# Dynamic Programming

Two hallmark :

- Solution substructure : a solution to a problem contains solutions to subproblems.
- Overlapping subproblems : a recursive solution contains a "small" number of distinct subproblems possibly repeated many times.

add notes

# Application: DNA Sequence Alignment

- DNA sequences can be viewed as strings of **A**, **C**, **G**, and **T** characters, which represent nucleotides.
- Finding the similarities between two DNA sequences is an important computation performed in bioinformatics.
  - For instance, when comparing the DNA of different organisms, such alignments can highlight the locations where those organisms have identical DNA patterns.

# Application: DNA Sequence Alignment

- Finding the best alignment between two DNA strings involves minimizing the number of changes to convert one string to the other.

```
X: ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
     | ||   || || | |||||||||||
     G TC   GT CG G AAGCCGGCCGAA
    GTCGT CGGAA GCCG  GC C G AA
    ||||| ||||| ||||  || | | ||
Y:  GTCGTTCGGAATGCCGTTGCTCTGTAA
```

**Figure 12.1:** Two DNA sequences, X and Y, and their alignment in terms of a longest subsequence, GTCGTCGGAAGCCGGCCGAA, that is common to these two strings.

- A brute-force search would take exponential time, but we can do much better using **dynamic programming**.

# Naïve Algorithm

- For every subsequence of $X$, check whether it's a subsequence of $Y$.
- Time: $\Theta(n2^m)$.
  - $2^m$ subsequences of $X$ to check.
  - Each subsequence takes $\Theta(n)$ time to check: scan $Y$ for first letter, for second, and so on.

## Longest-Common-Subsequence Problem: DP Solution

- Let *F[i,j]* denote the length of the longest common subsequence of the first *i* elements of *A* and the first *j* elements of *B*. The objective of the LCS problem is to find *F[n,m]*.
- We can write:

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

# Recursive Solution

- Define $c[i, j]$ = length of LCS of $X_i$ and $Y_j$ .
- We want $c[m,n]$.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

This gives a recursive algorithm and solves the problem.
But does it solve it well?

# Recursive Solution

$$c[\alpha,\beta]=\begin{cases}0 & \text{if } \alpha \text{ empty or } \beta \text{ empty},\\ c[prefix\alpha, prefix\beta]+1 & \text{if } end(\alpha)=end(\beta),\\ \max(c[prefix\alpha,\beta], c[\alpha, prefix\beta]) & \text{if } end(\alpha)\neq end(\beta).\end{cases}$$

*c*[springtime, printing]

*c*[springtim, printing] *c*[springtime, printin]

[springti, printing] [springtim, printin] [springtim, printin] [springtime, printi]

[springt, printing] [springti, printin] [springtim, printi] [springtime, print]

# Computing the length of an LCS: DP solution

```
LCS-LENGTH (X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then c[i, j] ← c[i−1, j−1] + 1
                 b[i, j] ← "↖"
            else if c[i−1, j] ≥ c[i, j−1]
                then c[i, j] ← c[i−1, j]
                     b[i, j] ← "↑"
                else c[i, j] ← c[i, j−1]
                     b[i, j] ← "←"
return c and b
```

$b[i, j]$ points to table entry whose subproblem we used in solving LCS of $X_i$ and $Y_j$.

$c[m,n]$ contains the length of an LCS of $X$ and $Y$.

Time: $O(mn)$

Drawing
Editing
k to add notes
68%
ENG US
12:10
18-04-2024

# Constructing an LCS

```
PRINT-LCS (b, X, i, j)
if i = 0 or j = 0
    then return
if b[i, j ] = "↖"
    then PRINT-LCS(b, X, i–1, j–1)
            print x_i
    elseif b[i, j ] = "↑"
            then PRINT-LCS(b, X, i–1, j)
else PRINT-LCS(b, X, i, j–1)
```

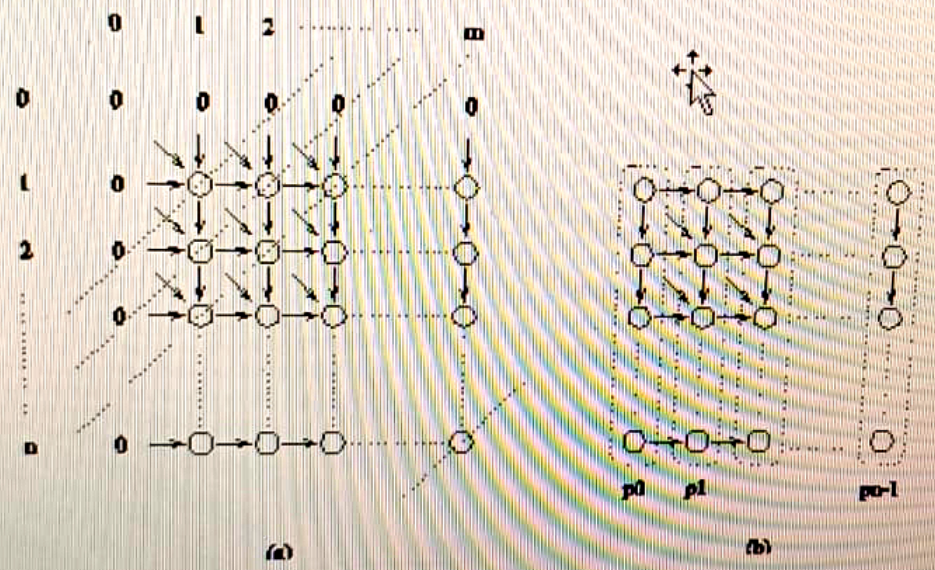
- Initial call is PRINT-LCS (*b*, *X*, *m*, *n*).
- When *b*[*i*, *j* ] = ↖, we have extended LCS by one character. So LCS = entries with ↖ in them.
- Time: $O(m+n)$

