CS 381 – Spring 2019

Week 7

Dynamic programming (DP)

Break a problem into a series of overlapping subproblems, and use the corresponding recurrence to build up solutions to larger and larger subproblems.

- Overlapping Subproblems

- Optimal Substructure (Optimality Conditions)

  An optimal solution to a problem (instance) contains optimal solutions to subproblems.

DP typically solves optimization problems by combining optimum solutions for subproblems.

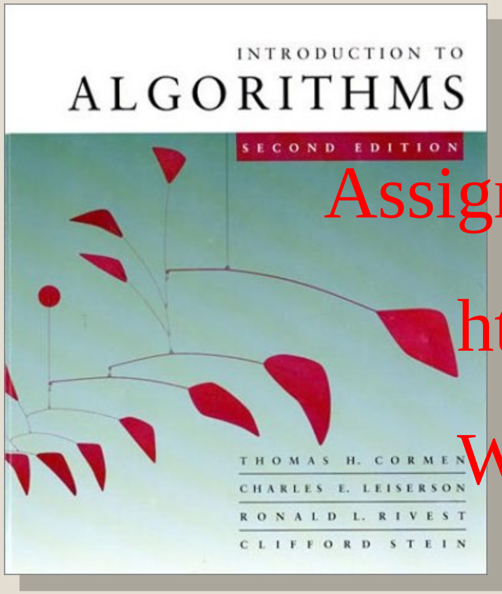
Steps taken when designing a DP algorithm

1. Characterize the structure of an optimal solution
2. **Recursively define the value of an optimal solution in terms of optimum subsolutions**
3. Compute the subsolution entries (never re-compute).
4. Construct an optimal solution from the computed entries and other information.

Dynamic Programming Problems

1) Non-Adjacent Selection
2) Rod Cutting
3) Weighted Selection
4) Longest Common Subsequence

5) Sequence Alignment
6) Coins in a Line
7) 0/1 Knapsack
8) Matrix Chain Multiplication



Dynamic Programming

- Longest Common Subsequence

- Optimal substructure

- Overlapping subproblems
- Sequence alignment (Edit Dist.)

Based on slides by Erik D. Demaine, Charles E. Leiserson and Kevin Wayne

Longest Common Subsequence (LCS)

- Given two sequences $x[1 \,..\, m]$ and $y[1 \,..\, n]$, find a longest subsequence common to them both.

Dynamic programming

Example: ***Longest Common Subsequence (LCS)***

- Given two sequences $x[1 \,.\,.\, m]$ and $y[1 \,.\,.\, n]$, find a longest subsequence common to them both.

"a" *not* "the"

Dynamic programming

Example: ***Longest Common Subsequence (LCS)***

- Given two sequences $x[1 \,.\,.\, m]$ and $y[1 \,.\,.\, n]$, find a longest subsequence common to them both.

"a" *not* "the"

x: A B C B D A B

y: B D C A B A

Dynamic programming

Example: ***Longest Common Subsequence (LCS)***

- Given two sequences $x[1\,.\,.\,m]$ and $y[1\,.\,.\,n]$, find a longest subsequence common to them both.

"a" *not* "the"

x: A B C B D A B

y: B D C A B A

BCBA = LCS(x, y)

functional notation, but not a function

Brute-force LCS algorithm

Check every subsequence of $x[1 \,.\,.\, m]$ to see if it is also a subsequence of $y[1 \,.\,.\, n]$.

Brute-force LCS algorithm

Check every subsequence of $x[1 \,.\,.\, m]$ to see if it is also a subsequence of $y[1 \,.\,.\, n]$.

Analysis

- Checking $= O(n)$ time per subsequence.

- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time $= O(n2^m)$

$= $ exponential time.

Towards a better algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Towards a better algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $|s|$.

Towards a better algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $|s|$.

Strategy: Consider ***prefixes*** of x and y.

- Define $c[i, j] = |\mathrm{LCS}(x[1 \,..\, i], y[1 \,..\, j])|$.
- Then, $c[m, n] = |\mathrm{LCS}(x, y)|$.

