CENG 218
Design and Analysis of Algorithms

Izmir Institute of Technology

Lecture 11: Dynamic programming

Slides were mostly prepared using the material provided by Prof. Charles E. Leiserson and Prof. Erik Demaine from MIT

Dynamic programming (DP)

It is a design technique, like divide-and-conquer.

Contrast to divide-and-conquer, DP applies when the subproblems overlap.

When subproblems overlap, divide-and-conquer approach does unnecessary work by repeatedly solving common subproblems.

DP approach, however, solves each subproblem just once and saves the answer in a table.

Example 1: *LCS*

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 = $LCS(x, y)$

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 is $O(n)$ time **per subsequence** since we are searching it in y.
- 2^m subsequences of x
  - each bit-vector of length m is a distinct subsequence of x
  - Or, enumerate all elements in x and count subsets
- Worst-case running time is $O(n2^m)$ which is exponential time.

Towards a better algorithm

Simplification:

1. First, find the *length* of a LCS.
2. Extend the algorithm to find the LCS itself.

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

Strategy: Consider ***prefixes*** of x and y and find the LCS(x, y) in terms of those.

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

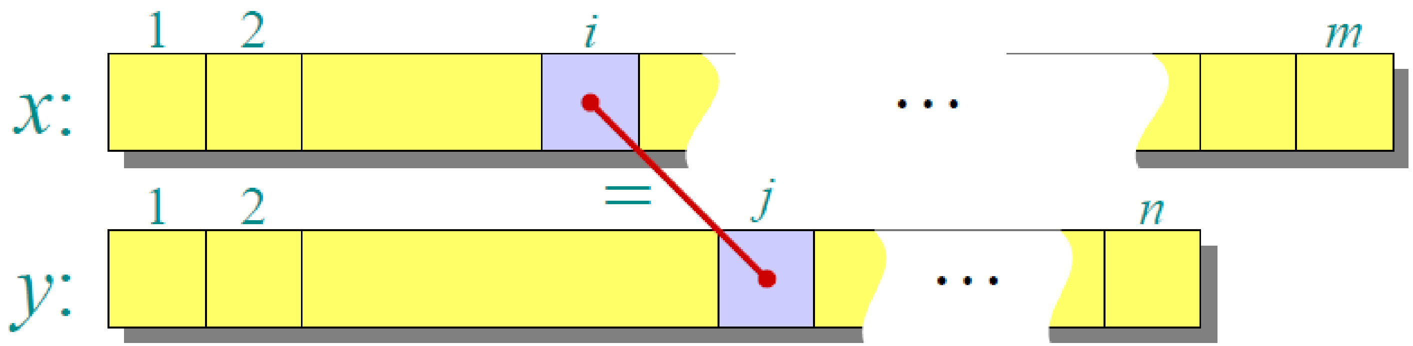
Recursive formulation

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

Recursive formulation

Proof.

Case 1: $x[i] = y[j]$



Let $z[1..k] = \mathrm{LCS}(x[1..i], y[1..j])$, where $c[i, j]=k$.
Then, $z[k] = x[i]$ $(= y[j])$.
Or, maybe $x[i]$ is in it and $y[j]$ is not (or vice versa).
But, in this case we might as well match $x[i]$ with $y[j]$.
Thus, $z[1..k-1]$ is CS of $x[1..i-1]$ and $y[1..j-1]$.

Proof (continued)

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

Proof by contradiction: Suppose w is a longer CS of $x[1\,.\,.\,i-1]$ and $y[1\,.\,.\,j-1]$, that is, $|w| > k-1$. Then, $w \,||\, z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1\,.\,.\,i]$ and $y[1\,.\,.\,j]$ with $|w \,||\, z[k]| > k$. Contradiction!

Thus, $c[i-1, j-1]=k-1$, implying $c[i, j]=c[i-1, j-1]+1$.

Proof (continued)

Case 2: $x[i] \mathrel{!=} y[j]$

Then, the LCS of $x[1 \,.\,.\, i]$ and $y[1 \,.\,.\, j]$ cannot contain both $x[i]$ and $y[j]$. The answer ignores either $x[i]$ or $y[j]$ or both.

Thus, $c[i, j] = \max\{\, c[i\text{-}1, j], c[i, j\text{-}1] \,\}$.

Dynamic programming hallmark #1

Optimal substructure

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

Hallmark here means a requirement to apply DP.

If z = LCS(x, y), then any prefix of z is an LCS of a prefix of x and a prefix of y.

