skipY ← memo-lcs(i, j-1)            // lcs if skip y_j
            if (skipX ≥ skipY)                  // better to skip x_i
                lcs[i,j] ← skipX ; H[i,j] ← '↑'
            else                                // better to skip y_j
                lcs[i,j] ← skipY ; H[i,j] ← '←'
    return lcs[i,j]                             // final lcs value
```

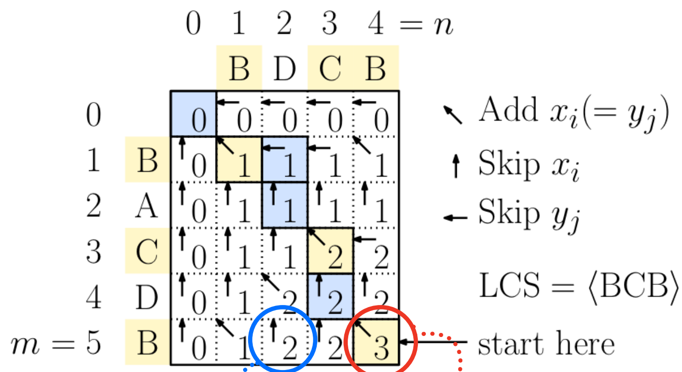
Running time: $\mathcal{O}(n \cdot m)$

$X = \langle BACDB \rangle$

$Y = \langle BDCB \rangle$

		0	1	2	3	$4 = n$
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
$m = 5$	B	0	1	2	2	3

$|LCS| = 3$



$x_5 = y_4 = \text{'B'}$

$\Rightarrow lcs[5,4] = lcs[4,3] + 1 = 3$

$\langle BCB \rangle = \langle BC \rangle + \text{'B'}$

Add $x_5 = y_4 = \text{'B'}$ to LCS

Cont. with H[4,3]

No change to LCS: ⟨BD⟩

Cont. with H[4,2]

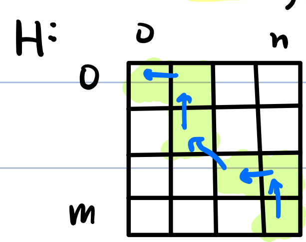
$x_5 = \text{'B'} \neq y_2 = \text{'D'}$

$\Rightarrow lcs[5,2] = \max(lcs[4,2], lcs[5,1]) = 2$

$\langle BD \rangle = \max(\langle BD \rangle, \langle B \rangle)$

Extracting the LCS:

- We use the H matrix
- Start at H[m,n] + trace back to H[0,0]



- Entries: H[i,j]
  - '↖': Add $x_i = y_j$ to LCS, continue with H[i-1, j-1]
  - '↑': Skip x_i. Continue with H[i-1, j]
  - '←': Skip y_j. Continue with H[i, j-1]

- Note that changes to i+j mimic recursive structure

```
get-lcs-sequence()                  // get the LCS sequence
    LCS ← ∅                         // initialize
    i ← m; j ← n                    // start at bottom-right
    while (i ≠ 0 or j ≠ 0)          // end at top-left
        switch (H[i,j])
            '+' : prepend x_i to LCS    // match x_i = y_j
                  i--; j--
            '↑' : i--                   // skip x_i
            '←' : j--                   // skip y_j
    return LCS
```

(see figure above for example)

Running time: $O(n+m)$ - Each iteration decrements either i or j (or both)

Bottom-up implementation (see pdf for details)

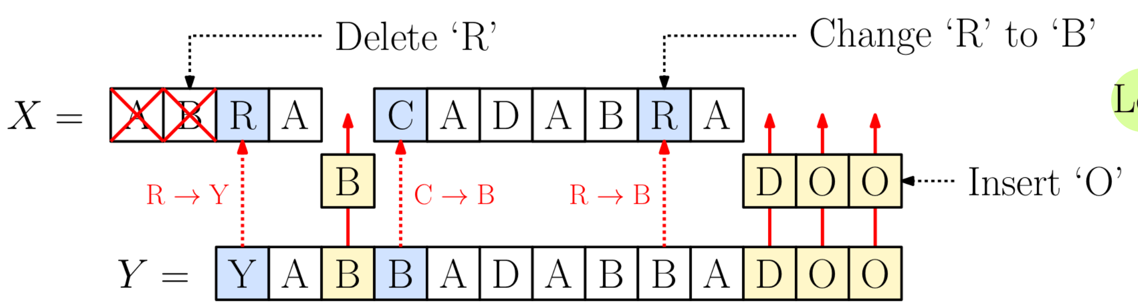
- fill row by row $i \leftarrow 0..m$
  + col by col $j \leftarrow 0..n$
- Also $O(n \cdot m)$

Edit Distance:

- Widely used in genomics
- Given $X = \langle x_1, \ldots x_m \rangle$ + $Y = \langle y_1, \ldots y_n \rangle$ how many edit ops are needed to convert X into Y, where edit ops:
  - insert a char of Y into X
  - delete a char of X
  - change a char of X to match a char of Y
- This defines a metric on strings called the Levenshtein distance (Vladimir Levenshtein)

Example:

Change ⟨ABRACADABRA⟩ → ⟨YABBADABBADOO⟩



$\mathrm{Lev}(X, Y) = 9$

- We'll present the recursive DP formulation.
