طراحی الگوریتم ها (CE221)

جلسه سیزدهم: برنامه نویسی پویا

سجاد شیرعلی شهرضا

بهار 1401

دوشنبه، 22 فروردین 1401

Based on Stanford CS161 Slides from Summer 2021 by Karey Shi

اطلاع رسانی

- بخشِ مرتبط کتاب برای این جلسه: 15
- یادآوری مهلت ارسال تمرین دوم: 8 صبح روز شنبه 27 فروردین 1401

مقدمه ای بر برنامه نویسی پویا

یک روش طراحی الگوریتم

DYNAMIC PROGRAMMING

Dynamic programming (DP) is an algorithm design paradigm.
It's often used to solve optimization problems (e.g. *shortest* path).

DYNAMIC PROGRAMMING

Dynamic programming (DP) is an algorithm design paradigm.
It's often used to solve optimization problems (e.g. *shortest* path).

We'll see two examples of DP today:
Bellman-Ford and Floyd-Warshall algorithms.
We will go over some DP practice problems in depth next week.

But first, an overview of DP!

DYNAMIC PROGRAMMING

Elements of dynamic programming:

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

e.g. $\mathbf{d^{(k)}[b] = \min\{ d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a, b)\} \}}$

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.
e.g. $\mathbf{d^{(k)}[b] = \min\{ d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a, b)\} \}}$

Overlapping sub-problems: the subproblems overlap a lot!
This means we can save time by solving a sub-problem once & cache the answer.
(this is sometimes called “memoization”)

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

e.g. $\mathbf{d^{(k)}[b] = \min\{ d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a, b)\} \}}$

Overlapping sub-problems: the subproblems overlap a lot!

This means we can save time by solving a sub-problem once & cache the answer.

(this is sometimes called "memoization")

e.g. **Lots of different entries in the row $d^{(k)}$ may ask for $d^{(k-1)}[v]$**

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).

e.g. **Bellman-Ford (as we will see shortly!) computes $d^{(0)}$, then $d^{(1)}$, then $d^{(2)}$, etc.**

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).
e.g. **Bellman-Ford (as we will see soon!) computes $d^{(0)}$, then $d^{(1)}$, then $d^{(2)}$, etc.**

Top-down: instead uses recursive calls to solve smaller problems, while using memoization/caching to keep track of small problems that you've already computed answers for (simply fetch the answer instead of re-solving that problem and waste computational effort)

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).
e.g. **Bellman-Ford (as we will see soon!) computes $d^{(0)}$, then $d^{(1)}$, then $d^{(2)}$, etc.**

Top-down: instead uses recursive calls to solve smaller problems, while using memoization/caching to keep track of small problems that you've already computed answers for (simply fetch the answer instead of re-solving that problem and waste computational effort)
We will see a way later to implement **Bellman-Ford** using a top-down approach.

DYNAMIC PROGRAMMING

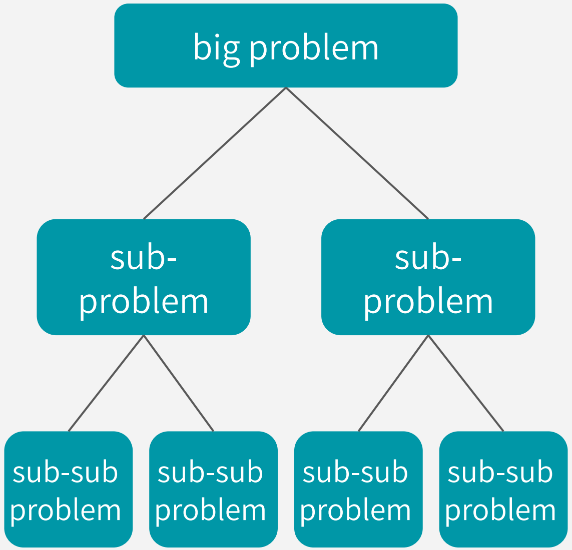
Why “dynamic programming”?

Richard Bellman invented the term in the 1950’s. He was working for the RAND corporation at the time, which was employed by the Air Force, and government projects needed flashy non-mathematical non-researchy names to get funded and approved.

