CS340 - Analysis of Algorithms

Dynamic Programming
Least Common Subsequence

Announcements:

Hw6 Due Monday November 10th

Divide and Conquer Quiz tomorrow in Lab

Quiz Format

1. Solving recurrences with masters theorem
2. Algorithm with runtime and proof (divide and conquer)
    a. Problems to study:
        i. Hw5 2 and 3
        ii. Solved exercise 1 in the book

Dynamic Programming

DP Design

Break down the problem by **expressing the optimal solution in terms of optimal solutions to sub-parts**

- Subproblems may overlap
- The number of subproblems must be reasonably small

To do this, determine the recursive formulation

- First describe the precise function you want to evaluate in English
- Then, give a formal recursive definition of the function

Dynamic Programming

Find a (small) choice whose correct answer would reduce the problem size

For each possible answer, temporarily adopt that choice and recurse

Don’t be clever with choices, try them all!

Longest Common Subsequence

String Algorithms: Searching and Matching

Core problems:

- Substring search: find whether a short pattern appears inside a longer text (and where)
- Sequence Comparison
- String Similarity

Applications:

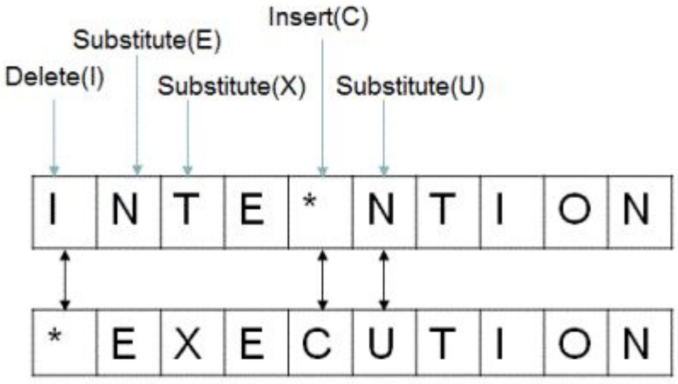
- Unix diff and git merge tools
- Code similarity and plagiarism detection
- Spell checking and fuzzy search

String Similarity

Edit Distance: given two strings, quantifies the dissimilarity between them.

Defined as the minimum number of single-character *edits* needed to transform one string into the other

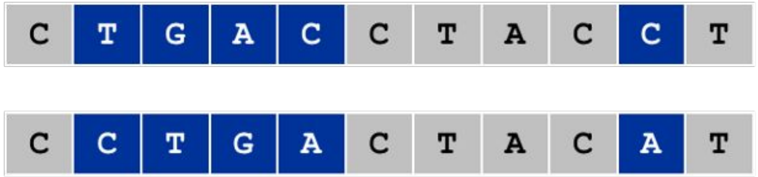
Edits can be insertions, deletions, or substitutions



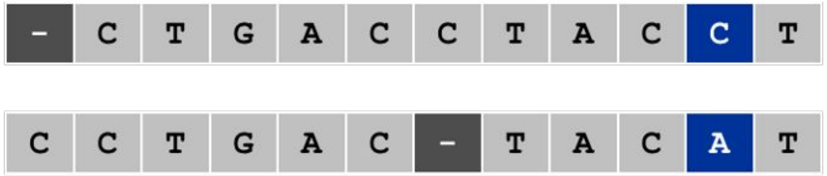
Edit Distance

The **edit distance** is the sum of:

- Gap penalty (insertions / deletions) δ
- Substitution penalty α_{pq}



$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$



$2\delta + \alpha_{CA}$

String Similarity is used frequently...