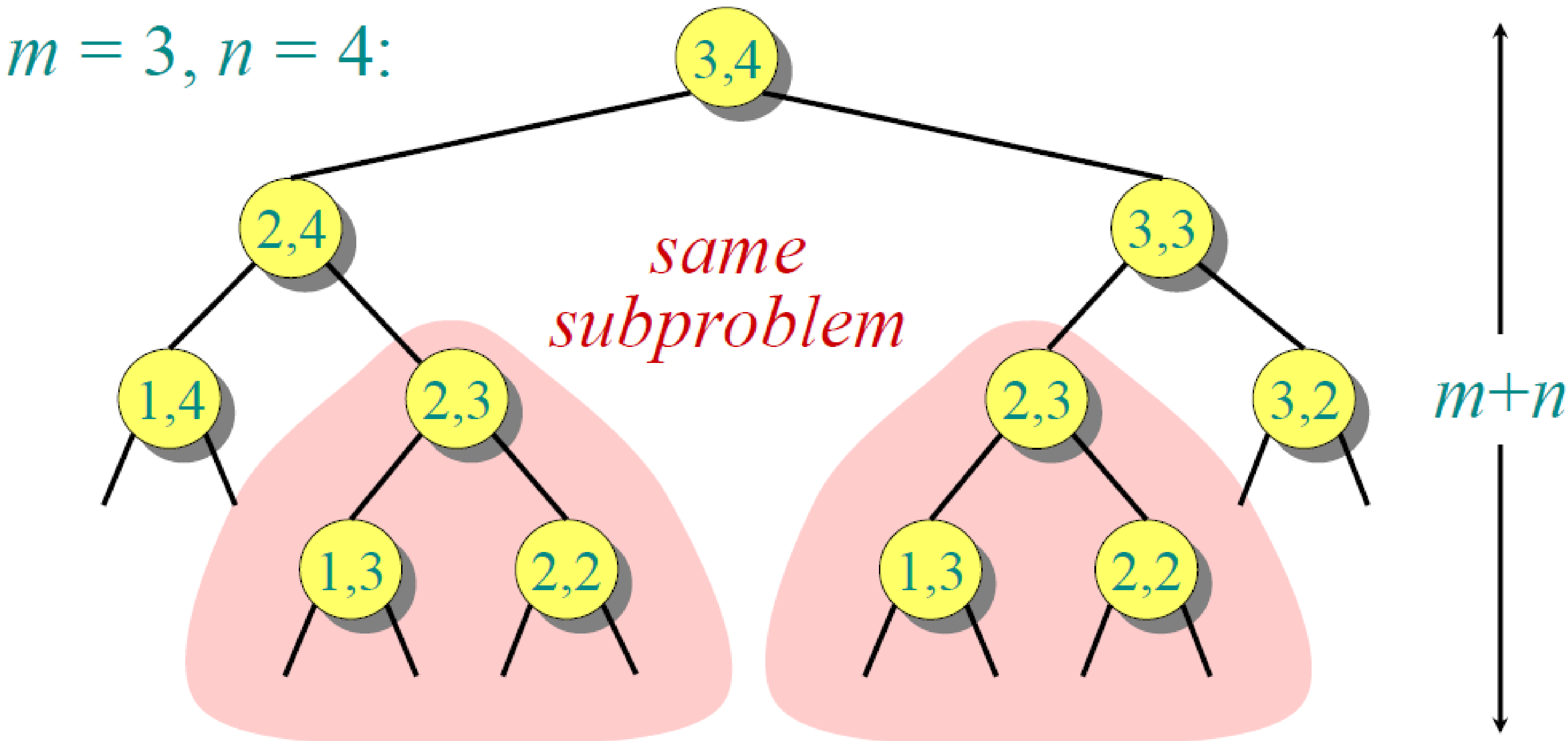
optimal substructure is satisfied

Recursive algorithm for LCS

LCS(x, y, i, j)
  if $x[i] = y[j]$
    $c[i, j] \leftarrow$ 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 $c[i, j]$

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

Recursion tree



Height = $m + n$ => $T(n)=O(2^{m+n})$,
and 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.

How many distinct LCS subproblems are there for two strings of lengths m and n?

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.

Let $c[1..m,1..n]$ a 2D array initialized with NIL

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

checking the table

Memoization algorithm

Call the following subroutine with LCS(x,y,m,n)

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

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

Bottom-up (non-recursive) dynamic programming algorithm

IDEA:
Compute the table bottom-up.
Time = $\Theta(mn)$.

