Algorithm Design and Analysis

CSE 565

LECTURE 16

Dynamic Programming

- Least Common Subsequence
- Saving space

Adam Smith

Least Common Subsequence

A.k.a. “sequence alignment”

“edit distance”

…

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)

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

BCAB
~~BCBA~~ = $\mathrm{LCS}(x, y)$

Motivation

- Approximate string matching [Levenshtein, 1966]
  - Search for “occurance”, get results for “occurrence”
- Computational biology [Needleman-Wunsch, 1970’s]
  - Simple measure of genome similarity

    `cgtacgtacgtacgtacgtacgtatcgtacgt`
    `acgtacgtacgtacgtacgtacgtacgtacgt`

Motivation

- Approximate string matching [Levenshtein, 1966]
  - Search for "occurance", get results for "occurrence"
- Computational biology [Needleman-Wunsch, 1970's]
  - Simple measure of genome similarity

```
 acgtacgtacgtacgtcgtacgtatcgtacgt
aacgtacgtacgtacgtcgtacgta cgtacgt
```

- n – length(LCS(x,y)) is called the "edit distance"

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.

Dynamic programming 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.

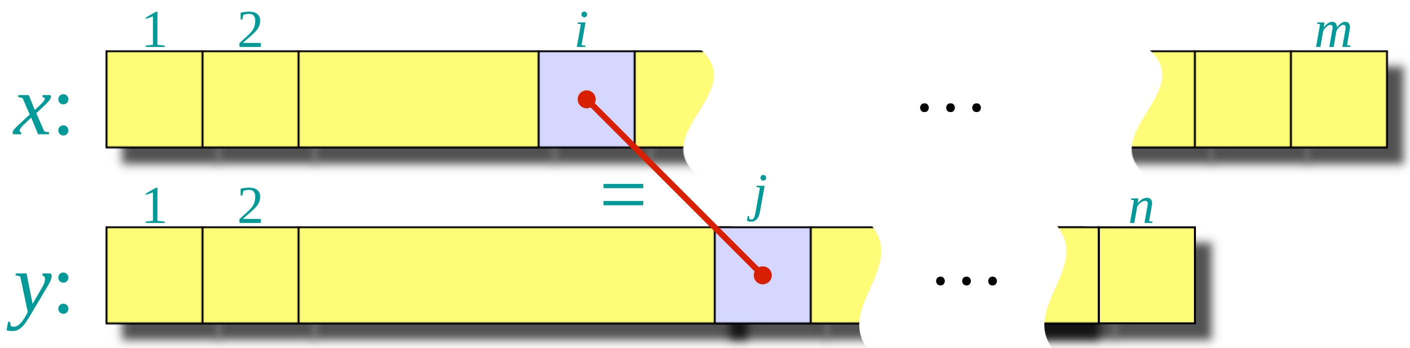
- Define $c[i, j] = |\text{LCS}(x[1 \,.\,.\, i], y[1 \,.\,.\, j])|$.
- Then, $c[m, n] = |\text{LCS}(x, y)|$.

Recursive formulation

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

Base case: $c[i,j]=0$ if $i=0$ or $j=0$.

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

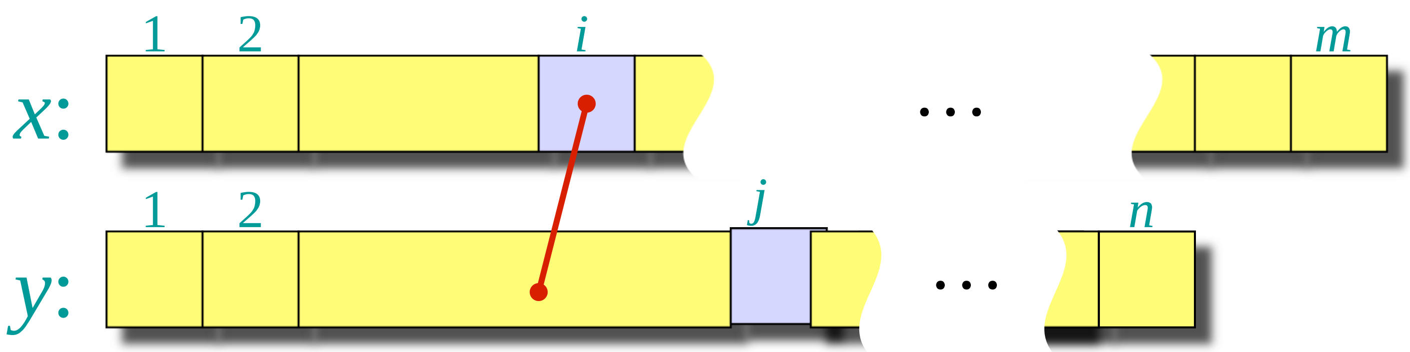


The second case is similar.