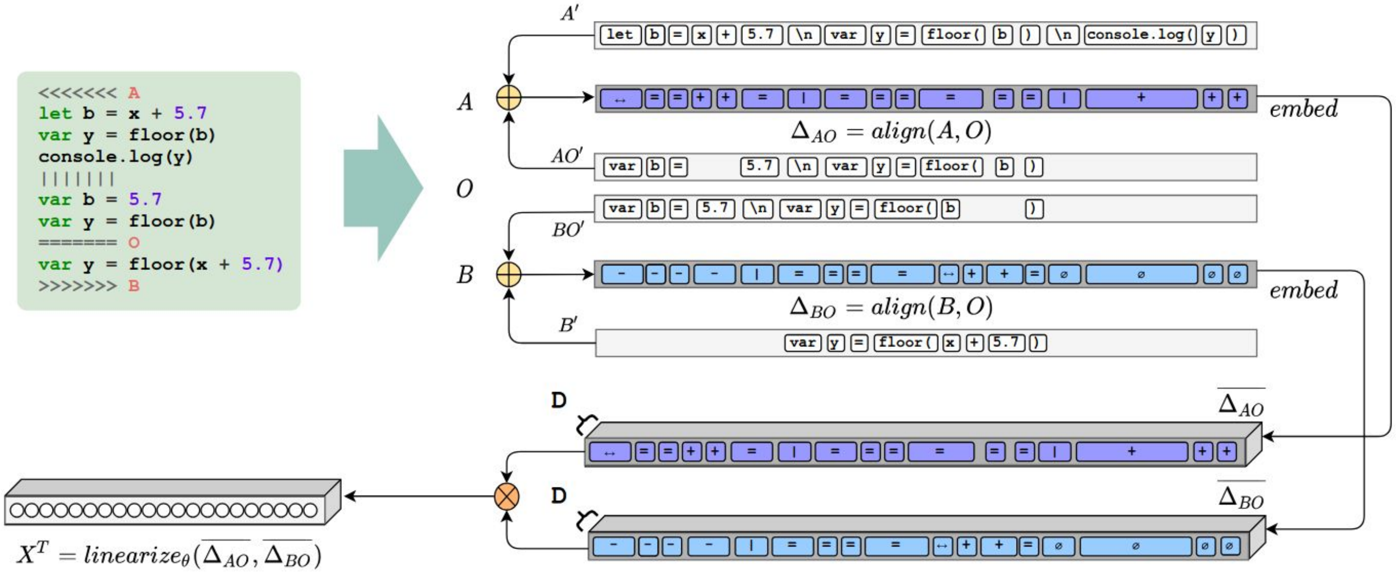


Figure 5: *Merge2Matrix*: implemented with the Aligned Linearized input representation used in DEEPMERGE.

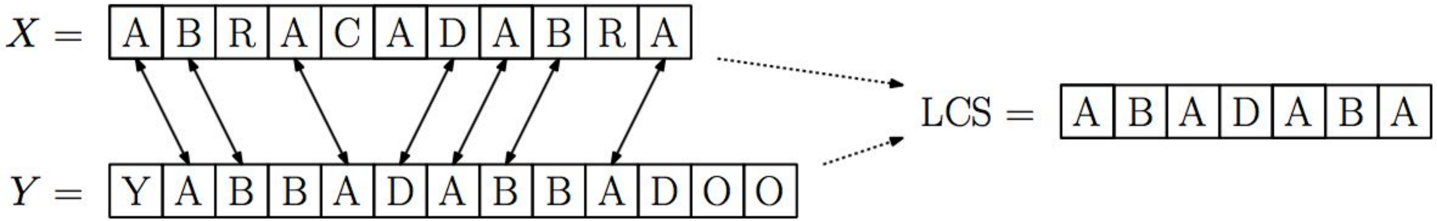
https://arxiv.org/abs/2105.07569

Is Z a subsequence of X?

- Given two character sequences $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Z = \langle z_1, z_2, \dots, z_k \rangle$, Z is a *subsequence* of X if there is a strictly increasing sequence of k indices $\langle i_1, i_2, \dots i_k \rangle, (1 \leq i_1 < i_2 \dots < i_k \leq m)$ such that $Z = \langle x_{i_1}, x_{i_2}, \dots, x_{i_k} \rangle$
- X = <ABRACADABRA>
- Z = <AADAA>

Longest Common Subsequence

- Given two strings X and Y, the *longest common subsequence* of X and Y is the longest subsequence Z that is a subsequence of both X and Y



- Not necessarily unique: <ABC> <BAC>

Longest Common Subsequence

Example 2:

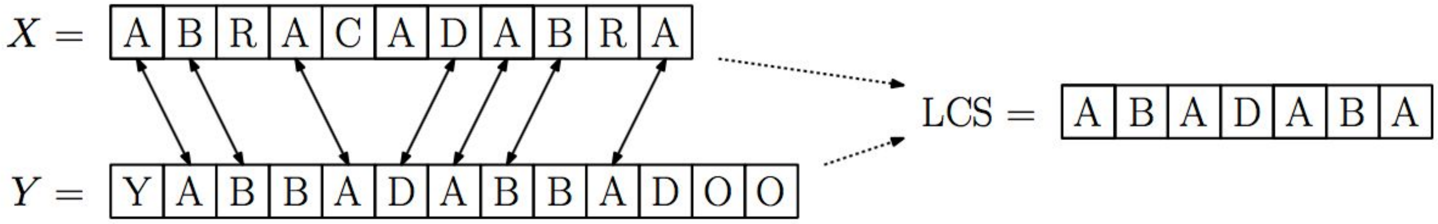
X = "ABCDGH"

Y = "AEDFHR"

Lcs = ADH

Longest Common Subsequence

Brute force?