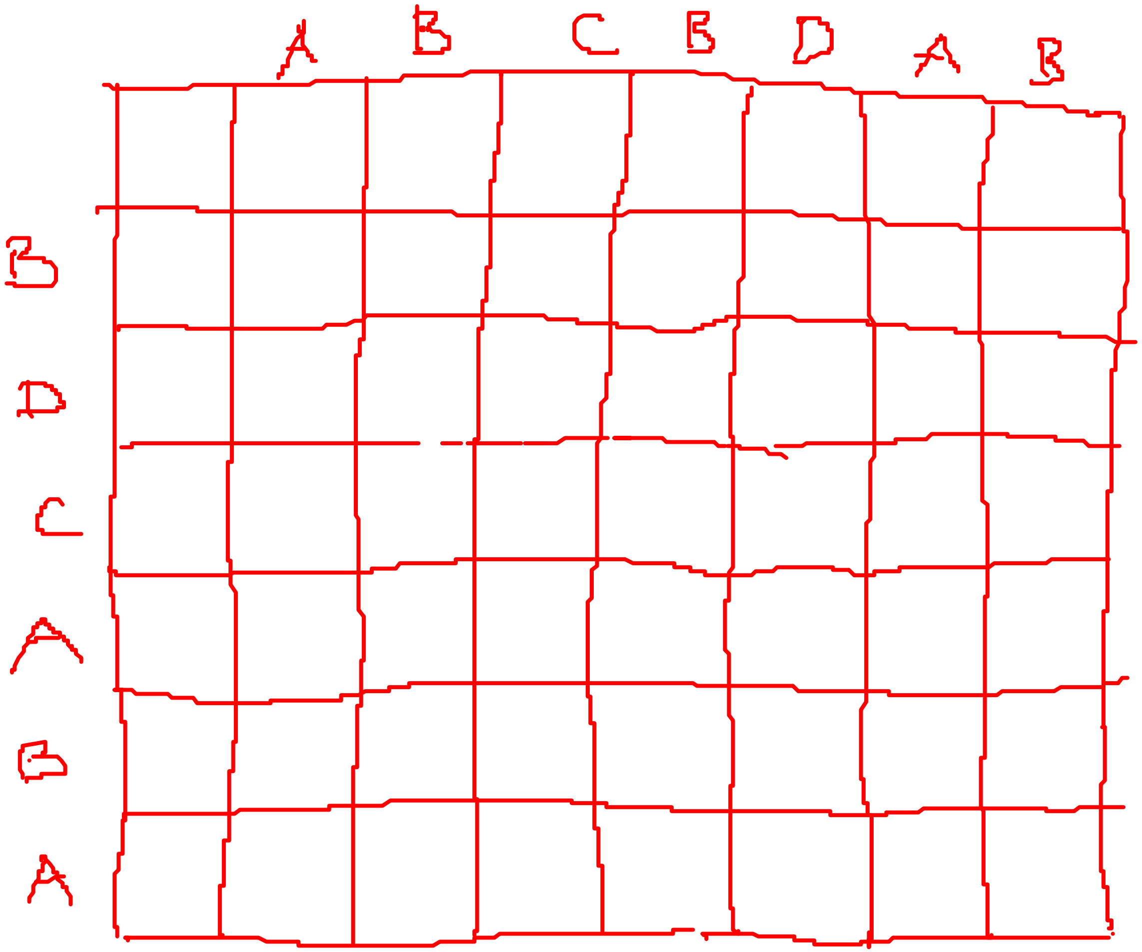
Note: Additional zero vectors at the top and at the left.

