

Pre-lecture brain teaser

What is the running time of the following algorithm:

Consider computing $f(x, y)$ by recursive function + memoization.

$$f(x, y) = \sum_{i=1}^{\min(x,y)} x * f(x + y - i, i - 1),$$
$$f(0, y) = y \quad f(x, 0) = x.$$

The resulting algorithm when computing $f(n, n)$ would take:

- (a) $O(n^2)$
- (b) $O(n^3)$
- (c) $O(2^n)$
- (d) $O(n^n)$
- (e) The function is ill defined - it can not be computed.

ECE-374-B: Lecture 14 - Dynamic Programming II

Instructor: Nickvash Kani

March 07, 2023

University of Illinois at Urbana-Champaign

Pre-lecture brain teaser

What is the running time of the following algorithm:

Consider computing $f(x, y)$ by recursive function + memoization.

Handwritten: $O(n)$
 $f(x, y)$
 $f(x+y-1, 1)$

$$f(x, y) = \sum_{i=1}^{\min(x, y)} x * f(x + y - i, i - 1),$$

Handwritten: can never be more than $\min(x, y)$

$$f(0, y) = y \quad f(x, 0) = x.$$

Handwritten: will always be less than $x+y$

Handwritten: $f(x+y-1, 0)$
 $f(y-1, x)$

The resulting algorithm when computing $f(n, n)$ would take:

- (a) $O(n^2)$
- (b) $O(n^3)$
- (c) $O(2^n)$
- (d) $O(n^n)$

Handwritten: # of subproblems $A(n) = O(n^2)$

Handwritten: amount of work / recursive call $B(n) = O(n)$

$$A(n) \cdot B(n) = O(n^3)$$

- (e) The function is ill defined - it can not be computed.

Recipe for Dynamic Programming

1. Develop a recursive backtracking style algorithm \mathcal{A} for given problem.
2. Identify structure of subproblems generated by \mathcal{A} on an instance I of size n
 - 2.1 Estimate **number of different subproblems** generated as a function of n . Is it polynomial or exponential in n ?
 - 2.2 If the number of problems is “small” (polynomial) then they typically have some “clean” structure.
3. Rewrite subproblems in a compact fashion.
4. Rewrite recursive algorithm in terms of notation for subproblems.
5. Convert to iterative algorithm by bottom up evaluation in an appropriate order.
6. Optimize further with data structures and/or additional ideas.

Why is it called dynamic programming?

Dynamic programming was a technique “invented” by Richard Bellman. From his autobiography:

*I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? **In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word “programming”. I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying I thought, lets kill two birds with one stone. Lets take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.***

Edit Distance and Sequence Alignment

Spell Checking Problem

Given a string “exponen” that is not in the dictionary, how should a spell checker suggest a nearby string?

Spell Checking Problem

Given a string “exponent”^{exponent} that is not in the dictionary, how should a spell checker suggest a nearby string?

What does nearness mean?

exponent

Question: Given two strings $x_1x_2 \dots x_n$ and $y_1y_2 \dots y_m$ what is a distance between them?

Spell Checking Problem

Given a string “exponen” that is not in the dictionary, how should a spell checker suggest a nearby string?

What does nearness mean?

Question: Given two strings $x_1x_2 \dots x_n$ and $y_1y_2 \dots y_m$ what is a distance between them?

Edit Distance: minimum number of “edits” to transform x into y .

Edit Distance

Definition

Edit distance between two words X and Y is the number of letter **insertions**, letter **deletions** and letter **substitutions** required to obtain Y from X .

Example

The edit distance between FOOD and MONEY is at least 4:

FOOD \rightarrow MOOD \rightarrow MONOD \rightarrow MONED \rightarrow MONEY

↘ ↓ ↘ ↘

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

F	O	O		D
M	O	N	E	Y

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

F	O	O		D
M	O	N	E	Y

Formally, an **alignment** is a set M of pairs (i, j) such that each index appears at most once, and there is no “crossing”: $i < i'$ and i is matched to j implies i' is matched to $j' > j$. In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (4, 5)\}$.

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

			δ	α
F	O	O		D
M	O	N	E	Y

Formally, an **alignment** is a set M of pairs (i, j) such that each index appears at most once, and there is no “crossing”: $i < i'$ and i is matched to j implies i' is matched to $j' > j$. In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (4, 5)\}$. Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.

Edit Distance Problem

Problem

Given two words, find the edit distance between them, i.e., an alignment of smallest cost.

Applications

- Spell-checkers and Dictionaries
- Unix `diff`
- DNA sequence alignment ... but, we need a new metric

Sequence alignment problem - Similarity Metric

Definition

For two strings X and Y , the cost of alignment M is

- [Gap penalty] For each gap in the alignment, we incur a cost δ .
- [Mismatch cost] For each pair p and q that have been matched in M , we incur cost α_{pq} ; typically $\alpha_{pp} = 0$.

Sequence alignment problem - Similarity Metric

Definition

For two strings X and Y , the cost of alignment M is

- [Gap penalty] For each gap in the alignment, we incur a cost δ .
- [Mismatch cost] For each pair p and q that have been matched in M , we incur cost α_{pq} ; typically $\alpha_{pp} = 0$.

Edit distance is special case when $\delta = \alpha_{pq} = 1$.