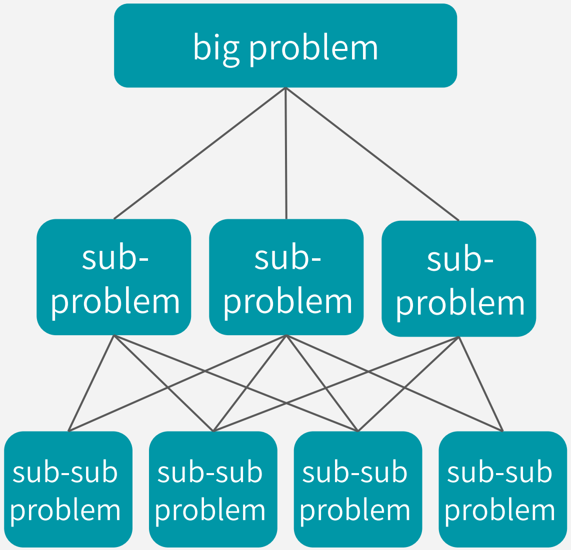
“It’s impossible to use the word dynamic in a pejorative sense…
I thought dynamic programming was a good name.
It was something not even a Congressman could object to.”

DIVIDE & CONQUER vs DP

DIVIDE-AND-CONQUER



DYNAMIC PROGRAMMING



سوال؟

RECIPE FOR APPLYING DP

RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?

RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*

RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.

RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc*. Go back and modify your algorithm in step 3 to make this happen.

سوال؟

پیدا کردن بزرگ ترین زیر رشته مشترک

LONGEST COMMON SUBSEQUENCE

A sequence **Z** is a **SUBSEQUENCE** of **X** if **Z** can be obtained from **X** by deleting symbols

BDFH is a subsequence of **ABCDEFGH**

C is a subsequence of **ABCDEFGH**

ABCDEFGH is a subsequence of **ABCDEFGH**

LONGEST COMMON SUBSEQUENCE

A sequence **Z** is a **SUBSEQUENCE** of **X** if **Z** can be obtained from **X** by deleting symbols

BDFH is a subsequence of **ABCDEFGH**

C is a subsequence of **ABCDEFGH**

ABCDEFGH is a subsequence of **ABCDEFGH**

A sequence **Z** is a **LONGEST COMMON SUBSEQUENCE (LCS)** of **X** and **Y** if **Z** is a subsequence of both **X** and **Y** and any sequence longer than **Z** is not a subsequence of at least one of **X** or **Y**.

ABDFGH is the LCS of

ABCDEFGH and **ABDFGHI**

LONGEST COMMON SUBSEQUENCE

A sequence **Z** is a **SUBSEQUENCE** of **X** if **Z** can be obtained from **X** by deleting symbols

BDFH is a subsequence of **ABCDEFGH**

C is a subsequence of **ABCDEFGH**

A se nd **Y**

TASK: Given sequences X and Y, find the length of their LCS, Z.

(Later, we'll also output Z, but we'll start off with the length)

and ar **X** or **Y**.

ABDFGH is the LCS of

ABCDEFGH and **ABDFGHI**

APPLICATIONS OF LCS

Bioinformatics!

Detect similarities between DNA or protein sequences

Computational linguistics!

Extract similarities in words/word-forms and determine how words are related

the `diff` unix command!

Identify differences between the contents of two files

and so much more...

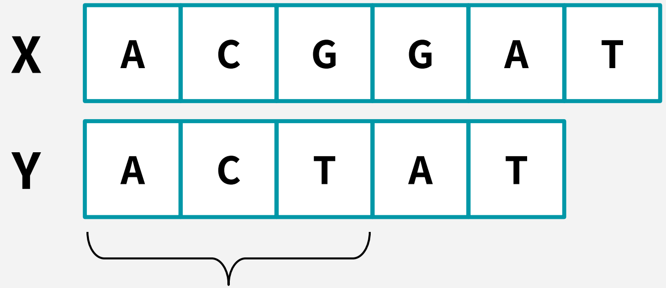
سوال؟

LCS: RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc*. Go back and modify your algorithm in step 3 to make this happen.

STEP 1: OPTIMAL SUBSTRUCTURE

SUBPROBLEM: Find the length of LCS's of *prefixes* to X and Y.



