CS 381 – FALL 2019

Week 6.3, Friday, Sept 27

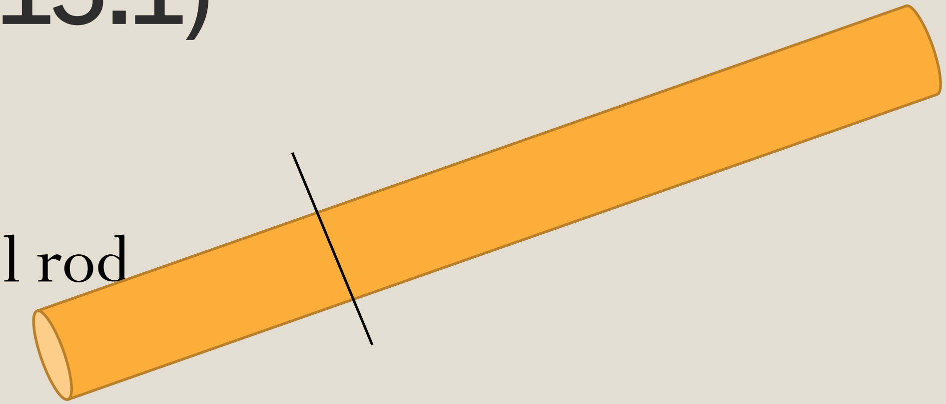
Tentative Date for Final Exam: Thursday, December 12
(7-9PM STEW 130)

Announcements

- No PSOs this week (due to Midterm)
  - PSOs resume next week (as normal)

- Midterm grading in progress…

Rod Cutting Problem (15.1)



- Input is
  - **n**, the length of a steel rod
  - an array **p** of size n

The rod is cut into shorter rods.

- A rod of length k is sold for profit p[k], 1≤k ≤n.

Cut the rod into pieces that maximize the total profit

  - No cuts can be undone
  - Making a cut is “free”

Example

n=5

	1	2	3	4	5
p	**3**	**5**	**10**	**12**	**14**

Making no cut has a profit of 14

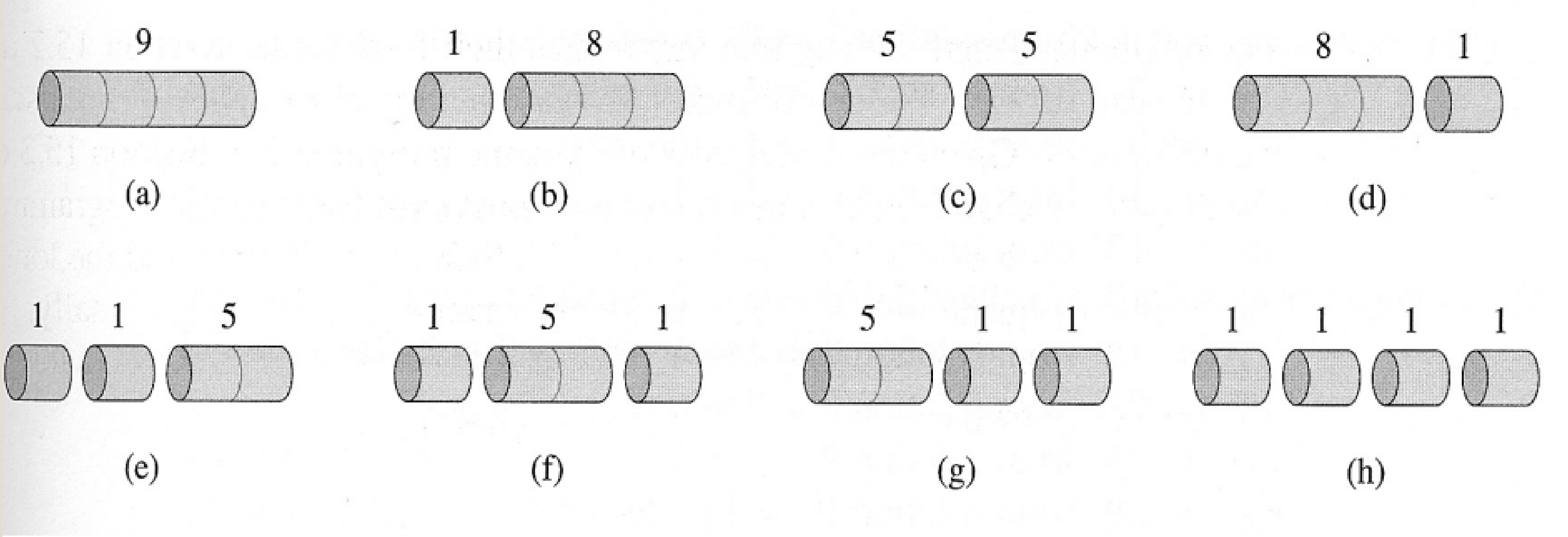
Making one cut creating pieces of length 1 and 4