Generate all 2^m subsequences of string X (length m)

For each subsequence, check if it's also a subsequence of Y

DP Formulation

Break down the problem by **expressing the optimal solution in terms of optimal solutions to sub-parts**

To do this, determine the recursive formulation

- First describe the precise function you want to evaluate in English
- Then, give a formal recursive definition of the function

What are sub-parts here?

Smaller parts of the strings

Longest Common Subsequence

$X =$

A	B	R	A	C	A	D	A	B	R	A

$Y =$

Y	A	B	B	A	D	A	B	B	A

A *prefix* X_i is a sequence of chars that forms the beginning of that string up to i

$X_5 =$

A	B	R	A	C

$Y_6 =$

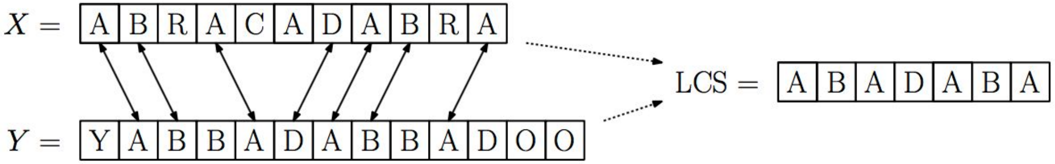
Y	A	B	B	A	D

$X_0 =$ “”

Let `lcs(i,j)` be the length of the longest common subsequence of X_i and Y_j

`lcs(5,6) =`

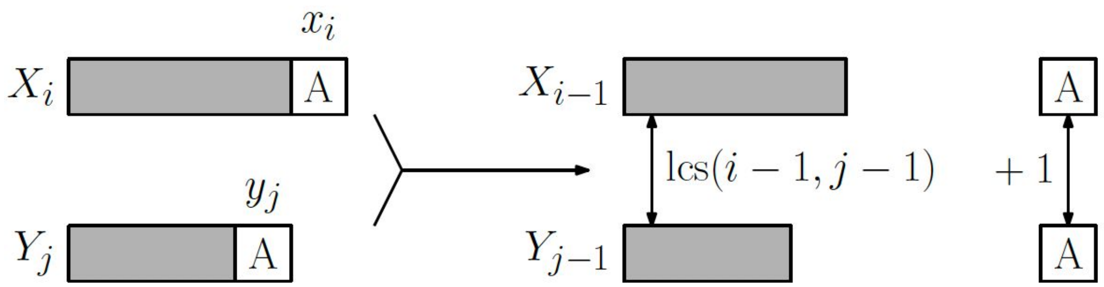
`3 (ABA)`



DP Formulation

Take it or Leave it: X_i and Y_j are either in the LCS or not

```
lcs(i, j)
    If x_i == y_j: //Take both
        lcs(i,j) = lcs(i-1, j-1) + 1
```



DP Formulation

- $x_i \neq y_j$: x_i and y_j not both in the LCS
  - x_i is not in LCS: $\text{lcs}(i,j) = \text{lcs}(i-1,j)$
  - y_j is not in LCS: $\text{lcs}(i,j) = \text{lcs}(i,j-1)$
  - $\text{lcs}(i,j) = \max(\text{lcs}(i-1,j), \text{lcs}(i,j-1))$

- $\text{lcs}(i,j) = \begin{cases} 0, & i = 0 \,||\, j = 0 \\ \text{lcs}(i-1,j-1)+1, & x_i = y_j \\ \max\big(\text{lcs}(\mathrm{i}-1,\mathrm{j}), \text{lcs}(i,j-1)\big), & x_i \neq y_j \end{cases}$

DP Formulation

Base case:

```
lcs(i,0) = 0

lcs(0,j) = 0
```

Memoized Implementation

$$\mathrm{lcs}(i,j) = \begin{cases} 0, & i = 0 \mid\mid j = 0 \\ \mathrm{lcs}(i-1, j-1) + 1, & x_i = y_j \\ \max\big(\mathrm{lcs}(\mathrm{i}-1, \mathrm{j}), \mathrm{lcs}(i, j-1)\big), & x_i \neq y_j \end{cases}$$

What does the table look like?