Notation: denote this prefix **ACT** by $\mathbf{Y_3}$

STEP 1: OPTIMAL SUBSTRUCTURE

SUBPROBLEM: Find the length of LCS's of *prefixes* to X and Y.

X	A	C	G	G	A	T
Y	A	C	T	A	T	

Notation: denote this prefix **ACT** by Y_3

Examples:

C[2,3] = 2 (LCS of X_2 and Y_3 is AC)
C[5,4] = 3 (LCS of X_5 and Y_4 is ACA)

Let **C[i, j] = length_of_LCS(X_i, Y_j)**

STEP 1: OPTIMAL SUBSTRUCTURE

SUBPROBLEM: Find the length of LCS's of *prefixes* to X and Y.

X	**A**	**C**	**G**	**G**	**A**	**T**
Y	**A**	**C**	**T**	**A**	**T**	

Notation: denote this prefix **ACT** by $\mathbf{Y_3}$

Examples:

C[2,3] = 2 (LCS of X_2 and Y_3 is AC)
C[5,4] = 3 (LCS of X_5 and Y_4 is ACA)

Let **C[i, j] = length_of_LCS($\mathbf{X_i}$, $\mathbf{Y_j}$)**

Why is this a good choice?

STEP 1: OPTIMAL SUBSTRUCTURE

Let **$C[i, j]$ = length_of_LCS(X_i, Y_j)**

Consider the ends of our prefixes, X[i] and Y[j]. We have two cases:

Case 1: X[i] = Y[j]

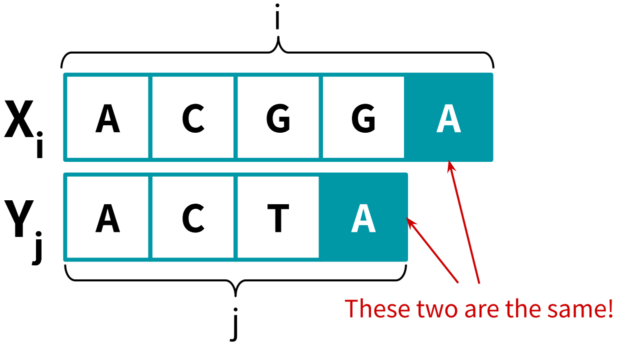
Case 2: X[i] ≠ Y[j]

STEP 1: OPTIMAL SUBSTRUCTURE

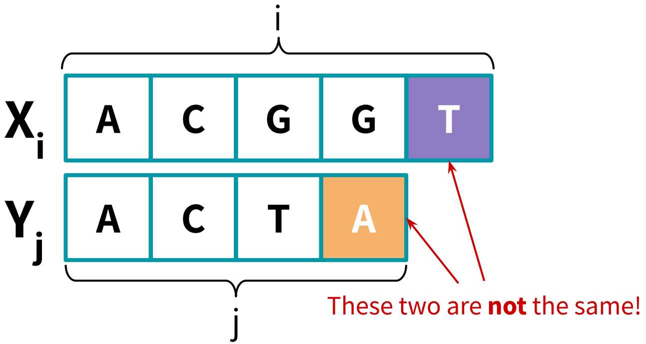
Let **C[i, j] = length_of_LCS(X_i, Y_j)**

Consider the ends of our prefixes, X[i] and Y[j]. We have two cases:

Case 1: X[i] = Y[j]



Case 2: X[i] ≠ Y[j]