Edit distance as alignment

What is the edit distance between...

What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost 1 unit?

374

473

(a) 1

(b) 2

(c) 3

(d) 4

(e) 5

What is the edit distance between...

What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost 1 unit?

373

473

(a) 1

(b) 2

(c) 3

(d) 4

(e) 5

What is the edit distance between...

What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost 1 unit?

37

473

3 7
4 7 3

4 7 ~~3~~ 3 7

1. 1

2. 2

3. 3

4. 4

5. 5

Sequence Alignment

Input Given two words X and Y , and gap penalty δ and mismatch costs α_{pq}

Goal Find alignment of minimum cost

Edit distance: The algorithm

Edit distance - Basic observation

Let $X = \alpha x$ and $Y = \beta y$

α, β : strings.

x and y single characters.

Think about optimal edit distance between X and Y as alignment, and consider last column of alignment of the two strings:

α	x	or	α	x	or	αx	
β	y		βy			β	y

Prefixes must have optimal alignment!

Problem Structure

Let $X = x_1x_2 \cdots x_m$ and $Y = y_1y_2 \cdots y_n$. If (m, n) are not matched then either the m^{th} position of X remains unmatched or the n^{th} position of Y remains unmatched.

- **Case** x_m and y_n are matched. *Substitution*
 - Pay mismatch cost $\alpha_{x_my_n}$ plus cost of aligning strings $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_{n-1}$
- **Case** x_m is unmatched. *Deletion*
 - Pay gap penalty plus cost of aligning $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_n$
- **Case** y_n is unmatched. *Insertion*
 - Pay gap penalty plus cost of aligning $x_1 \cdots x_m$ and $y_1 \cdots y_{n-1}$

Subproblems and Recurrence

$x_1 \dots x_{i-1}$	x_i
$y_1 \dots y_{j-1}$	y_j

or

$x_1 \dots x_{i-1}$	x
$y_1 \dots y_{j-1}y_j$	

or

$x_1 \dots x_{i-1}x_i$	
$y_1 \dots y_{j-1}$	y_j

Optimal Costs

Let $\text{Opt}(i, j)$ be optimal cost of aligning $x_1 \dots x_i$ and $y_1 \dots y_j$.

Then

$$\text{Opt}(i, j) = \min \begin{cases} \alpha_{x_i y_j} + \text{Opt}(i-1, j-1), \\ \delta + \text{Opt}(i-1, j), \\ \delta + \text{Opt}(i, j-1) \end{cases}$$

eapple
apple

Subproblems and Recurrence

$x_1 \dots x_{i-1}$	x_i	or	$x_1 \dots x_{i-1}$	x	or	$x_1 \dots x_{i-1} x_i$	
$y_1 \dots y_{j-1}$	y_j		$y_1 \dots y_{j-1} y_j$			$y_1 \dots y_{j-1}$	y_j

Optimal Costs

Let $\text{Opt}(i, j)$ be optimal cost of aligning $x_1 \dots x_i$ and $y_1 \dots y_j$.

Then

$$\text{Opt}(i, j) = \min \begin{cases} \alpha_{x_i y_j} + \text{Opt}(i - 1, j - 1), \\ \delta + \text{Opt}(i - 1, j), \\ \delta + \text{Opt}(i, j - 1) \end{cases}$$

Base Cases: $\text{Opt}(i, 0) = \delta \cdot i$ and $\text{Opt}(0, j) = \delta \cdot j$

$x_1 \dots x_i$

Recursive Algorithm

Assume X is stored in array $A[1..m]$ and Y is stored in $B[1..n]$
Array $COST$ stores cost of matching two chars. Thus $COST[a, b]$
give the cost of matching character a to character b .

```
EDIST(A[1..m], B[1..n])
  If ( $m = 0$ ) return  $n\delta$ 
  If ( $n = 0$ ) return  $m\delta$ 
   $m_1 = \delta + EDIST(A[1..(m - 1)], B[1..n])$ 
   $m_2 = \delta + EDIST(A[1..m], B[1..(n - 1)])$ 
   $m_3 = COST[A[m], B[n]] + EDIST(A[1..(m - 1)], B[1..(n - 1)])$ 
  return  $\min(m_1, m_2, m_3)$ 
```

Example: DEED and DREAD

	ϵ	D	R	E	A	D
ϵ						
D						
E						
E						
D						

↙ ↘

$$\text{Opt}(i, j) =$$

$$\min \begin{cases} \alpha_{x_i y_j} + \text{Opt}(i-1, j-1), \\ \delta + \text{Opt}(i-1, j), \\ \delta + \text{Opt}(i, j-1) \end{cases}$$

Base Cases:

- $\text{Opt}(i, 0) = \delta \cdot i$
- $\text{Opt}(0, j) = \delta \cdot j$

Example: DEED and DREAD

	ε	D	R	E	A	D
ε	0	1	2	3	4	5
D	1					
E	2					
E	3					
D	4					

$\text{Opt}(i, j) =$

$$\min \begin{cases} \alpha_{x_i y_j} + \text{Opt}(i-1, j-1), \\ \delta + \text{Opt}(i-1, j), \\ \delta + \text{Opt}(i, j-1) \end{cases}$$

Base Cases:

- $\text{Opt}(i, 0) = \delta \cdot i$
- $\text{Opt}(0, j) = \delta \cdot j$

Example: DEED and DREAD

	ϵ	D	R	E	A	D
ϵ	0	1	2	3	4	5
D	1	0	1	2	3	4
E	2	1	1	1	2	3
E	3					
D	3					

$\text{Opt}(i, j) =$

$$\min \begin{cases} \alpha_{x_i y_j} + \text{Opt}(i-1, j-1), \\ \delta + \text{Opt}(i-1, j), \\ \delta + \text{Opt}(i, j-1) \end{cases}$$

Base Cases:

- $\text{Opt}(i, 0) = \delta \cdot i$
- $\text{Opt}(0, j) = \delta \cdot j$

$\text{Opt}(DE, DRE)$
 $\begin{matrix} D & E \\ D & R & E \end{matrix}$

Example: DEED and DREAD

	ε	D	R	E	A	D
ε	0	1	2	3	4	5
D	1	0	1	2	3	4
E	2	1	1	1	2	3
E	3	2	2	1	2	3
D	3					

$\text{Opt}(i, j) =$

$$\min \begin{cases} \alpha_{x_i y_j} + \text{Opt}(i-1, j-1), \\ \delta + \text{Opt}(i-1, j), \\ \delta + \text{Opt}(i, j-1) \end{cases}$$

Base Cases:

- $\text{Opt}(i, 0) = \delta \cdot i$
- $\text{Opt}(0, j) = \delta \cdot j$

Example: DEED and DREAD

	ϵ	D	R	E	A	D
ϵ	0	1	2	3	4	5
D	1	0	1	2	3	4
E	2	1	1	1	2	3
E	3	2	2	1	2	3
D	4	3	3	2	2	2

D	R	E	A	D
D	E	E		D

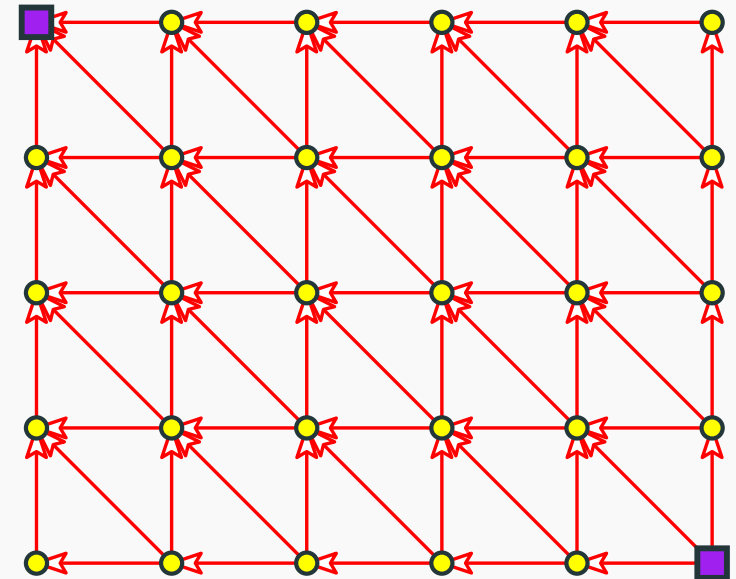
$$\rightarrow O_p + (4, 5)$$

$$= 2 \text{ (DEED, DREAD)}$$

Example: DEED and DREAD

	ε	<i>D</i>	<i>R</i>	<i>E</i>	<i>A</i>	<i>D</i>
ε	0	1	2	3	4	5
<i>D</i>	1	0	1	2	3	4
<i>E</i>	2	1	1	1	2	3
<i>E</i>	3	2	2	1	2	3
<i>D</i>	3	3	3	2	2	2

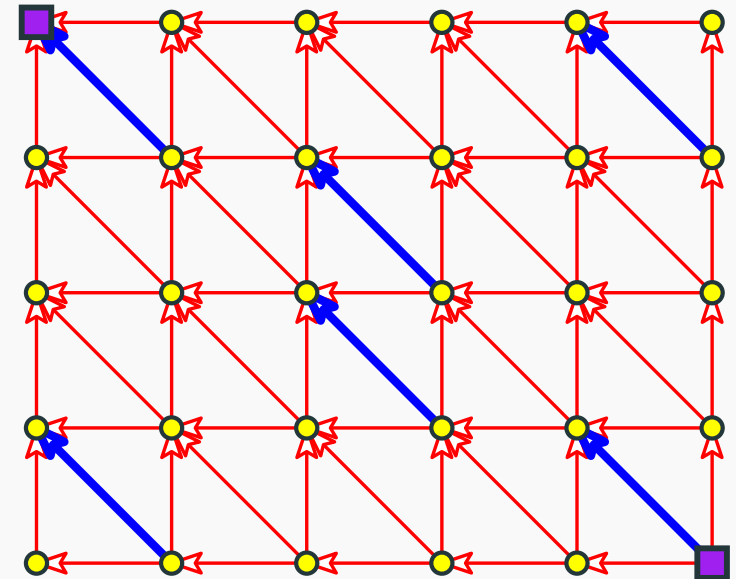
<i>D</i>	<i>R</i>	<i>E</i>	<i>A</i>	<i>D</i>
<i>D</i>	<i>E</i>	<i>E</i>		<i>D</i>



