CSCI 3100

Dynamic Programming (cont)

Announcements

- HW due Friday
  (next one posted by Fri)

Last time: LIS

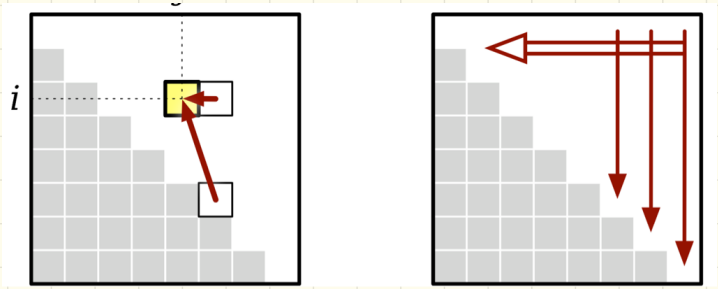
Applications:

- physics
- matrix theory
- representation theory

More theoretical:

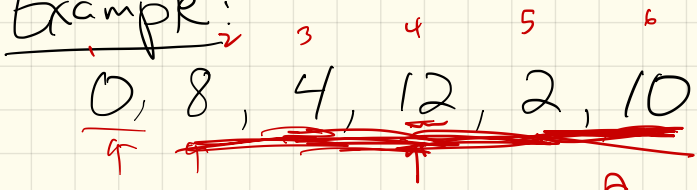
Idea:

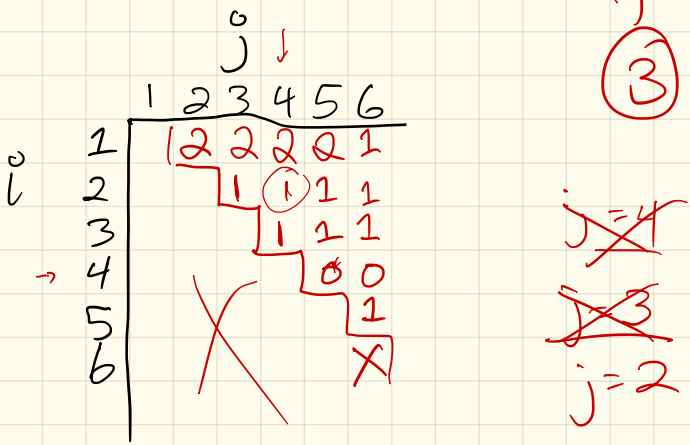
$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i, j+1),\ 1 + LIS(j, j+1)\} & \text{otherwise} \end{cases}$$



$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i, j+1),\ 1 + LIS(j, j+1)\} & \text{otherwise} \end{cases}$$

Example:





Recap: Dynamic programming is just smart recursion.

- Recurse - don't repeat

Often computed values are stored in some table for later lookups

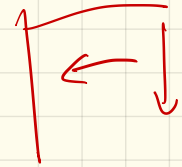
- or -

Can rearrange to fill table from ground up.

Steps:

1. Formulate the recursion
2. Build solution from base case up.
   - identify subproblems
   - identify dependencies:
     ie: F(n) depended on F(n-1) & F(n-2)
   - choose data structure
     ie: often matrix or F(n) some temps
   - choose evaluation order
   - write pseudo code, then analyze time/space

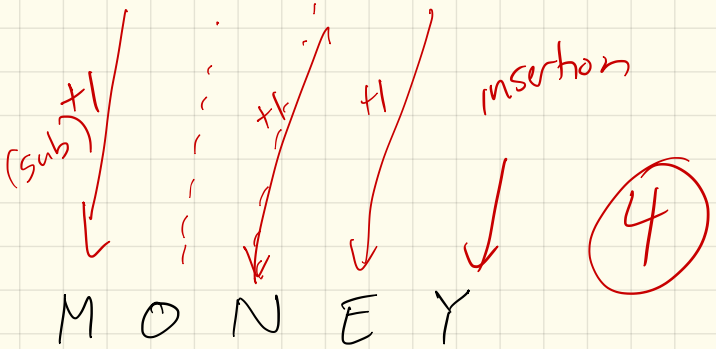
LIS



Edit Distance

The minimum number of deletions, insertions, or substitutions of letters to transform between two strings.