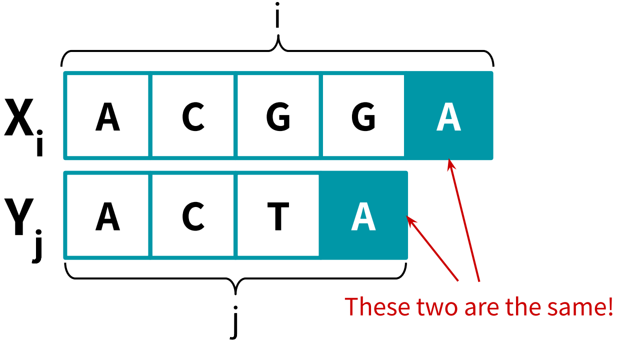


STEP 1: OPTIMAL SUBSTRUCTURE

Let **C[i, j] = length_of_LCS(X_i, Y_j)**

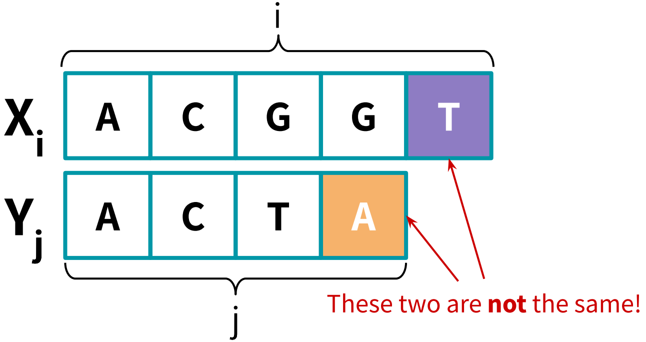
Consider the ends of our prefixes, X[i] and Y[j]. We have two cases:

Case 1: X[i] = Y[j]



What is **C[i, j]**?

Case 2: X[i] ≠ Y[j]



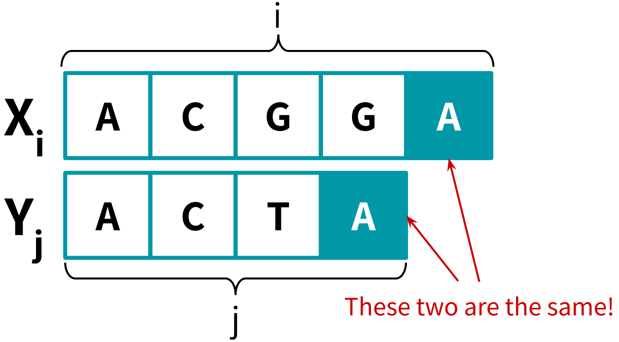
What is **C[i, j]**?

STEP 1: OPTIMAL SUBSTRUCTURE

Let **C[i, j] = length_of_LCS(X_i, Y_j)**

Consider the ends of our prefixes, X[i] and Y[j]. We have two cases:

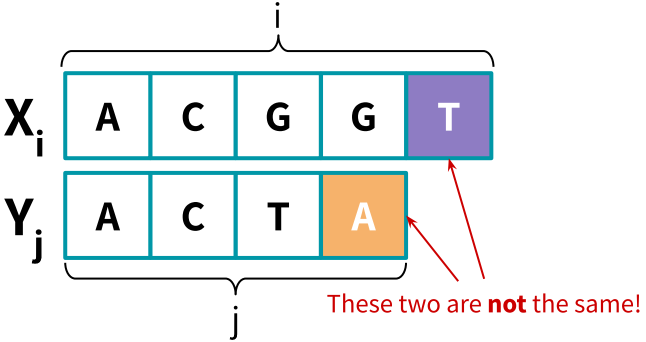
Case 1: X[i] = Y[j]



Then, **C[i, j] = 1 + C[i–1, j–1]**

because LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1}) followed by **A** .

Case 2: X[i] ≠ Y[j]

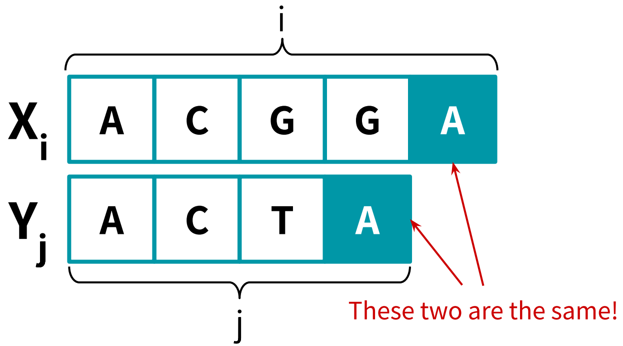


STEP 1: OPTIMAL SUBSTRUCTURE

Let **C[i, j] = length_of_LCS(X_i, Y_j)**

Consider the ends of our prefixes, X[i] and Y[j]. We have two cases:

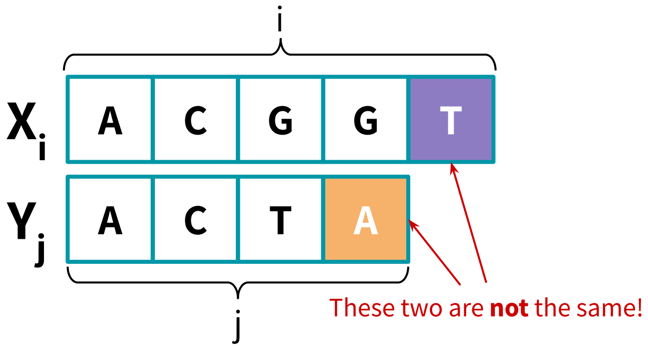
Case 1: X[i] = Y[j]



Then, **C[i, j] = 1 + C[i–1, j–1]**

because LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1}) followed by **A** .

Case 2: X[i] ≠ Y[j]



Then, **C[i, j] = max{ C[i–1, j], C[i, j–1] }**

Give **A** a chance to “match”: LCS(X_i, Y_j) = LCS(X_{i-1}, Y)

Give **T** a chance to “match”: LCS(X_i, Y_j) = LCS(X_i, Y_{j-1})

سوال؟

LCS: RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc*. Go back and modify your algorithm in step 3 to make this happen.

STEP 2: RECURSIVE FORMULATION

Our recursive formulation:

$$
\mathbf{C[\,i, j\,]} \;=\; \begin{cases} \mathbf{0} & \text{if } i = 0 \text{ or } j = 0 \\ \mathbf{C[i-1, j-1] + 1} & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \mathbf{max\{\;C[i-1, j],\;\; C[i, j-1]\;\}} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}
$$

STEP 2: RECURSIVE FORMULATION

Our recursive formulation:

$$
\mathbf{C[\,i, j\,]} \;=\; \begin{cases} \mathbf{0} & \text{if } i = 0 \text{ or } j = 0 \\ \mathbf{C[i\text{–}1, j\text{–}1] + 1} & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \mathbf{max\{\ C[i\text{–}1, j],\ \ C[i, j\text{–}1]\ \}} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}
$$

CASE 0 (base case)

X_0

Y_j	A	C	T	A

STEP 2: RECURSIVE FORMULATION

Our recursive formulation:

$$
\mathbf{C[\,i,j\,]} \;=\; \begin{cases} \mathbf{0} & \text{if } i = 0 \text{ or } j = 0 \\ \mathbf{C[i-1,j-1]+1} & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \mathbf{max\{\;C[i-1,j],\;\; C[i,j-1]\;\}} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}
$$

CASE 0 (base case)

X_0

Y_j	A	C	T	A

CASE 1

X_i	A	C	G	G	A
Y_j	A	C	T	A	

STEP 2: RECURSIVE FORMULATION

Our recursive formulation:

$$
C[i, j] = \begin{cases} \mathbf{0} & \text{if } i = 0 \text{ or } j = 0 \\ \mathbf{C[i-1, j-1] + 1} & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \mathbf{max\{ \ C[i-1, j], \ C[i, j-1] \ \}} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}
$$

CASE 0 (base case)

X_0				
Y_j	A	C	T	A

CASE 1

X_i	A	C	G	G	A
Y_j	A	C	T	A	

CASE 2

X_i	A	C	G	G	T
Y_j	A	C	T	A	

سوال؟

LCS: RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc*. Go back and modify your algorithm in step 3 to make this happen.

STEP 3: WRITE A DP ALGORITHM

We’ll store answers to our subproblems C[i, j] in a table (this is our cache)!

Now that we’ve defined our recursive formulation, translating to appropriate pseudocode is straightforward: establish your base cases & define your cases!

We’ll do this in a bottom-up fashion. Why?