Example: DEED and DREAD

	ε	<i>D</i>	<i>R</i>	<i>E</i>	<i>A</i>	<i>D</i>
ε	0	1	2	3	4	5
<i>D</i>	1	0	1	2	3	4
<i>E</i>	2	1	1	1	2	3
<i>E</i>	3	2	2	1	2	3
<i>D</i>	3	3	3	2	2	2

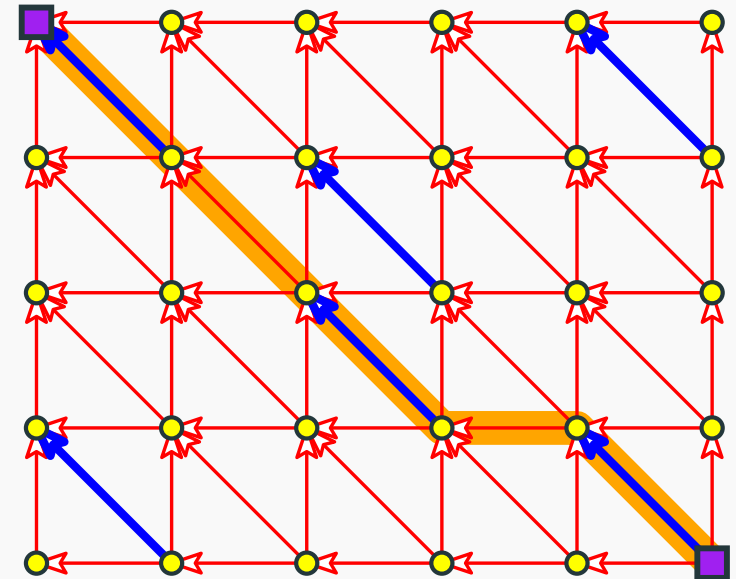
<i>D</i>	<i>R</i>	<i>E</i>	<i>A</i>	<i>D</i>
<i>D</i>	<i>E</i>	<i>E</i>		<i>D</i>



Example: DEED and DREAD

	ε	<i>D</i>	<i>R</i>	<i>E</i>	<i>A</i>	<i>D</i>
ε	0	1	2	3	4	5
<i>D</i>	1	0	1	2	3	4
<i>E</i>	2	1	1	1	2	3
<i>E</i>	3	2	2	1	2	3
<i>D</i>	3	3	3	2	2	2

<i>D</i>	<i>R</i>	<i>E</i>	<i>A</i>	<i>D</i>
<i>D</i>	<i>E</i>	<i>E</i>		<i>D</i>



Dynamic programming algorithm for edit-distance

As part of the input...

The cost of aligning a character against another character

Σ : Alphabet

We are given a cost function (in a table):

$$\forall b, c \in \Sigma \quad \text{COST}[b][c] = \text{cost of aligning } b \text{ with } c. \quad /$$

$$\forall b \in \Sigma \quad \text{COST}[b][b] = 0$$

δ : price of deletion of insertion of a single character /

Dynamic program for edit distance

```
EDIST(A[1..m], B[1..n])
```

```
int M[0..m][0..n]
```

```
for i = 1 to m do M[i, 0] = iδ
```

```
for j = 1 to n do M[0, j] = jδ
```

```
for i = 1 to m do
```

```
  for j = 1 to n do
```

$$M[i][j] = \min \begin{cases} \text{COST}[A[i]][B[j]] + M[i-1][j-1], \\ \delta + M[i-1][j], \\ \delta + M[i][j-1] \end{cases}$$

memoized array

Base Cases

calculate recurrence row by row, left to right

Dynamic program for edit distance

```
EDIST(A[1..m], B[1..n])
  int M[0..m][0..n]
  for i = 1 to m do M[i, 0] = iδ
  for j = 1 to n do M[0, j] = jδ

  for i = 1 to m do
    for j = 1 to n do
      M[i][j] = min { COST[A[i]][B[j]] + M[i-1][j-1],
                    δ + M[i-1][j],
                    δ + M[i][j-1]
```