## Longest-Common-Subsequence Problem

- The algorithm computes the two-dimensional *F* table in a row- or column-major fashion. The complexity is $\Theta(nm)$.
- Treating nodes along a diagonal as belonging to one level, each node depends on two subproblems at the preceding level and one subproblem two levels prior.
- This DP formulation is nonserial monadic.

Drawing
Select
Editing
o add notes

# Longest-Common-Subsequence Problem




(a) Computing entries of table for the longest-common-subsequence problem. Computation proceeds along the dotted diagonal lines. (b) Mapping elements of the table to processing elements.

# Longest-Common-Subsequence: Example

- Consider the LCS of two amino-acid sequences H E A G A W G H E E and P A W H E A E. For the interested reader, the names of the corresponding amino-acids are A: Alanine, E: Glutamic acid, G: Glycine, H: Histidine, P: Proline, and W: Tryptophan.

| | | H | E | A | G | A | W | G | H | E | E |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| W | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| H | 0 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 3 | 3 |
| E | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 4 | 4 |
| A | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 4 |
| E | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 5 |

- The *F* table for computing the LCS of the sequences. The LCS is A W H E E.

# Visualizing the LCS Algorithm

| L | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 |
| 3 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 |
| 4 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 |
| 5 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 4 | 4 |
| 6 | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 5 | 5 | 5 |
| 7 | 0 | 1 | 1 | 2 | 2 | 3 | 4 | 4 | 4 | 4 | 5 | 5 | 6 |
| 8 | 0 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 5 | 5 | 6 |
| 9 | 0 | 1 | 1 | 2 | 3 | 4 | 4 | 5 | 5 | 5 | 6 | 6 | 6 |

0 1 2 3 4 5 6 7 8 9 10 11

Y=CGATAATTGAGA

X=GTTCCTAATA

0 1 2 3 4 5 6 7 8 9

# Dynamic Programming

Two hallmark :

- Solution substructure : a solution to a problem contains solutions to subproblems.
- Overlapping subproblems : a recursive solution contains a "small" number of distinct subproblems possibly repeated many times.

add notes

# Application: DNA Sequence Alignment

- Finding the best alignment between two DNA strings involves minimizing the number of changes to convert one string to the other.

```
X: ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
     | ||   || || | |||||||||||
     G TC   GT CG G AAGCCGGCCGAA
    GTCGT CGGAA GCCG  GC C G AA
    ||||| ||||| ||||  || | | ||
Y:  GTCGTTCGGAATGCCGTTGCTCTGTAA
```

**Figure 12.1:** Two DNA sequences, X and Y, and their alignment in terms of a longest subsequence, GTCGTCGGAAGCCGGCCGAA, that is common to these two strings.

- A brute-force search would take exponential time, but we can do much better using **dynamic programming**.

# Application: DNA Sequence Alignment

- DNA sequences can be viewed as strings of **A**, **C**, **G**, and **T** characters, which represent nucleotides.
- Finding the similarities between two DNA sequences is an important computation performed in bioinformatics.
  - For instance, when comparing the DNA of different organisms, such alignments can highlight the locations where those organisms have identical DNA patterns.

# Naïve Algorithm

- For every subsequence of $X$, check whether it's a subsequence of $Y$.
- Time: $\Theta(n2^m)$.
  - $2^m$ subsequences of $X$ to check.
  - Each subsequence takes $\Theta(n)$ time to check: scan $Y$ for first letter, for second, and so on.

## Longest-Common-Subsequence Problem: DP Solution

- Let *F[i,j]* denote the length of the longest common subsequence of the first *i* elements of *A* and the first *j* elements of *B*. The objective of the LCS problem is to find *F[n,m]*.
- We can write:

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

# Recursive Solution

- Define $c[i, j]$ = length of LCS of $X_i$ and $Y_j$ .
- We want $c[m,n]$.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

This gives a recursive algorithm and solves the problem.
But does it solve it well?