Ex: F O O D

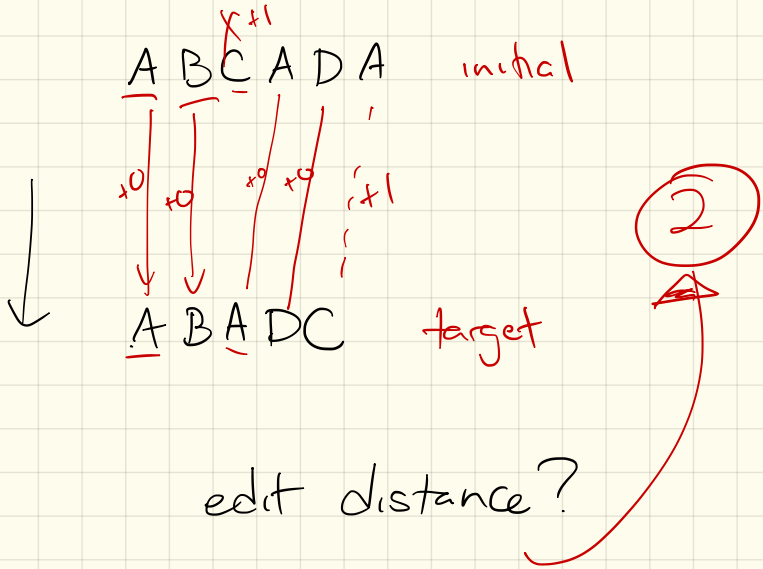


Uses? spell checkers

sequence alignment

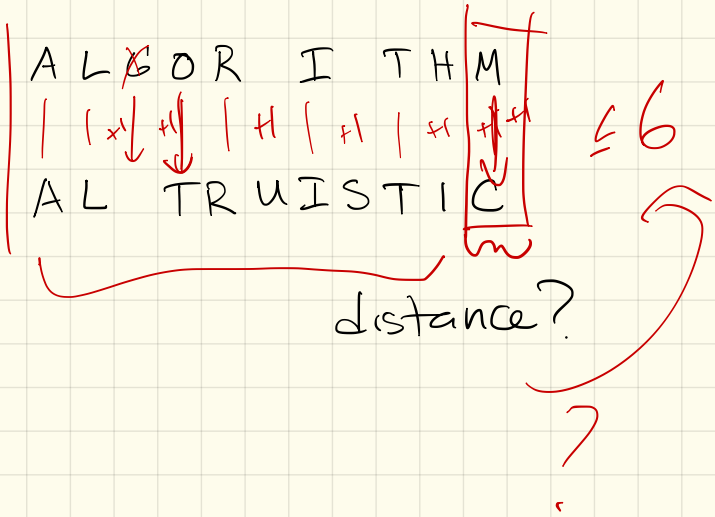
Don't be greedy!

The temptation is to do this as you go:



How to solve:

Aligning/matching will help:

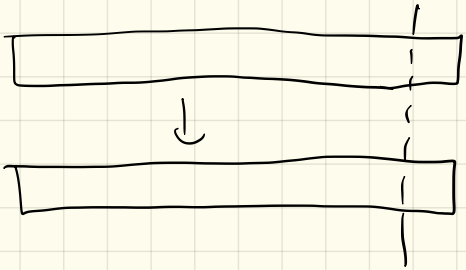


Recursive formulation:

If I align like this, can observe:

If you delete last (aligned) column, the rest will still be optimal for shorter substrings edit distance.

Why?



C — C
↓ ↓ +
C C —

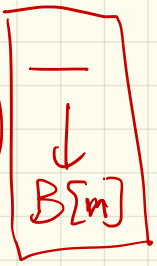
Turning this into a matrix:

Let EDIT(A[1..m], B[1..n]) be edit distance b/t A & B.

When we choose how to align, 3 possibilities:

- insertion:

EDIT(A[1..m], B[1..n-1]) + 1

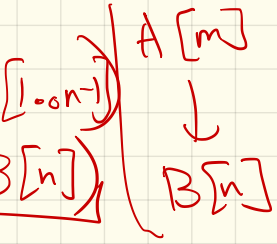


- deletion:

EDIT(A[1..m-1], B[1..n]) + 1

- substitution:

EDIT(A[1..m-1], B[1..n-1]) + (A[m] = B[n])



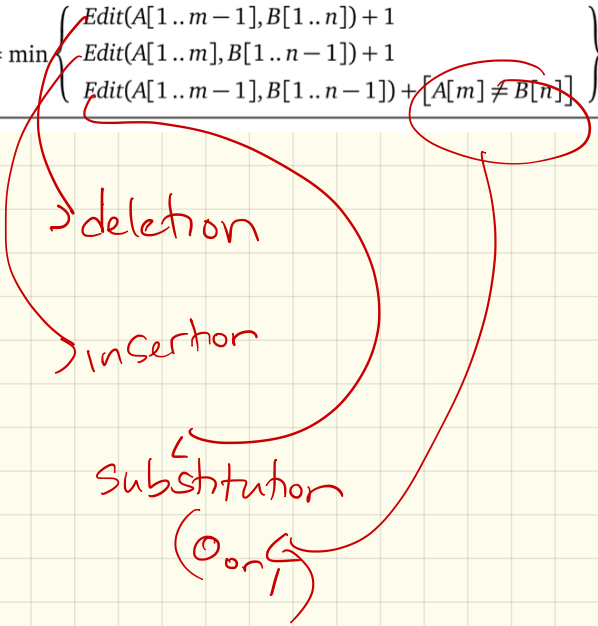
0 or 1

if equal, +0
if not, +1

C ↓ X

Turn this into recursion:

$$Edit(A[1..m], B[1..n]) = \min \begin{cases} Edit(A[1..m-1], B[1..n]) + 1 \\ Edit(A[1..m], B[1..n-1]) + 1 \\ Edit(A[1..m-1], B[1..n-1]) + [A[m] \neq B[n]] \end{cases}$$