Analysis

- Running time is $O(mn)$
- Space used is $O(mn)$

Reducing space for edit distance

Matrix and DAG of computation of edit distance

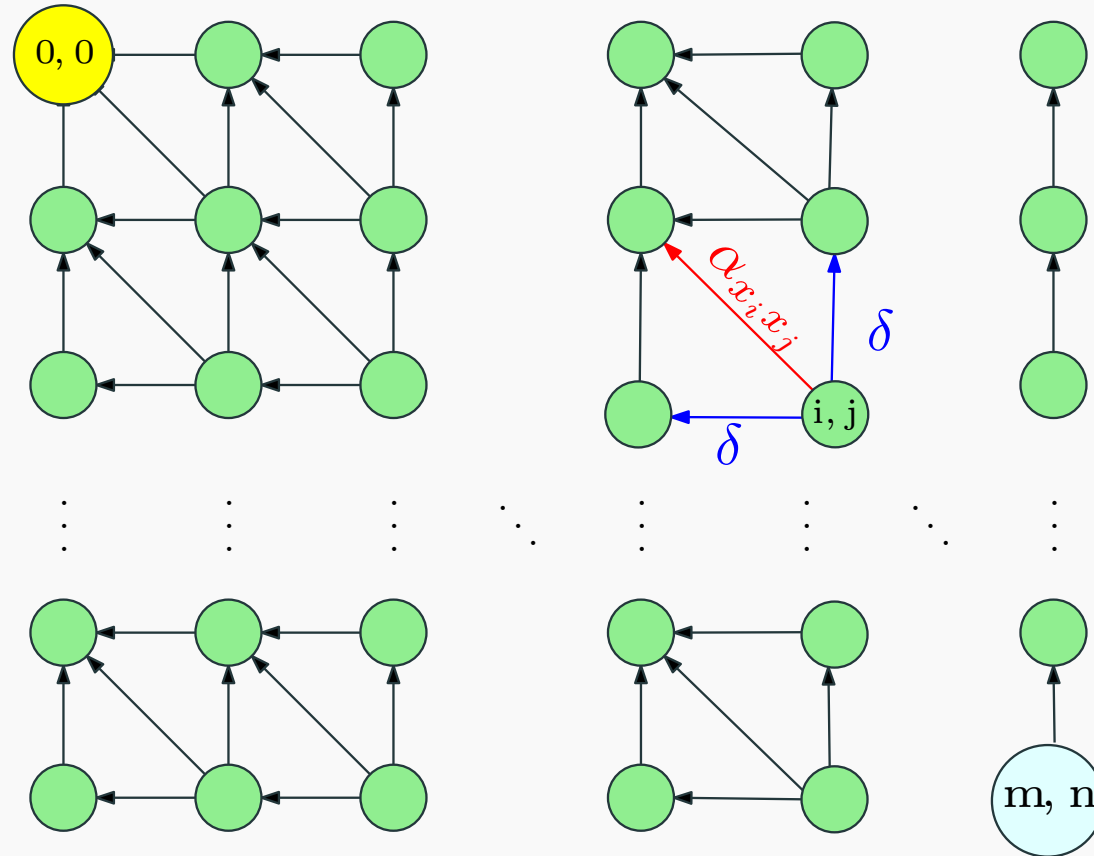


Figure 1: Iterative algorithm in previous slide computes values in row order.

Optimizing Space

- Recall

$$M(i, j) = \min \begin{cases} \alpha_{x_i y_j} + M(i - 1, j - 1), \\ \delta + M(i - 1, j), \\ \delta + M(i, j - 1) \end{cases}$$

- Entries in j^{th} column only depend on $(j - 1)^{\text{st}}$ column and earlier entries in j^{th} column
- Only store the current column and the previous column reusing space; $N(i, 0)$ stores $M(i, j - 1)$ and $N(i, 1)$ stores $M(i, j)$

Example: DEED vs. DREAD filled by column

	ϵ	D	R	E	A	D
ϵ						
D						
E						
E						
D						

Example: DEED vs. DREAD filled by column

	ε	<i>D</i>	<i>R</i>	<i>E</i>	<i>A</i>	<i>D</i>
ε	0	1	2	3	4	5
<i>D</i>	1					
<i>E</i>	2					
<i>E</i>	3					
<i>D</i>	3					

Example: DEED vs. DREAD filled by column

	ε	D	R	E	A	D
ε	0	1	2	3	4	5
D	1	0				
E	2	1				
E	3	2				
D	3	3				

Example: DEED vs. DREAD filled by column

	ϵ	D	R	E	A	D
ϵ	0	1	2	3	4	5
D	1	0	1			
E	2	1	1			
E	3	2	2			
D	3	3	3			

Example: DEED vs. DREAD filled by column

	ϵ	D	R	E	A	D
ϵ	0	1	2	3	4	5
D	1	0	1	2		
E	2	1	1	1		
E	3	2	2	1		
D	3	3	3	2		

Example: DEED vs. DREAD filled by column

	ϵ	<i>D</i>	<i>R</i>	<i>E</i>	<i>A</i>	<i>D</i>
ϵ	0	1	2	3	4	5
<i>D</i>	1	0	1	2	3	3
<i>E</i>	2	1	1	1	2	2
<i>E</i>	3	2	2	1	2	2
<i>D</i>	3	3	3	2	2	3

Example: DEED vs. DREAD filled by column

	ϵ	<i>D</i>	<i>R</i>	<i>E</i>	<i>A</i>	<i>D</i>
ϵ	0	1	2	3	4	5
<i>D</i>	1	0	1	2	3	4
<i>E</i>	2	1	1	1	2	3
<i>E</i>	3	2	2	1	2	3
<i>D</i>	3	3	3	2	2	2