We know we need answers to shorter prefixes first, and it’s pretty easy to iterate in order and fill out answers to smaller prefixes before building up to longer prefixes (ultimately getting our final answer C[m, n] where |X| = m, and |Y| = n)

STEP 3: WRITE A DP ALGORITHM

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\ C[i-1, j],\ C[i, j-1]\ \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

len(X) = m & len(Y) = n

```
LCS(X,Y):
    Initialize an (m+1) x (n+1) 0-indexed array C
    C[i,0] = C[0,j] = 0 for all i=0,...,m and j=0,...,n
    for i = 1,...,m  and j = 1,...,n:
        if X[i] = Y[j]:
            C[i,j] = C[i-1,j-1] + 1
        else:
            C[i,j] = max{ C[i,j-1], C[i-1,j] }
    return C[m,n]
```

STEP 3: WRITE A DP ALGORITHM

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\ C[i-1, j],\ C[i, j-1]\ \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

```
LCS(X,Y):                  len(X) = m & len(Y) = n
    Initialize an (m+1) x (n+1) 0-indexed array C
    C[i,0] = C[0,j] = 0 for all i=0,...,m and j=0,...,n
    for i = 1,...,m  and j = 1,...,n:
        if X[i] = Y[j]:
            C[i,j] = C[i-1,j-1] + 1
        else:
            C[i,j] = max{ C[i,j-1], C[i-1,j] }
    return C[m,n]
```

Make sure that our base cases are set up

STEP 3: WRITE A DP ALGORITHM

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\ C[i-1,j],\ C[i,j-1]\ \} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

len(X) = m & len(Y) = n

```
LCS(X,Y):
    Initialize an (m+1) x (n+1) 0-indexed array C
    C[i,0] = C[0,j] = 0 for all i=0,...,m and j=0,...,n
    for i = 1,...,m  and j = 1,...,n:
        if X[i] = Y[j]:
            C[i,j] = C[i-1,j-1] + 1
        else:
            C[i,j] = max{ C[i,j-1], C[i-1,j] }
    return C[m,n]
```

Make sure that our base cases are set up

Case 1

Case 2

STEP 3: WRITE A DP ALGORITHM

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\ C[i-1, j],\ C[i, j-1]\ \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

```
LCS(X,Y):
    Initialize an (m+1) x (n+1) 0-indexed array C
    C[i,0] = C[0,j] = 0 for all i=0,...,m and j=0,...,n
    for i = 1,...,m  and j = 1,...,n:
        if X[i] = Y[j]:
            C[i,j] = C[i-1,j-1] + 1
        else:
            C[i,j] = max{ C[i,j-1], C[i-1,j] }
    return C[m,n]
```

len(X) = m & len(Y) = n

Make sure that our base cases are set up

Case 1

Case 2

Runtime: O(mn)

Constant amount of work to fill out each of the mn entries in C

سوال؟

EXAMPLE

Y

X

		A	C	T	G
	0	0	0	0	0
A	0				
C	0				
G	0				
G	0				
A	0				

```
Initialize an (m+1) x (n+1) 0-indexed array C
   C[i,0] = C[0,j] = 0 for all i=0,...,m and j=0,...,n
```

Fill in our base cases first

EXAMPLE

		Y: A	C	T	G
	0	0	0	0	0
X: A	0	1			
C	0				
G	0				
G	0				
A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] = Y[j]

C[i,j] = C[i-1,j-1] + 1

EXAMPLE

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1		
C	0				
G	0				
G	0				
A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	
C	0				
G	0				
G	0				
A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

Y

			A	C	T	G
		0	0	0	0	0
	A	0	1	1	1	1
	C	0				
X	G	0				
	G	0				
	A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

Y

X \ Y		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1			
G	0				
G	0				
A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2		
G	0				
G	0				
A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] = Y[j]

C[i,j] = C[i-1,j-1] + 1

EXAMPLE

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	
G	0				
G	0				
A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

	Y		A	C	T	G
		0	0	0	0	0
	A	0	1	1	1	1
	C	0	1	2	2	2
X	G	0				
	G	0				
	A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1			
G	0				
A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2		
G	0				
A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	
G	0				
A	0				

```
for i = 1,...,m  and j = 1,...,n:
    if X[i] = Y[j]:
        C[i,j] = C[i-1,j-1] + 1
    else:
        C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

Y

			A	C	T	G
		0	0	0	0	0
	A	0	1	1	1	1
	C	0	1	2	2	2
X	G	0	1	2	2	3
	G	0				
	A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] = Y[j]

C[i,j] = C[i-1,j-1] + 1

EXAMPLE

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1			
A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2		
A	0				

```
for i = 1,...,m  and j = 1,...,n:
    if X[i] = Y[j]:
        C[i,j] = C[i-1,j-1] + 1
    else:
        C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