		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

Bottom-up algorithm

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            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
```

Bottom-up algorithm



Bottom-up algorithm

IDEA:
Compute the table bottom-up.
Time = $\Theta(mn)$.

Construct an LCS by tracing backwards.

Space = $\Theta(mn)$.

		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

Summary for LCS problem

Computation time for sequences $x[1\,..\,m]$ and $y[1\,..\,n]$:

1) Brute-force algorithm: $O(n2^m)$

2) Recursive algorithm: $O(2^{m+n})$

3) Recursive algorithm
with memoization: $\Theta(mn)$

4) Bottom-up algorithm: $\Theta(mn)$

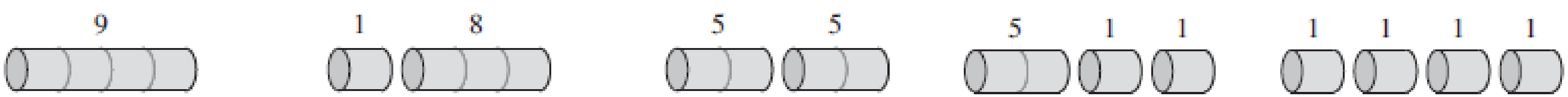
Dynamic programming

Example 2: Rod-cutting

Given a rod of length n inches and a table of prices p_i for i = 1,2, .., n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Consider the case when n = 4. All possible cuts are:



Cutting the rod into two 2-inch pieces produces the maximum revenue: 5+5=10.

Rod-cutting: Optimal substructure

To solve the original problem, we solve smaller problems of the same type. We say, rod-cutting problem exhibits optimal substructure (hallmark #1).

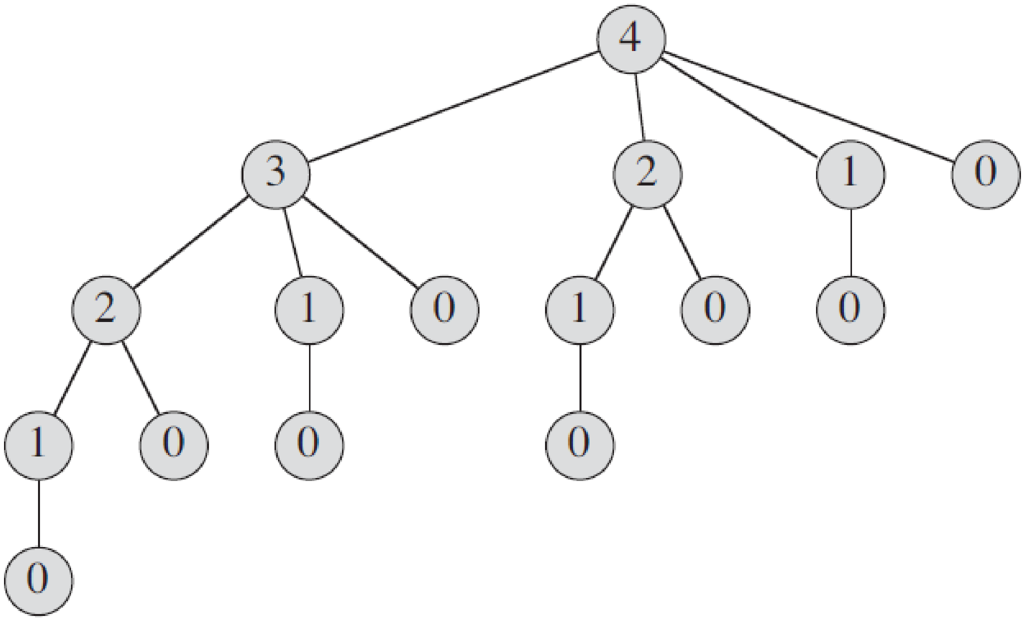
$$
\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \quad \text{(no cuts)} ,\\
r_2 &= 5 && \text{from solution } 2 = 2 \quad \text{(no cuts)} ,\\
r_3 &= 8 && \text{from solution } 3 = 3 \quad \text{(no cuts)} ,\\
r_4 &= 10 && \text{from solution } 4 = 2 + 2 ,\\
r_5 &= 13 && \text{from solution } 5 = 2 + 3 ,\\
r_6 &= 17 && \text{from solution } 6 = 6 \quad \text{(no cuts)} ,\\
r_7 &= 18 && \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3 ,\\
r_8 &= 22 && \text{from solution } 8 = 2 + 6 ,\\
r_9 &= 25 && \text{from solution } 9 = 3 + 6 ,\\
r_{10} &= 30 && \text{from solution } 10 = 10 \quad \text{(no cuts)} .
\end{aligned}
$$

Rod-cutting: Recursive top-down implementation

```
CUT-ROD(p, n)
if n == 0
    return 0
q = -∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n - i))
return q
```

p is the price list, *n* is the problem size.
For *n*=40, it takes about several minutes. Why?

Rod-cutting: Overlapping subproblems (Recursive top-down implementation)



The recursion tree showing recursive calls from a call of CUT-ROD(p,n) when n=4.

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

Rod-cutting: Memoized approach

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

Initializing table

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

If the problem is previously solved for *n*

$$T(n) = \Theta(n^2).$$

Rod-cutting: Bottom-up approach

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

Again $T(n) = \Theta(n^2)$. This implementation is even simpler.

The End

In summary, DP

- defines the value of an optimal solution based on optimal solutions of subproblems.
- computes the solutions in top-down (recursive calls and memoization) or bottom-up fashion.

Textbook Section 15.1 (rod-cutting),
15.2 (matrix-chain multiplication),
15.3 (DP elements) and 15.4 (LCS).