Recursive formulation

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

Case $x[i] \neq y[j]$: best matching might use $x[i]$ or $y[j]$ (or neither) but not both.



Dynamic-programming hallmark #1

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

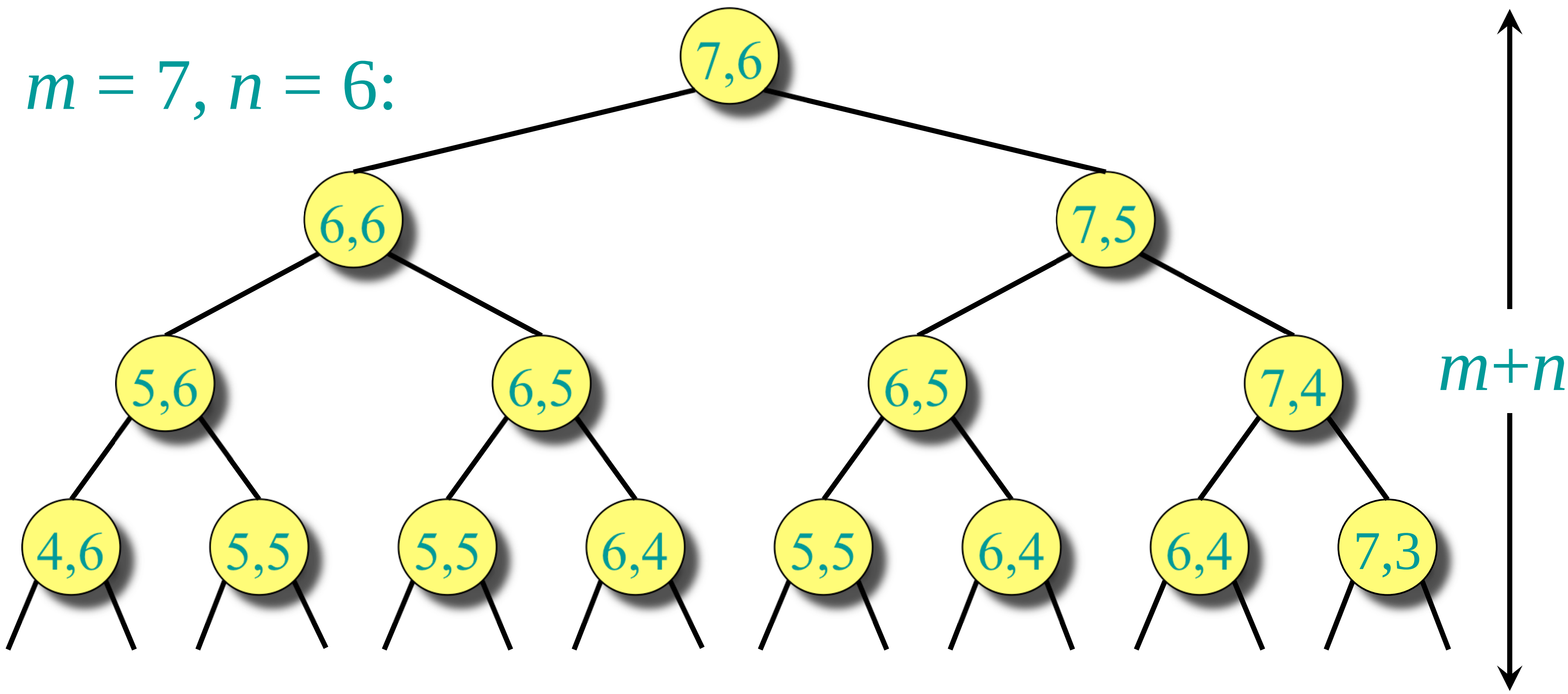
If $z = \mathrm{LCS}(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.