Y

X

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	
A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

Y

X			A	C	T	G
		0	0	0	0	0
	A	0	1	1	1	1
	C	0	1	2	2	2
	G	0	1	2	2	3
	G	0	1	2	2	3
	A	0				

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] = Y[j]

C[i,j] = C[i-1,j-1] + 1

EXAMPLE

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1			

```
for i = 1,...,m  and j = 1,...,n:
    if X[i] = Y[j]:
        C[i,j] = C[i-1,j-1] + 1
    else:
        C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] = Y[j]

C[i,j] = C[i-1,j-1] + 1

EXAMPLE

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2		

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

			A	C	T	G
		0	0	0	0	0
	A	0	1	1	1	1
	C	0	1	2	2	2
X	G	0	1	2	2	3
	G	0	1	2	2	3
	A	0	1	2	2	

Y

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

X[i] ≠ Y[j]

C[i,j] = max{ C[i,j-1], C[i-1,j] }

EXAMPLE

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

```
for i = 1,...,m  and j = 1,...,n:
    if X[i] = Y[j]:
        C[i,j] = C[i-1,j-1] + 1
    else:
        C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

So the LCM of X and Y has length 3.

سوال؟

LCS: RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc*. Go back and modify your algorithm in step 3 to make this happen.

STEP 4: FIND ACTUAL LCS

X \ Y		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

Suppose we want to recover the actual LCS.

How can we construct the actual LCS given this table C that we just filled out?

STEP 4: FIND ACTUAL LCS

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

Suppose we want to recover the actual LCS.

How can we construct the actual LCS given this table C that we just filled out?

We'll start at C[m,n] and work backwards to trace out how we ended up with a 3 as our answer!

If we see that the character in X matches the character in Y, then we mark that character as part of our LCS and take a *diagonal step* backwards.

Otherwise, if the characters don't match, we just simply take a step towards the larger adjacent entry

STEP 4: FIND ACTUAL LCS

		Y: A	C	T	G
	0	0	0	0	0
X: A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

```
RECOVER_LCS(X, Y, C):
    // C is already filled out
    L = []
    i = m
    j = n
    while i > 0 and j > 0:
        if X[i] = Y[j]:
            append X[i] to the beginning of L
            i = i-1
            j = j-1
        else if C[i,j] = C[i,j-1]:
            j = j-1
        else:
            i = i-1
    return L
```

STEP 4: FIND ACTUAL LCS

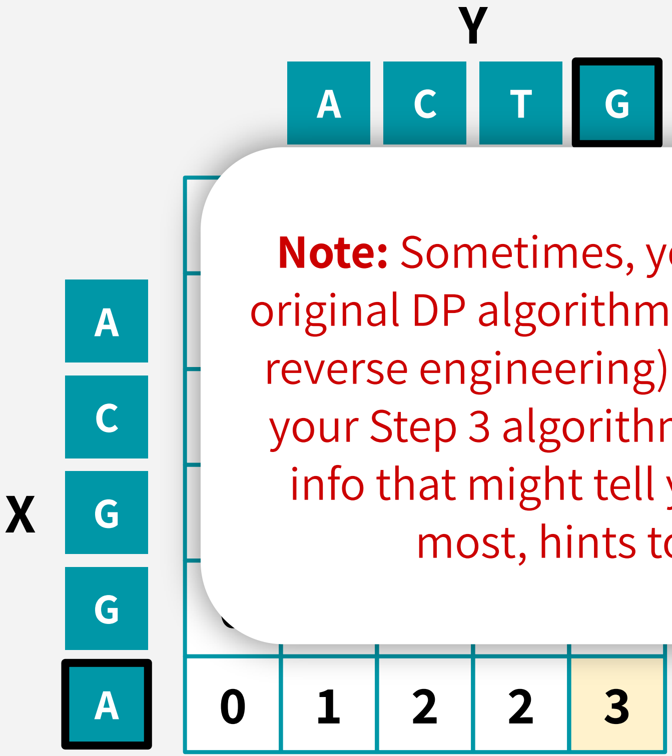
Y

		A	C	T	G
		0	0	0	0	0
X	A	0	1	1	1	1
	C	0	1	2	2	2
	G	0	1	2	2	3
	G	0	1	2	2	3
	A	0	1	2	2	3

```
RECOVER_LCS(X, Y, C):
    // C is already filled out
    L = []
    i = m
    j = n
    while i > 0 and j > 0:
        if X[i] = X[j]:
            append X[i] to the beginning of L
            i = i-1
            j = j-1
        else if C[i,j] = C[i,j-1]:
            j = j-1
        else:
            i = i-1
    return L
```

This extra subroutine takes O(m+n) time!