Computing in column order to save space

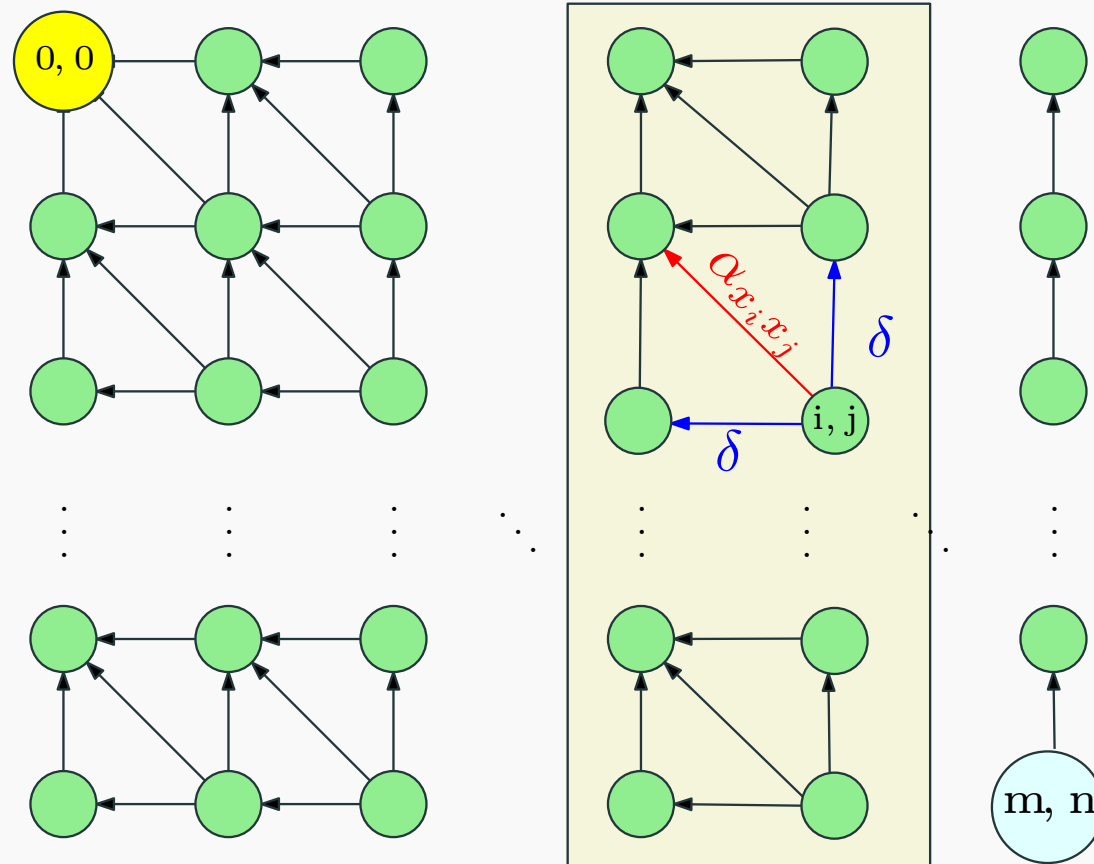


Figure 2: $M(i, j)$ only depends on previous column values. Keep only two columns and compute in column order.

Space Efficient Algorithm

```
for all  $i$  do  $N[i, 0] = i\delta$ 
for  $j = 1$  to  $n$  do
   $N[0, 1] = j\delta$  (* corresponds to  $M(0, j)$  *)
  for  $i = 1$  to  $m$  do
    
$$N[i, 1] = \min \begin{cases} \alpha_{x_i y_j} + N[i - 1, 0] \\ \delta + N[i - 1, 1] \\ \delta + N[i, 0] \end{cases}$$

  for  $i = 1$  to  $m$  do
    Copy  $N[i, 0] = N[i, 1]$ 
```

Analysis

Running time is $O(mn)$ and space used is $O(2m) = O(m)$

$$O(\min(m, n))$$

Analyzing Space Efficiency

- From the $m \times n$ matrix M we can construct the actual alignment (exercise)
- Matrix N computes cost of optimal alignment but no way to construct the actual alignment
- Space efficient computation of alignment? More complicated algorithm — see notes and Kleinberg-Tardos book.

Longest Common Subsequence Problem

LCS Problem

Definition

LCS between two strings X and Y is the length of longest common subsequence between X and Y .

ABAZDC
BACBAD
BAO

ABAZDC
BACBAD
ABD

LCS Problem

Definition

LCS between two strings X and Y is the length of longest common subsequence between X and Y .

ABAZDC
BACBAD



ABAZDC
BACBAD

Example

LCS between ABAZDC and BACBAD is 4 via ABAD

LCS Problem

Definition

LCS between two strings X and Y is the length of longest common subsequence between X and Y .

ABAZDC
BACBAD

ABAZDC
BACBAD

Example

LCS between *ABAZDC* and *BACBAD* is 4 via *ABAD*

Derive a dynamic programming algorithm for the problem.

How do we plan out the recursion?

How do we plan out the recursion?

Start off with $A[1..m]$ and $B[1..n]$ and reason the following:

How do we plan out the recursion?

Start off with $A[1..m]$ and $B[1..n]$ and reason the following:

- Assuming $A[m] \neq B[n]$
 - The one or neither of the end characters are in the LCS.
Therefore becomes:

$$\max(LCS(A[1..m-1], B[1..n]), LCS(A[1..m], B[1..n-1]))$$

How do we plan out the recursion?

Start off with $A[1..m]$ and $B[1..n]$ and reason the following:

- Assuming $A[m] \neq B[n]$
 - The one or neither of the end characters are in the LCS.
Therefore becomes:

$$\max(LCS(A[1..m-1], B[1..n]), LCS(A[1..m], B[1..n-1]))$$

- Assuming $A[m] = B[n]$

How do we plan out the recursion?