Recursive formulation

Theorem.

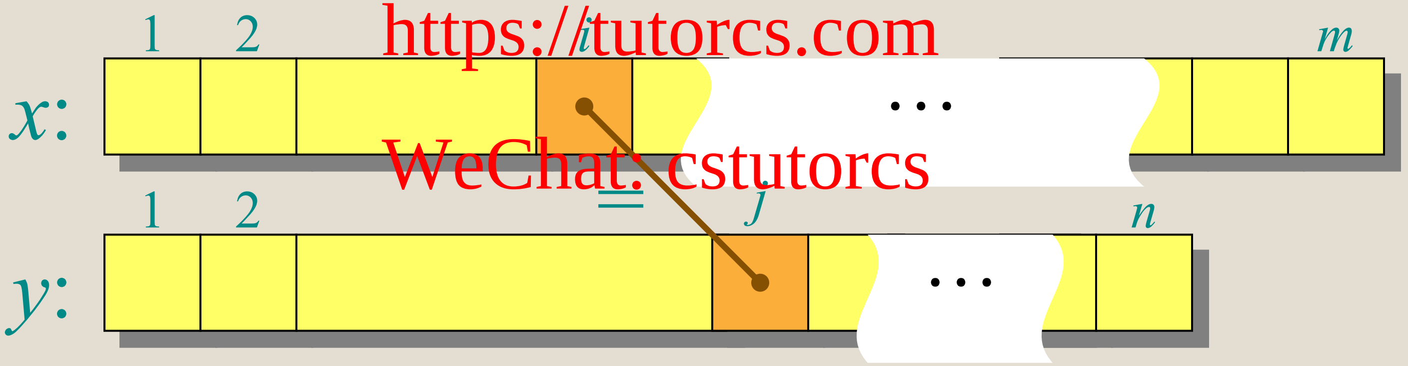
$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

Recursive formulation

Theorem.

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

Proof. Case $x[i] = y[j]$:

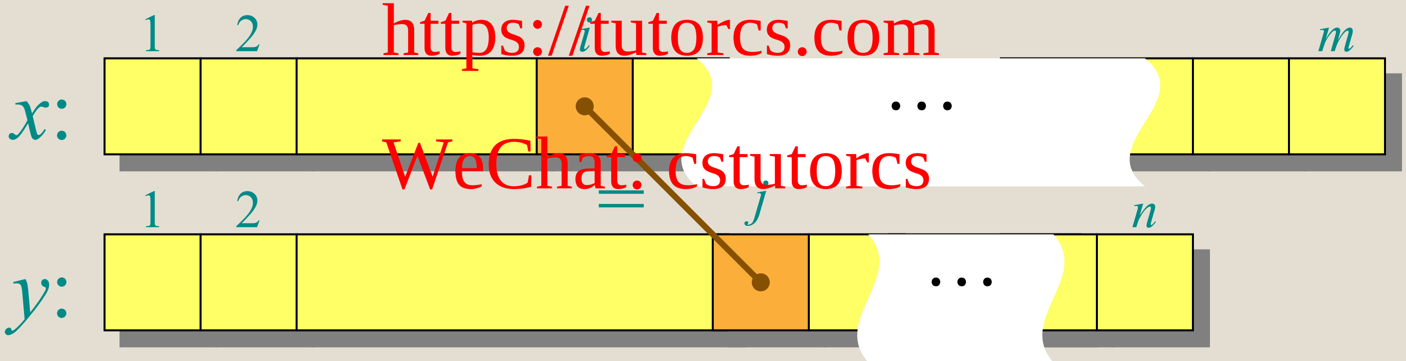


Recursive formulation

Theorem.

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

Proof. Case $x[i] = y[j]$:



Let $z[1 \,..\, k] = \mathrm{LCS}(x[1 \,..\, i], y[1 \,..\, j])$, where $c[i, j] = k$. Then, $z[k] = x[i]$, or else z could be extended. Thus, $z[1 \,..\, k-1]$ is CS of $x[1 \,..\, i-1]$ and $y[1 \,..\, j-1]$.