STEP 4: FIND ACTUAL LCS




```
RECOVER_LCS(X, Y, C):
    // C is already filled out
    f L
        i = i-1
    return L
```

…outine takes O(m+n) time!

Note: Sometimes, you don’t need to track more info in your original DP algorithm from Step 3 (all we did here was do some reverse engineering). Other times, you may want to augment your Step 3 algorithm to keep track of extra information (e.g. info that might tell you which subproblem contributed the most, hints to help you reverse engineer, etc.)

STEP 4: FIND ACTUAL LCS

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

LCS of X and Y:

STEP 4: FIND ACTUAL LCS

X \ Y		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

X[5] ≠ Y[4]

We don't add anything to our LCS.

But we can go up → C[4,4]

LCS of X and Y:

STEP 4: FIND ACTUAL LCS

	Y	A	C	T	G
X		0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

X[4] = Y[4]

We can add “G” to our LCS

We go diagonally back → C[3,3]

LCS of X and Y: G

STEP 4: FIND ACTUAL LCS

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

X[3] ≠ Y[3]

We don’t add anything to our LCS.

But we can go up → C[2,3]

(Going left is okay too since it’s a tie. How you choose to break ties might result in different LCS’s when there are multiple. In this example, there’s actually only one LCS so we we’ll end up with the same LCS either way)

LCS of X and Y: G

STEP 4: FIND ACTUAL LCS

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

$X[2] \neq Y[3]$

We don’t add anything to our LCS.
But we can go left → C[2,2]

LCS of X and Y: | | | G |

STEP 4: FIND ACTUAL LCS

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

$X[2] = Y[2]$

We can add “C” to our LCS.

We go diagonally back → C[1,1]

LCS of X and Y: | | C | G |

STEP 4: FIND ACTUAL LCS

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

X[1] = Y[1]

We can add “A” to our LCS.

We’re done!

LCS of X and Y: A C G

سوال؟

LCS: RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc*. Go back and modify your algorithm in step 3 to make this happen.
5. **Can we do better?** Any wasted space? Other things to optimize?
   (We won't focus on this step too much in lecture/assignments/exams, but in practice, this is definitely a very important step to always consider)!

STEP 5: CAN WE DO BETTER?

STEP 5: CAN WE DO BETTER?

- If we only care about the length of the LCS, then we don’t need to store the entire table (we can just store 2 rows at a time).

STEP 5: CAN WE DO BETTER?

- If we only care about the length of the LCS, then we don't need to store the entire table (we can just store 2 rows at a time).
- If we want to recover the entire LCS, we do need to keep the whole table.

STEP 5: CAN WE DO BETTER?

- If we only care about the length of the LCS, then we don't need to store the entire table (we can just store 2 rows at a time).
- If we want to recover the entire LCS, we do need to keep the whole table.
- Can we improve the runtime of O(mn)?

STEP 5: CAN WE DO BETTER?

- If we only care about the length of the LCS, then we don't need to store the entire table (we can just store 2 rows at a time).
- If we want to recover the entire LCS, we do need to keep the whole table.
- Can we improve the runtime of O(mn)?
  - If you have a bounded alphabet size, you can reduce the running time of the DP algorithm by a logarithmic factor (using the Method of Four Russians).

STEP 5: CAN WE DO BETTER?

- If we only care about the length of the LCS, then we don’t need to store the entire table (we can just store 2 rows at a time).
- If we want to recover the entire LCS, we do need to keep the whole table.
- Can we improve the runtime of O(mn)?
    - If you have a bounded alphabet size, you can reduce the running time of the DP algorithm by a logarithmic factor (using the Method of Four Russians).
    - The general LCS problem is *NP-hard*, so performing much better is an open problem!

سوال؟