2D (mxn) array

|X| = m |Y| = n

Memoized Recursive Implementation

```
m-lcs(i,j) {
  if (lcs[i,j] == -1) { // not yet computed
    if ((i==0)||(j==0)) lcs[i,j] = 0
    else if (x[i]==y[j])
      lcs[i,j] = m-lcs(i-1,j-1)+1
    else
      lcs[i,j]= max(m-lcs(i-1,j), m-lcs(i,j-1))
  }
  return lcs[i,j]
}
```

Memoized Iterative Implementation

X = BACDB
Y = BDCB

```
compute-lcs(i,j) {
  for (i=0 to m) lcs[i, 0] = 0
  for (j=0 to n) lcs[0, j] = 0
  for (i=1 to m) {
    for (j=1 to n) {
      if (x[i] == y[j])
        lcs[i, j] = lcs[i-1, j-1] + 1
      else
        lcs[i, j] = max(lcs[i-1, j], lcs[i, j-1])
  }}
}
```

LCS length

		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

Runtime Analysis

O(mn)

How large are m and n?

English: m and n are around 10

Biology: m and n are around 100,000

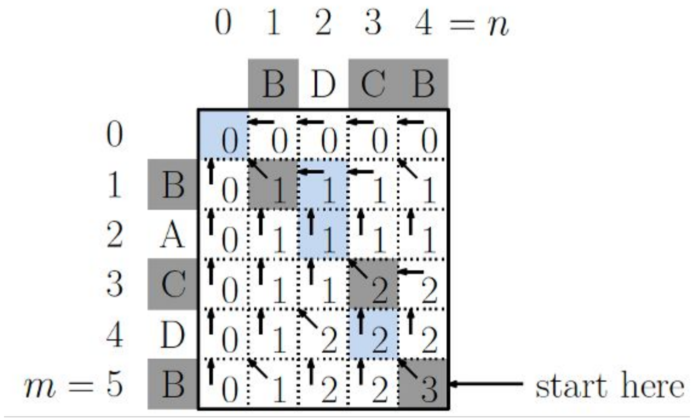
Extracting the LCS

- How do we recover the actual LCS, besides computing the length?
- When we make a decision, we need to save information to help remember it
  - `addXY`: add $x_i = y_j$ to the LCS,
    - `lcs[i-1, j-1]` $\nwarrow$
  - `skipX`: do not include x_i
    - `lcs[i-1, j]` $\uparrow$
  - `skipY`: do not include y_j
    - `lcs[i, j-1]` $\leftarrow$

Add “arrows” in a second matrix

Implementation with Solutions

```
lcs = new array[0..m, 0..n] h = new array[0..m, 0..n]
compute-lcs(i,j) {
  for (i=0 to m) {lcs[i, 0] = 0; h[i, 0] = skipX}
  for (j=0 to n) {lcs[0, j] = 0; h[0, j] = skipY}
  for (i=1 to m) {
    for (j=1 to n) {
      if (x[i] == y[j])
        {lcs[i, j]=lcs[i-1, j-1]+1; h[i, j] = addXY}
      else if (lcs[i-1, j] >= lcs[i, j-1])
        {lcs[i, j] = lcs[i-1, j]; h[i, j] = skipX}
      else
        {lcs[i, j] = lcs[i, j-1]; h[i, j] = skipY}
  }}
}
```



Correctness

- Induction to show that every table entry $\mathrm{LCS}[i,j]$ is computed correctly
  - base case: $i = j = 0$, $\mathrm{LCS}[i,j] = 0$
  - Assume claim holds for all $\mathrm{LCS}[i',j']$ where $i' + j' < i + j$
  - Consider $\mathrm{LCS}[i,j]$.
    - $x_i = y_j$, $\mathrm{LCS}[i,j] = \mathrm{LCS}[i-1,j-1] + 1$
    - otherwise, $\mathrm{LCS}[i,j] = \max(\mathrm{LCS}[i-1,j], \mathrm{LCS}[i,j-1])$

Summary

- Hw6 due Monday

- Quiz on divide and conquer tomorrow