Proof (continued)

Claim: $z[1 \,..\, k-1] = \mathrm{LCS}(x[1 \,..\, i-1], y[1 \,..\, j-1])$.

Suppose w is a longer CS of $x[1 \,..\, i-1]$ and $y[1 \,..\, j-1]$, that is, $|w| > k-1$. Then, ***cut and paste***: $w \,\|\, z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1 \,..\, i]$ and $y[1 \,..\, j]$ with $|w \,\|\, z[k]| > k$. Contradiction, proving the claim.

Proof (continued)

Claim: $z[1\,..\,k-1] = \mathrm{LCS}(x[1\,..\,i-1], y[1\,..\,j-1])$.

Suppose w is a longer CS of $x[1\,..\,i-1]$ and $y[1\,..\,j-1]$, that is, $|w| > k-1$. Then, ***cut and paste***: $w \,||\, z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1\,..\,i]$ and $y[1\,..\,j]$ with $|w \,||\, z[k]| > k$. Contradiction, proving the claim.

Thus, $c[i-1, j-1] = k-1$, which implies that $c[i, j] = c[i-1, j-1] + 1$.

Other cases are similar. $\square$

Dynamic-programming hallmark #1

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

Dynamic-programming hallmark #1

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If $z = \mathrm{LCS}(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.

Recursive algorithm for LCS

LCS(x, y, i, j)
    if $x[i] = y[j]$
        then *return* LCS(x, y, i–1, j–1) + 1
        else *return* max{LCS(x, y, i–1, j),
                                LCS(x, y, i, j–1)}