  - Implementable (memoized or bottom-up) in $O(m \cdot n)$ time

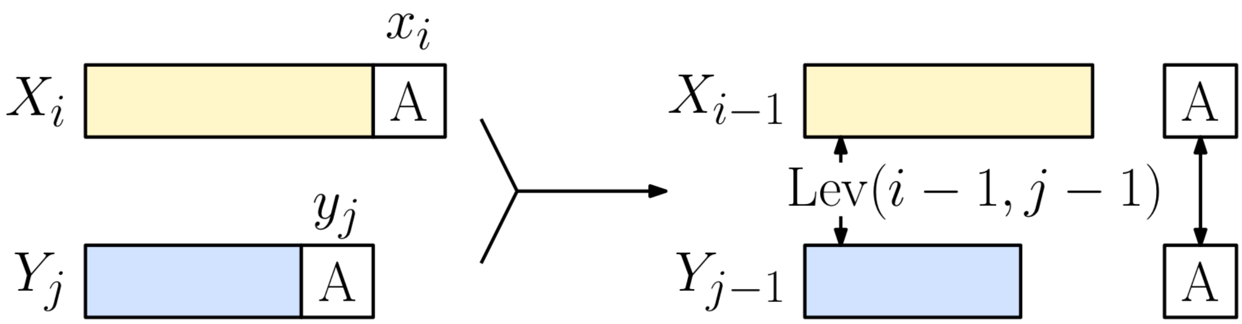
- Structurally similar to LCS
- Cute trick - inserting into X is like deleting from Y

Definition: Given $X = \langle x_1 \ldots x_m \rangle$ + $Y = \langle y_1 \ldots y_n \rangle$
for $0 \le i \le m$ + $0 \le j \le n$
$Lev(i,j)$ = Levenshtein distance between $X_i = \langle x_1 \ldots x_i \rangle$ + $Y_j = \langle y_1 \ldots y_j \rangle$
Final goal - $Lev(m,n)$

Basis:
$i=0$ - No chars in X. Need j inserts from Y
$\Rightarrow Lev(0,j) = j$
$j=0$ - No chars in Y. Need i deletes from X
$\Rightarrow Lev(i,0) = i$
- otherwise if $\min(i,j) > 0$

Last characters match: $x_i = y_j$
- We should go ahead + match them
+ continue with remainders X_{i-1} + Y_{j-1}

$$\boxed{\Rightarrow \text{if } (x_i = y_j) \quad Lev(i,j) = Lev(i-1, j-1)}$$



Last characters do not match: $x_i \neq y_j$

- Need to do something, but what?
  1. Insert y_j at end of X_i
  2. Delete x_i
  3. Change x_i into y_j

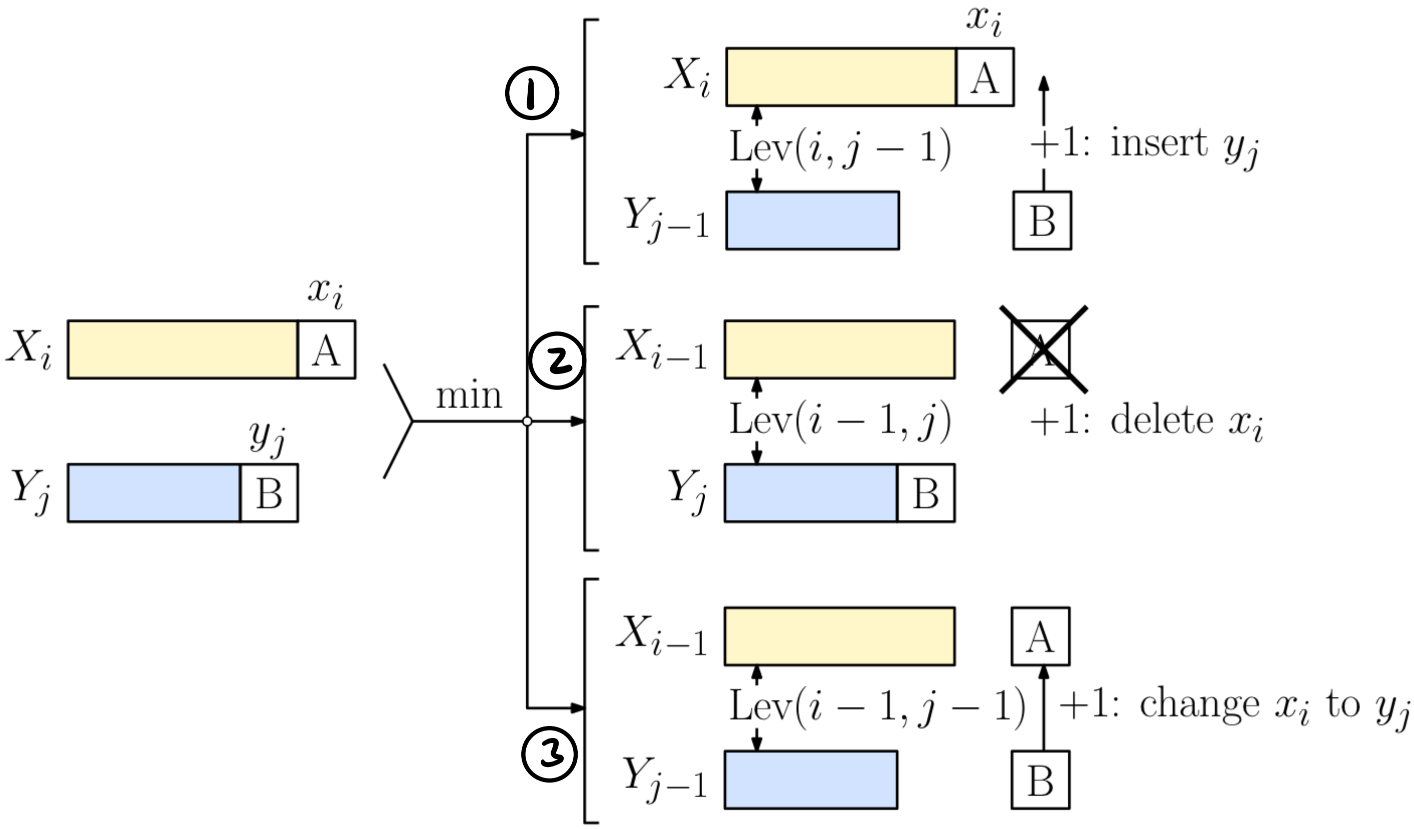
- In any of these cases, distance goes up by +1

If 1, we are done with y_j
+ continue with remainder Y_{j-1}
$\Rightarrow 1 + \mathrm{Lev}(i, j-1)$

If 2, we are done with x_i
+ continue with remainder X_{i-1}
$\Rightarrow 1 + \mathrm{Lev}(i-1, j)$