Start off with $A[1\dots m]$ and $B[1\dots n]$ and reason the following:

- Assuming $A[m] \neq B[n]$
 - The one or neither of the end characters are in the LCS. Therefore becomes:

$$\max(LCS(A[1\dots m - 1], B[1\dots n]), LCS(A[1\dots m], B[1\dots n - 1]))$$

- Assuming $A[m] = B[n]$
 - Either $A[m]$ and $B[n]$ are both in the LCS. Therefore:

$$LCS(A[1\dots m], B[1\dots n]) = 1 + LCS(A[1\dots m - 1], B[1\dots n - 1])$$

How do we plan out the recursion?

Start off with $A[1\dots m]$ and $B[1\dots n]$ and reason the following:

- Assuming $A[m] \neq B[n]$
 - The one or neither of the end characters are in the LCS. Therefore becomes:

$$\max(LCS(A[1\dots m - 1], B[1\dots n]), LCS(A[1\dots m], B[1\dots n - 1]))$$

- Assuming $A[m] = B[n]$
 - Either $A[m]$ and $B[n]$ are both in the LCS. Therefore:

$$LCS(A[1\dots m], B[1\dots n]) = 1 + LCS(A[1\dots m - 1], B[1\dots n - 1])$$

- Or $A[m]$ and $B[n]$ is not in the LCS. Therefore the LCS is either:

$$LCS(A[1\dots m - 1], B[1\dots n])$$

$$LCS(A[1\dots m], B[1\dots n - 1])$$

How do we plan out the recursion?

Start off with $A[1..m]$ and $B[1..n]$ and reason the following:

- Assuming $A[m] \neq B[n]$
 - The one or neither of the end characters are in the LCS. Therefore becomes:

$$\max(LCS(A[1..m-1], B[1..n]), LCS(A[1..m], B[1..n-1]))$$

- Assuming $A[m] = B[n]$
 - Either $A[m]$ and $B[n]$ are both in the LCS. Therefore:

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

- Or $A[m]$ and $B[n]$ is not in the LCS. Therefore the LCS is either:

$$LCS(A[1..m-1], B[1..n])$$

$$LCS(A[1..m], B[1..n-1])$$

- Base Case: If $A = \epsilon$ or $B = \epsilon$ then $LCS = 0$

LCS recursive definition

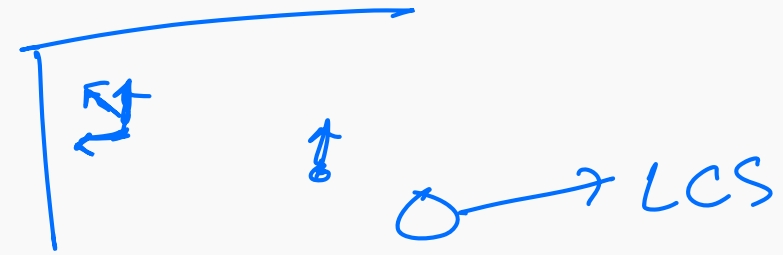
$A[1..n], B[1..m]$: Input strings.

$$LCS(i, j) = \begin{cases} 0 & \text{Base} & i = 0 \text{ or } j = 0 \\ \max \begin{pmatrix} LCS(i-1, j), \\ LCS(i, j-1) \end{pmatrix} & & A[i] \neq B[j] \\ \max \begin{pmatrix} LCS(i-1, j), \\ LCS(i, j-1), \\ 1 + LCS(i-1, j-1) \end{pmatrix} & & A[i] = B[j] \end{cases}$$

LCS recursive definition

$A[1..n], B[1..m]$: Input strings.

$$LCS(i, j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ \max \begin{pmatrix} LCS(i-1, j), \\ LCS(i, j-1) \end{pmatrix} & A[i] \neq B[j] \\ \max \begin{pmatrix} LCS(i-1, j), \\ LCS(i, j-1), \\ 1 + LCS(i-1, j-1) \end{pmatrix} & A[i] = B[j] \end{cases}$$



Running time: $O(n \cdot m)$

Space: $O(n \cdot m)$

$O(m)$

$A(n) = O(n^2)$

$B(n) = O(1)$

Longest common subsequence is just edit distance for the two sequences...

A, B : input sequences, Σ : “alphabet” all the different values in A and B

$$\forall b, c \in \Sigma : b \neq c$$

$$COST[b][c] = +\infty.$$

$$\forall b \in \Sigma$$

$$COST[b][b] = 1$$

1 : price of deletion or insertion of a single character

Longest common subsequence is just edit distance for the two sequences...

A, B : input sequences, Σ : “alphabet” all the different values in A and B

$$\forall b, c \in \Sigma : b \neq c$$

$$COST[b][c] = +\infty.$$

$$\forall b \in \Sigma$$

$$COST[b][b] = 1$$

1 : price of deletion or insertion of a single character

											ED	LCS
Maximum ED		D	R	E	A	D					9	0
Min LCS							D	E	E	D		
Sub-opt ED		D	R	E	A	D					8	1
Sub-opt LCS						D	E	E	D			
Min ED		D	R	E	A		D				6	3
Max LCS		D		E		E	D					

Longest common subsequence is just edit distance for the two sequences...