# Recursive Solution

$$c[\alpha,\beta]=\begin{cases}0 & \text{if } \alpha \text{ empty or } \beta \text{ empty},\\ c[prefix\alpha, prefix\beta]+1 & \text{if } end(\alpha)=end(\beta),\\ \max(c[prefix\alpha,\beta], c[\alpha, prefix\beta]) & \text{if } end(\alpha)\neq end(\beta).\end{cases}$$

*c*[springtime, printing]

*c*[springtim, printing] *c*[springtime, printin]

[springti, printing] [springtim, printin] [springtim, printin] [springtime, printi]

[springt, printing] [springti, printin] [springtim, printi] [springtime, print]

# Computing the length of an LCS: DP solution

```
LCS-LENGTH (X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then c[i, j] ← c[i−1, j−1] + 1
                 b[i, j] ← "↖"
            else if c[i−1, j] ≥ c[i, j−1]
                then c[i, j] ← c[i−1, j]
                     b[i, j] ← "↑"
                else c[i, j] ← c[i, j−1]
                     b[i, j] ← "←"
return c and b
```

$b[i, j]$ points to table entry whose subproblem we used in solving LCS of $X_i$ and $Y_j$.

$c[m,n]$ contains the length of an LCS of $X$ and $Y$.

Time: $O(mn)$

Drawing
Editing
k to add notes
68%
ENG US
12:10
18-04-2024

# Constructing an LCS

```
PRINT-LCS (b, X, i, j)
if i = 0 or j = 0
    then return
if b[i, j ] = "↖"
    then PRINT-LCS(b, X, i–1, j–1)
            print x_i
    elseif b[i, j ] = "↑"
            then PRINT-LCS(b, X, i–1, j)
else PRINT-LCS(b, X, i, j–1)
```

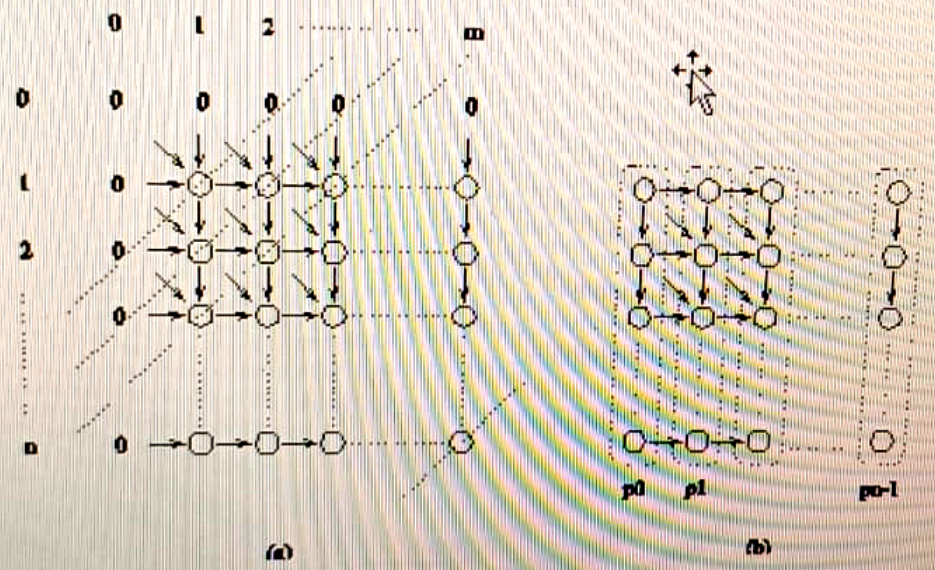
- Initial call is PRINT-LCS (*b*, *X*, *m*, *n*).
- When *b*[*i*, *j* ] = ↖, we have extended LCS by one character. So LCS = entries with ↖ in them.
- Time: $O(m+n)$

## Longest-Common-Subsequence Problem

- The algorithm computes the two-dimensional *F* table in a row- or column-major fashion. The complexity is $\Theta(nm)$.
- Treating nodes along a diagonal as belonging to one level, each node depends on two subproblems at the preceding level and one subproblem two levels prior.
- This DP formulation is nonserial monadic.

Drawing
Select
Editing
o add notes

# Longest-Common-Subsequence Problem




(a) Computing entries of table for the longest-common-subsequence problem. Computation proceeds along the dotted diagonal lines. (b) Mapping elements of the table to processing elements.

# Longest-Common-Subsequence: Example

- Consider the LCS of two amino-acid sequences H E A G A W G H E E and P A W H E A E. For the interested reader, the names of the corresponding amino-acids are A: Alanine, E: Glutamic acid, G: Glycine, H: Histidine, P: Proline, and W: Tryptophan.

| | | H | E | A | G | A | W | G | H | E | E |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| W | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| H | 0 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 3 | 3 |
| E | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 4 | 4 |
| A | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 4 |
| E | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 5 |

- The *F* table for computing the LCS of the sequences. The LCS is A W H E E.

# Visualizing the LCS Algorithm

| L | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 |
| 3 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 |
| 4 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 |
| 5 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 4 | 4 |
| 6 | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 5 | 5 | 5 |
| 7 | 0 | 1 | 1 | 2 | 2 | 3 | 4 | 4 | 4 | 4 | 5 | 5 | 6 |
| 8 | 0 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 5 | 5 | 6 |
| 9 | 0 | 1 | 1 | 2 | 3 | 4 | 4 | 5 | 5 | 5 | 6 | 6 | 6 |

0 1 2 3 4 5 6 7 8 9 10 11

Y=CGATAATTGAGA

X=GTTCCTAATA

0 1 2 3 4 5 6 7 8 9