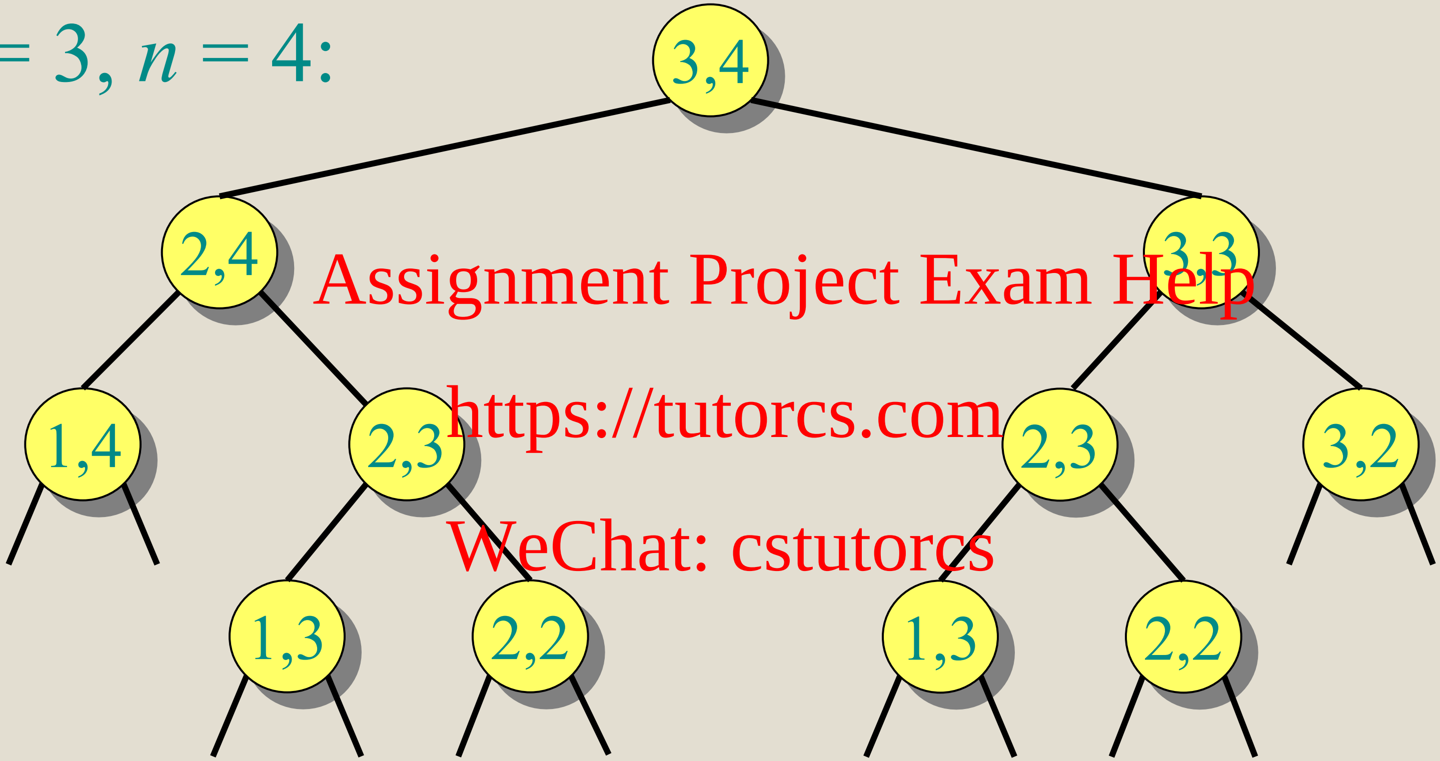
Recursive algorithm for LCS

LCS(x, y, i, j)
  if $x[i] = y[j]$
    then *return* LCS$(x, y, i-1, j-1) + 1$
    else *return* $\max\{$LCS$(x, y, i-1, j)$,
        LCS$(x, y, i, j-1)\}$

Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

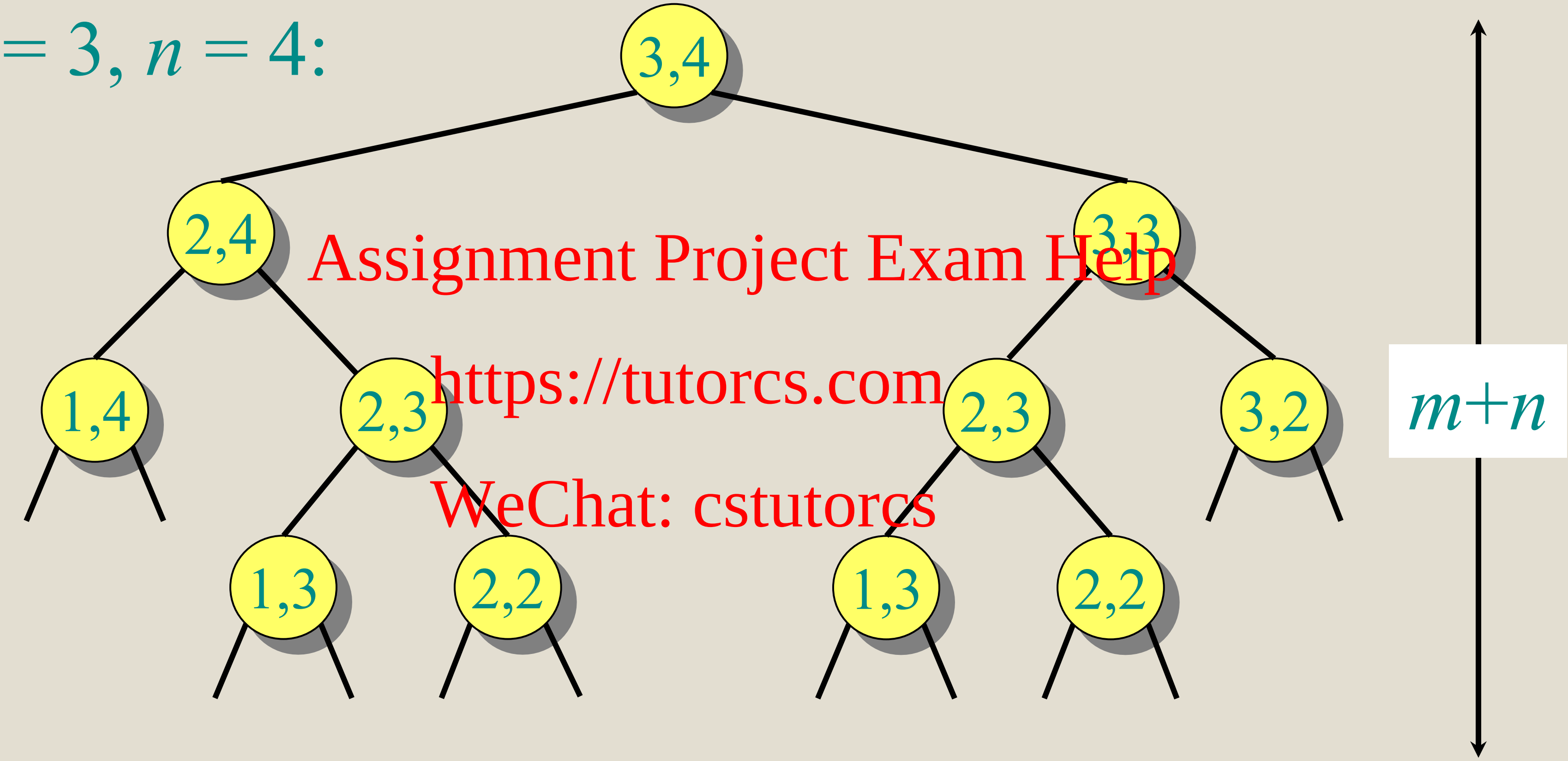
Recursion tree

$m = 3, n = 4$:



Recursion tree

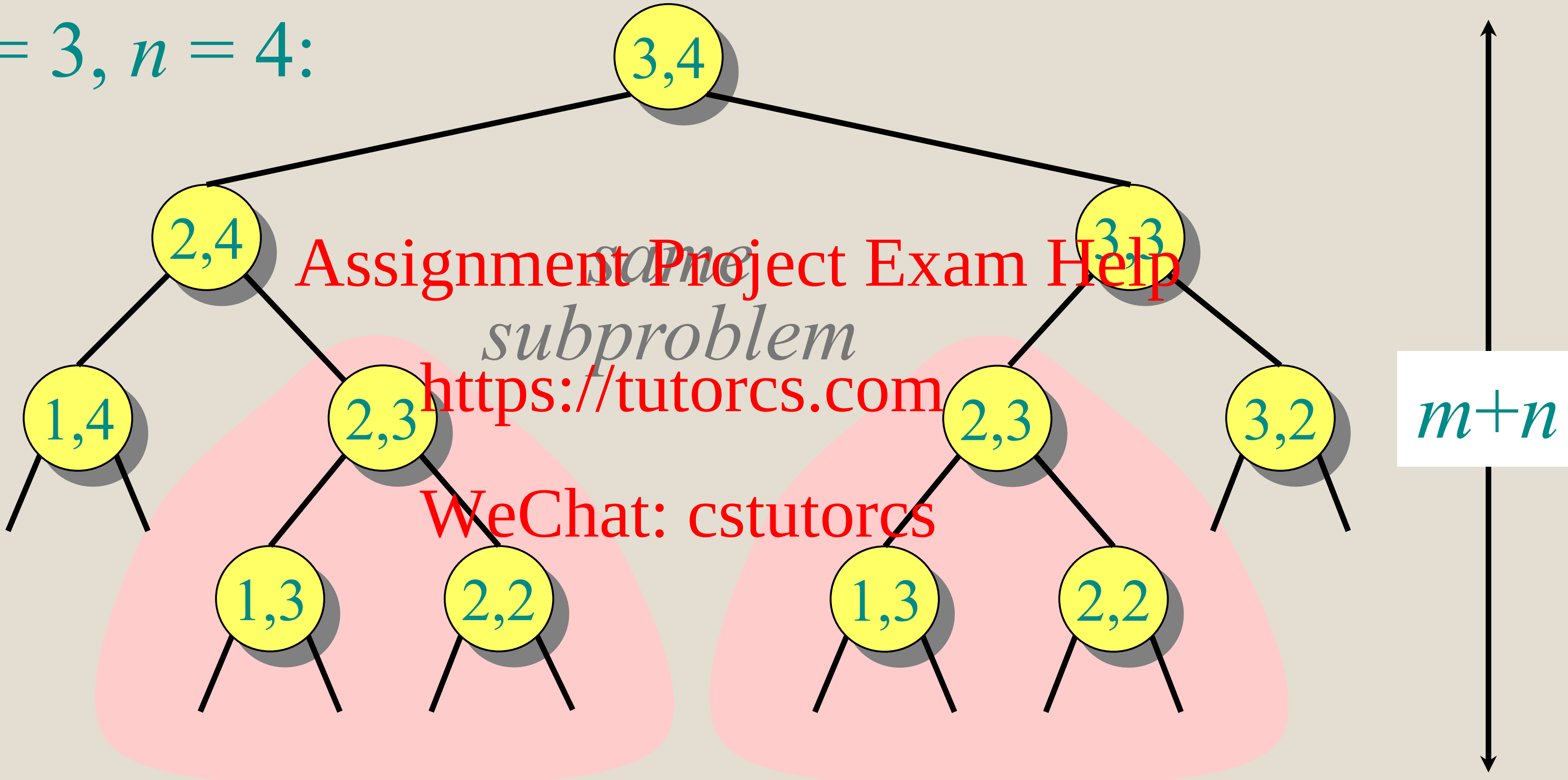
$m = 3, n = 4$:



Height $= m + n \Rightarrow$ potentially work exponential.

Recursion tree

$m = 3,\ n = 4$:



Height $= m + n \Rightarrow$ potentially exponential work, but we're solving subproblems already solved!

Dynamic-programming hallmark #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

Dynamic-programming hallmark #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths m and n is only mn.

Memoization algorithm (Top-Down DP)

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

Memoization algorithm (Top-Down DP)

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

LCS(x, y, i, j)
  if $c[i, j] = \text{NIL}$

    then if $x[i] = y[j]$

      then $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$
      else $c[i, j] \leftarrow \max\{LCS(x, y, i-1, j)$, LCS$(x, y, i, j-1)\}$
  return $c[i, j]$

same as before

Memoization algorithm (Top-Down DP)

LCS(x, y, i, j)
    if $c[i, j] = \text{NIL}$
        then if $x[i] = y[j]$
            then $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$
            else $c[i, j] \leftarrow \max\{\text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1)\}$
    return $c[i, j]$

same as before

Time $= \Theta(mn) =$ constant work per table entry.
Space $= \Theta(mn)$.

Dynamic-programming algorithm: bottom-up

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

LCS(x, y, i, j)
    if $c[i, j]$ = NIL
        if $x[i] = y[j]$
            then $c[i, j] \leftarrow$ LCS($x, y, i–1, j–1$) + 1
        else
            $c[i, j] \leftarrow \max\{$LCS($x, y, i–1, j$), LCS($x, y, i, j–1$)$\}$

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time $= \Theta(mn)$.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time $= \Theta(mn)$.

Reconstruct LCS by tracing backwards.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time $= \Theta(mn)$.

Reconstruct LCS by tracing backwards.

Space $= \Theta(mn)$.

Exercise: $O(\min\{m, n\})$.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Problem 5: Sequence Alignment (Edit Distance)

(Kleinberg-Tardos 6.6 + Wayne slides)

Similarity of short strings is easy for humans:

```
occurrence          occurrence
ocurrence           o-currence
```


How to define similarity?