A, B : input sequences, Σ : “alphabet” all the different values in A and B

$$\forall b, c \in \Sigma : b \neq c$$

$$COST[b][c] = +\infty.$$

$$\forall b \in \Sigma$$

$$COST[b][b] = 1$$

1 : price of deletion of insertion of a single character

Length of longest common sub-sequence =

Is in L^k ?

A variation

Input A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function **IsStringinL**(string x) that decides whether x is in L , and non-negative integer k

Goal Decide if $w \in L^k$ using **IsStringinL**(string x) as a black box sub-routine

Example

Suppose L is *English* and we have a procedure to check whether a string/word is in the *English* dictionary.

- Is the string “isthisanenglishsentence” in *English*⁵?
- Is the string “isthisanenglishsentence” in *English*⁴?
- Is “asinineat” in *English*²?
- Is “asinineat” in *English*⁴?
- Is “zibzzzad” in *English*¹?

Recursive Solution

When is $w \in L^k$?

Recursive Solution

$$\text{When is } w \in L^k? \quad \text{IsStringinLk}(A[1 \dots i], k) = \begin{cases} w \in L^k \text{ iff } w = \epsilon & k = 0 \\ w \in L^k \text{ iff } w \in L & k = 1 \\ w \in L^k \text{ iff } \begin{array}{l} u \in L^{k-1}, \\ v \in L \end{array} \text{ and } |w = uv & k > 1 \end{cases}$$

IsStringinLk(A[1...i], k):

if $k = 0$ and $i = 0$ then return YES

if $k = 0$ then return NO // $i > 0$

if $k = 1$ then

return **IsStringinL**(A[1...i])

for $\ell = 1 \dots i - 1$ do

if **IsStringinLk**(A[1... ℓ], $k - 1$) and **IsStringinL**(A[$\ell + 1 \dots i$]) then
return YES

return NO

Analysis

```
IsStringinLk(A[1...i], k):  
  if k = 0 and i = 0 then return YES  
  if k = 0 then return NO // i > 0  
  if k = 1 then  
    return IsStringinL(A[1...i])  
  
  for ℓ = 1...i - 1 do  
    if IsStringinLk(A[1...ℓ], k - 1) and IsStringinL(A[ℓ + 1...i]) then  
      return YES  
  
  return NO
```

- How many distinct sub-problems are generated by **IsStringinLk(A[1..n], k)**?

Analysis

```
IsStringinLk(A[1...i], k):  
  if k = 0 and i = 0 then return YES  
  if k = 0 then return NO // i > 0  
  if k = 1 then  
    return IsStringinL(A[1...i])  
  
  for ℓ = 1...i - 1 do  
    if IsStringinLk(A[1...ℓ], k - 1) and IsStringinL(A[ℓ + 1...i]) then  
      return YES  
  
  return NO
```

- How many distinct sub-problems are generated by **IsStringinLk**(A[1..n], k)? $O(nk)$

Analysis

```
IsStringinLk(A[1...i], k):  
  if k = 0 and i = 0 then return YES  
  if k = 0 then return NO // i > 0  
  if k = 1 then  
    return IsStringinL(A[1...i])  
  
  for l = 1...i-1 do  
    if IsStringinLk(A[1...l], k-1) and IsStringinL(A[l+1...i]) then  
      return YES  
  
  return NO
```

- How many distinct sub-problems are generated by **IsStringinLk(A[1..n], k)**? $O(nk)$
- How much space?

Analysis

```
IsStringinLk(A[1...i], k):  
  if k = 0 and i = 0 then return YES  
  if k = 0 then return NO // i > 0  
  if k = 1 then  
    return IsStringinL(A[1...i])  
  
  for ℓ = 1...i-1 do  
    if IsStringinLk(A[1...ℓ], k-1) and IsStringinL(A[ℓ+1...i]) then  
      return YES  
  
  return NO
```

- How many distinct sub-problems are generated by **IsStringinLk**(A[1..n], k)? $O(nk)$
- How much space? $O(nk)$

Analysis

```
IsStringinLk(A[1...i], k):  
  if k = 0 and i = 0 then return YES  
  if k = 0 then return NO // i > 0  
  if k = 1 then  
    return IsStringinL(A[1...i])  
  
  for l = 1...i-1 do  
    if IsStringinLk(A[1...l], k-1) and IsStringinL(A[l+1...i]) then  
      return YES  
  
  return NO
```

- How many distinct sub-problems are generated by **IsStringinLk**(A[1..n], k)? $O(nk)$
- How much space? $O(nk)$
- Running time if we use memoization?

Analysis

```
IsStringinLk(A[1...i], k):
```

```
  if  $k = 0$  and  $i = 0$  then return YES
```

```
  if  $k = 0$  then return NO //  $i > 0$ 
```

```
  if  $k = 1$  then
```

```
    return IsStringinL(A[1...i])
```

```
  for  $\ell = 1 \dots i - 1$  do
```

```
    if IsStringinLk(A[1... $\ell$ ],  $k - 1$ ) and IsStringinL(A[ $\ell + 1 \dots i$ ]) then  
      return YES
```

```
  return NO
```

- How many distinct sub-problems are generated by **IsStringinLk**(A[1..n], k)? $O(nk)$
- How much space? $O(nk)$
- Running time if we use memoization? $O(n^2k)$