If 3, we are done with both x_i + y_j
Continue with remainders X_{i-1}, Y_{j-1}
$\Rightarrow 1 + \mathrm{Lev}(i-1, j-1)$

3 options:

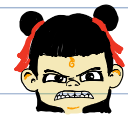


Which option?

- Remember the DP Credo – Try all
  Take best (min)
- Final DP formulation:

$$
\text{Lev}(i,j) = \begin{cases}
j \quad \text{(insert all } Y_j\text{)} & \text{if } i = 0 \\
i \quad \text{(delete all } X_i\text{)} & \text{if } j = 0 \\
\text{Lev}(i-1, j-1) \quad \text{(match)} & \text{if } i,j > 0 \;\&\; x_i = y_j \\
1 + \min \begin{cases} \text{Lev}(i, j-1) & \leftarrow \text{insert } y_j \\ \text{Lev}(i-1, j) & \leftarrow \text{delete } x_i \\ \text{Lev}(i-1, j-1) & \leftarrow \text{change } x_i \to y_j \end{cases} & \text{if } i,j > 0 \;\&\; x_i \neq y_j
\end{cases}
$$

Leave implementation as exercise.



- $O(n \cdot m)$ like LCS
- Can extract edits in $O(m+n)$ time

Summary:

DP Algorithms for
- Longest Common Subsequence (LCS)
- Edit (Levenshtein) distance

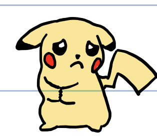
- Both run in $O(n \cdot m)$ time (quadratic)

Can we do better?

LCS - Yes - Near linear time

Levenshtein - In practice - yes

In theory - no



Lower bound of $O(n^{2-\varepsilon})$ for any $\varepsilon > 0$ under the Strong Exponential Time Hypothesis (SETH)