- Defined in the 1970's by computer scientists (edit distance) and molecular biologists (for DNA sequences)
- Many versions of the problem exist
- Dynamic programming is a common technique

String similarity

Q. How similar are two strings?

Ex. ocurrance and occurrence.

o	c	u	r	r	a	n	c	e	–
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	–	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	–	u	r	r	–	a	n	c	e
o	c	c	u	r	r	e	–	n	c	e

0 mismatches, 3 gaps

Edit distance

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.

C	T	–	G	A	C	C	T	A	C	G
C	T	G	G	A	C	G	A	A	C	G

$\text{cost} = \delta + \alpha_{CG} + \alpha_{TA}$

Applications. Unix diff, speech recognition, computational biology, ...

What makes the problem challenging?

- When there is a mismatch, is it better to pay for the mismatch or better to introduce a gap (i.e., leave one character unmatched)?
  - The decision impacts what happens afterwards
  - Pursuing both options can lead to exponential time

<u>Use dynamic programming</u>

- What subsolution is an optimum solution composed of?
- Does the principle of optimality hold?
- Can subsolutions be computed in a systematic and efficient way?

G C A T G C U

G A T T A C A

G C A T G – C U

G - A T T A C A

2 gaps, 2 mismatches

G C A - T G C U

G - A T T A C A

G C A T – G C U

G – A T T A C A

Sequence Alignment

$X = \text{CAGCACTTGGATTCTCCATGG} \qquad |X| = m$

$Y = \text{AGGACTG ATCCTCG} \qquad |Y| = n$

Assume mismatches and gaps have a cost of 1 each

$$
OPT(m,n) = \min \begin{cases} OPT(m-1,n-1\} & \text{if } x_m = y_n \\ OPT(m-1,n-1\} + 1 & \text{if } x_m \neq y_n \\ OPT(m-1,n\} + 1 & x_m \text{ has a gap} \\ OPT(m,n-1\} + 1 & y_n \text{ has a gap} \end{cases}
$$

$x_1\ x_2\ \ldots\ x_i = \text{CAGCACTTGGATTCTCCATGG}$

$y_1\ y_2\ \ldots\ y_j = \text{AGGACTGATCCTCG}$

$$\mathrm{OPT}(i,j) = \min \begin{cases} \mathrm{OPT}(i\text{-}1,j\text{-}1\} & \text{if } x_i = y_j \\ \mathrm{OPT}(i\text{-}1,j\text{-}1\} + 1 & \text{if } x_i \neq y_j \\ \mathrm{OPT}(i\text{-}1,j\} + 1 & x_i \text{ has a gap} \\ \mathrm{OPT}(i,j\text{-}1\} + 1 & y_j \text{ has a gap} \end{cases}$$

Compute the optimal alignment of a string of length m and a string of length n in **O(nm) time using O(nm) space.**

Sequence alignment/ Edit distance

Given: strings $\mathbf{X = x_1 x_2 \ldots . x_m}$ and $\mathbf{Y = y_1 y_2 \ldots . y_n}$

Find a minimum cost alignment.

Simplfied cost structure

- Match: 0
- Mismatch: 1
- Gap: 1

X = CAGCACTTGGATTCTCCATGG

Y = AGGACTG ATCCTCG

G C A T G C U T

C A T G C A T

Gap on upper T

G C A T G C U

C A T G C A T

Gap on lower T

G C A T G C U T

C A T G C A

Match T and T

G C A T G C U

C A T G C A

Does the principle of optimality hold?

Sequence Alignment

$X = \text{CAGCACTTGGATTCTCCATGG} \qquad |X| = m$

$Y = \text{AGGACTG ATCCTCG} \qquad |Y| = n$

Assume mismatches and gaps have a cost of 1 each

$$
OPT(m,n) = \min \begin{cases} OPT(m-1,n-1\} & \text{if } x_m = y_n \\ OPT(m-1,n-1\} + 1 & \text{if } x_m \neq y_n \\ OPT(m-1,n\} + 1 & x_m \text{ has a gap} \\ OPT(m,n-1\} + 1 & y_n \text{ has a gap} \end{cases}
$$