Recursive algorithm for LCS

LCS(x, y, i, j) // ignoring base cases
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 $c[i, j]$

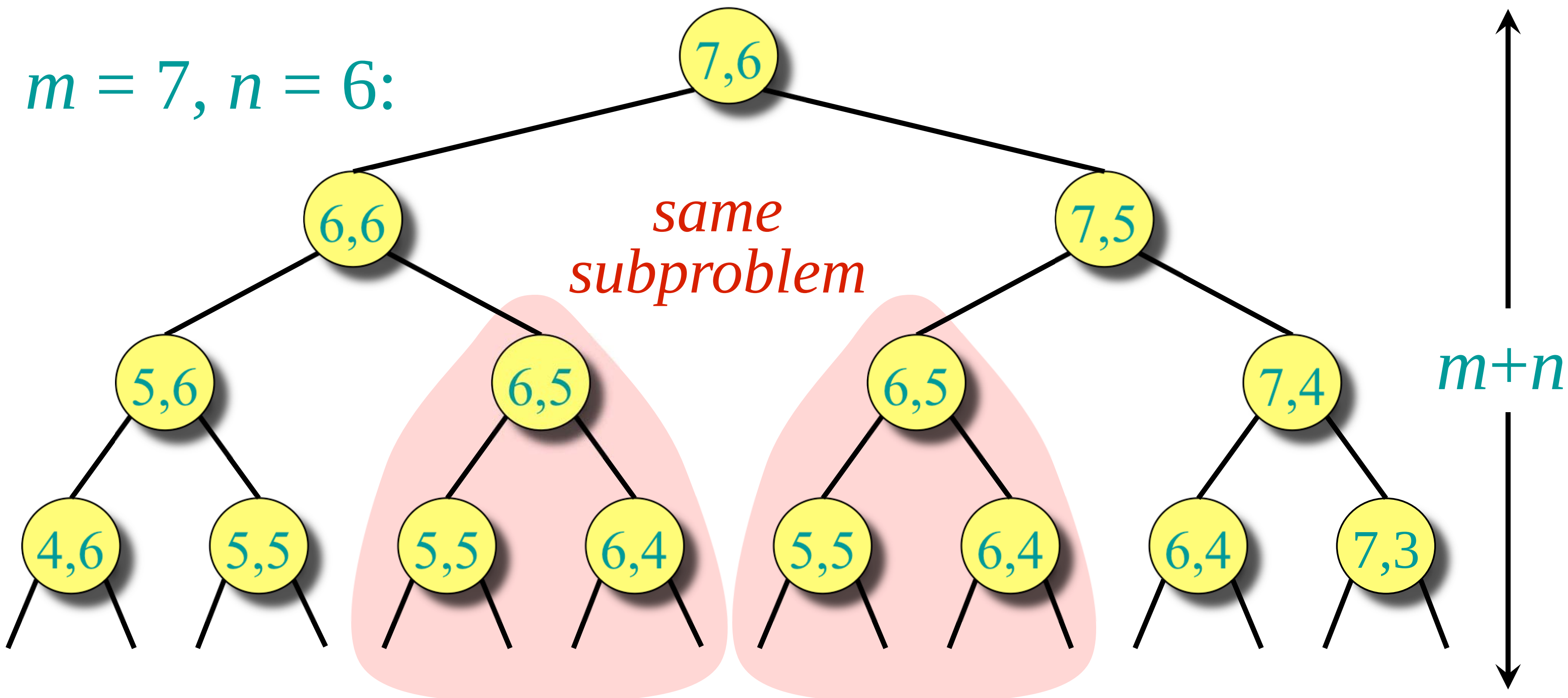
Worse case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion tree



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

Recursion tree



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

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

Memoization algorithm

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$(x, y, i, j–1)\}$

same as before

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

Dynamic-programming algorithm

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

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.

Multiple solutions are possible.

		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 = $\Theta(mn)$.

Section 6.7: $O(\min\{m, n\})$

		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

Saving space

- Why is space important?
  - Cost of storage
  - Cache misses
- Suppose we only want to compute the **cost** of the optimal solution
  - Fill the table left to right
  - Remember only the previous column
  - O(m) space in addition to input

		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

Computing the optimum

- View the matrix as a directed acyclic graph: our goal is to find a longest path from (0,0) to (m,n)
  - Every path must pass through column n/2
- Algorithm
  1. Find longest path lengths from (0,0) to all points in column n/2, i.e. (0,n/2),(1,n/2),…,(m,n/2)
  2. Find longest path lengths from all of column n/2 to (m,n)
  3. Find a point q where optimal path crosses column n/2. Add (q,n/2) to output list
  4. **Recursively** solve subproblems (0,0) -> (q,n/2) and (q,n/2) to (m,n)
- How much time and space?