Turning this into a proper recursion:

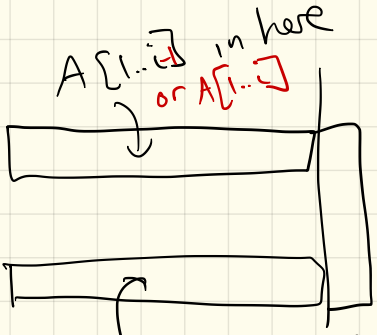
Let EDIT(i, j) := edit distance between:

$A[1..i]$

$B[1..j]$

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{l} Edit(i-1,j)+1, \\ Edit(i,j-1)+1, \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

last slide



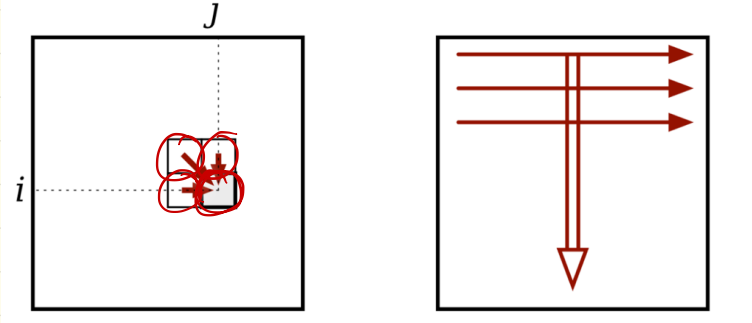
Now, don't bother analyzing the recursion.

(It's awful!)

Instead, be smart:

memoize!

Table:



Algorithm:

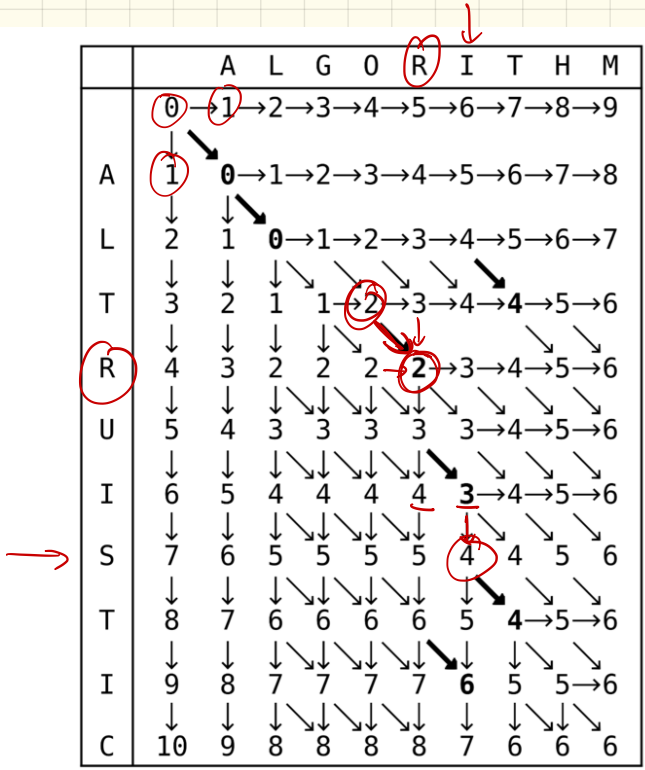
```
EditDistance(A[1..m], B[1..n]):
  for j ← 1 to n
      Edit[0, j] ← j

  for i ← 1 to m
      Edit[i, 0] ← i
      for j ← 1 to n
          if A[i] = B[j]
              Edit[i, j] ← min {Edit[i − 1, j] + 1, Edit[i, j − 1] + 1, Edit[i − 1, j − 1]}
          else
              Edit[i, j] ← min {Edit[i − 1, j] + 1, Edit[i, j − 1] + 1, Edit[i − 1, j − 1] + 1}

  return Edit[m, n]
```

(analyze next time)

Example:



		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

The memoization table for *Edit*(ALGORITHM, ALTRUISTIC)

```
A L G O R   I   T H M
A L T   R U I S T I C
```