$x_1\, x_2 \ldots x_i =$ CAGCACTTGGATTCTCCATGG

$y_1\, y_2 \ldots\; y_j \; =$ AGGACTGATCCTCG

$$
\mathrm{OPT}(i,j) = \min \begin{cases} \mathrm{OPT}(i\text{-}1,j\text{-}1\} & \text{if } x_i = y_j \\ \mathrm{OPT}(i\text{-}1,j\text{-}1\} + 1 & \text{if } x_i \neq y_j \\ \mathrm{OPT}(i\text{-}1,j\} + 1 & x_i \text{ has a gap} \\ \mathrm{OPT}(i,j\text{-}1\} + 1 & y_j \text{ has a gap} \end{cases}
$$

Compute the optimal alignment of a string of length m and a string of length n in **O(nm) time using O(nm) space.**

Sequence alignment

Goal. Given two strings $x_1\, x_2 \ldots x_m$ and $y_1\, y_2 \ldots y_n$ find min cost alignment.

Def. An alignment M is a set of ordered pairs $x_i - y_j$ such that each item occurs in at most one pair and no crossings.

$x_i - y_j$ and $x_{i'} - y_{j'}$ cross if $i < i'$, but $j > j'$

Def. The cost of an alignment M is:

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i\,:\, x_i \text{ unmatched}} \delta + \sum_{j\,:\, y_j \text{ unmatched}} \delta}_{\text{gap}}$$

x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	–	G
–	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5	y_6

an alignment of CTACCG and TACATG:

$$M = \{\, x_2 – y_1,\ x_3 – y_2,\ x_4 – y_3,\ x_5 – y_4,\ x_6 – y_6 \,\}$$

Sequence alignment: problem structure

Def. $OPT(i, j)$ = min cost of aligning prefix strings $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_j$.

Case 1. OPT matches $x_i - y_j$.
Pay mismatch for $x_i - y_j$ + min cost of aligning $x_1\, x_2 \ldots x_{i-1}$ and $y_1\, y_2 \ldots y_{j-1}$.

Case 2a. OPT leaves x_i unmatched.
Pay gap for x_i + min cost of aligning $x_1\, x_2 \ldots x_{i-1}$ and $y_1\, y_2 \ldots y_j$.

Case 2b. OPT leaves y_j unmatched.
Pay gap for y_j + min cost of aligning $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_{j-1}$.

optimal substructure property
(proof via exchange argument)

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Sequence alignment: algorithm

SEQUENCE-ALIGNMENT $(m, n, x_1, \ldots, x_m, y_1, \ldots, y_n, \delta, \alpha)$

FOR $i = 0$ TO m

  $M[i, 0] \leftarrow i\,\delta.$

FOR $j = 0$ TO n

  $M[0, j] \leftarrow j\,\delta.$

FOR $i = 1$ TO m

  FOR $j = 1$ TO n

$$M[i, j] \leftarrow \min \{\, \alpha[x_i, y_j] + M[i-1, j-1], \quad \delta + M[i-1, j], \quad \delta + M[i, j-1]\}.$$

RETURN $M[m, n]$.

Use a matrix S to record what action gives M-entry

Analysis of the sequence alignment algorithm

- Computes the optimal alignment of a string of length n and a string of length m in O(nm) time using O(nm) space
- Approach can be applied to other edit distance problems
- The solution is generated by recording for every entry the decision creating the minimum and tracing is back
- Can we reduce the space?
  - If we only need the optimal value, we need only O(n+m) space; time remains O(nm)
  - A more complex algorithm achieves O(nm) time and O(n+m) space and generates the solution

```
CAGCACTTGGATTCTCGG
CAGC-----G-T----GG
```

10 gaps

```
CAGCA-CTTGGATTCTCGG
---CAGCGTGG--------
```


11 gaps

1 mismatch

Variations

- Gaps at the end should not be penalized
- Not all gaps in a continuous sequence of gaps are equal
- Gaps in the longer input string costs more