- profit of 3 + 13= 15

Profit of 16 is possible (two length 1 pieces + one length 3)

- profit of p(1)+p(1)+p(3)=3+3+10 = 16

There are 2^{n-1} ways to cut a rod of length n.



n = 4

p = [1, 5, 8, 9]

Can we use DP?

Does the Principle of Optimality hold?

Goal: Characterize optimal solution in terms of solution(s) to smaller sub-problems

- DP computes the optimal solution to many optimal subproblems
    - Computed results are stored in a table (entries are never recomputed)
    - A DP algorithm does not know which subsolutions will be used in the optimum solution

Can we use DP?

Does the Principle of Optimality hold?

Assume we make an optimal cut creating one piece of length k and one of length n-k.

Then, both pieces are cut in an optimal way. Why? Otherwise we don’t have an optimal solution.

Can we use DP?

Does the Principle of Optimality hold?

Assume we make an optimal cut creating one piece of length k and one of length n-k.

Then, both pieces are cut in an optimal way. Why? Otherwise we don’t have an optimal solution.

How about overlapping subproblems?

Clicker Question

Let opt(n) be the profit of an optimal solution for a rod of even length **n**. Which of the following claims are necessarily true?

A. Opt(n) = p[n]
B. Opt(n) = Opt(n/2) + Opt(n/2)
C. Opt(n) = max{p[n], p[1] + Opt(n-1), p[2]+Opt(n-2)}
D. Opt(n) ≥ p[1] + Opt(n-1)
E. None of the claims are necessarily true!

Clicker Question

Let opt(n) be the profit of an optimal solution for a rod of even length **n**. Which of the following claims are necessarily true?

A. Opt(n) = p[n]
   - (e.g., p[n]=10, p[n-1]=9, p[1]=2)

B. Opt(n) = Opt(n/2) + Opt(n/2)
   - E.g., (p[n]=2n, p[i] = i for all i < n)

C. Opt(n) = max{p[n], p[1] + Opt(n-1), p[2]+Opt(n-2)}
   - E.g., (p[1]=p[2]=1, p[3] = …=p[n]=5)

D. Opt(n) ≥ p[1] + Opt(n-1)
   - **Equality holds if optimal solution includes rod of unit length**

E. None of the claims are necessarily true!

How to use DP?

- If we make an optimal cut creating a piece of length k and one of length n-k, both pieces are cut in a optimal way.
- Let opt(n) be the profit of an optimal solution for a rod of length **n**. Then,

$$\mathbf{opt(n) = max\ \{p[n],\ opt(1)+opt(n-1),} \\ \mathbf{opt(2)+opt(n-2),} \\ \ldots \\ \mathbf{opt(n-2)+opt(2),} \\ \mathbf{opt(n-1)+opt(1)\}}$$

Another way to look at the cuts …

$$\mathbf{opt(n) = max}_{1 \le i \le n} \{\mathbf{p[i] + opt(n-i)}\}$$

Another way to look at the cuts …

$$\mathbf{opt(n) = max}_{1 \leq i \leq n} \{\mathbf{p[i] + opt(n-i)}\}$$

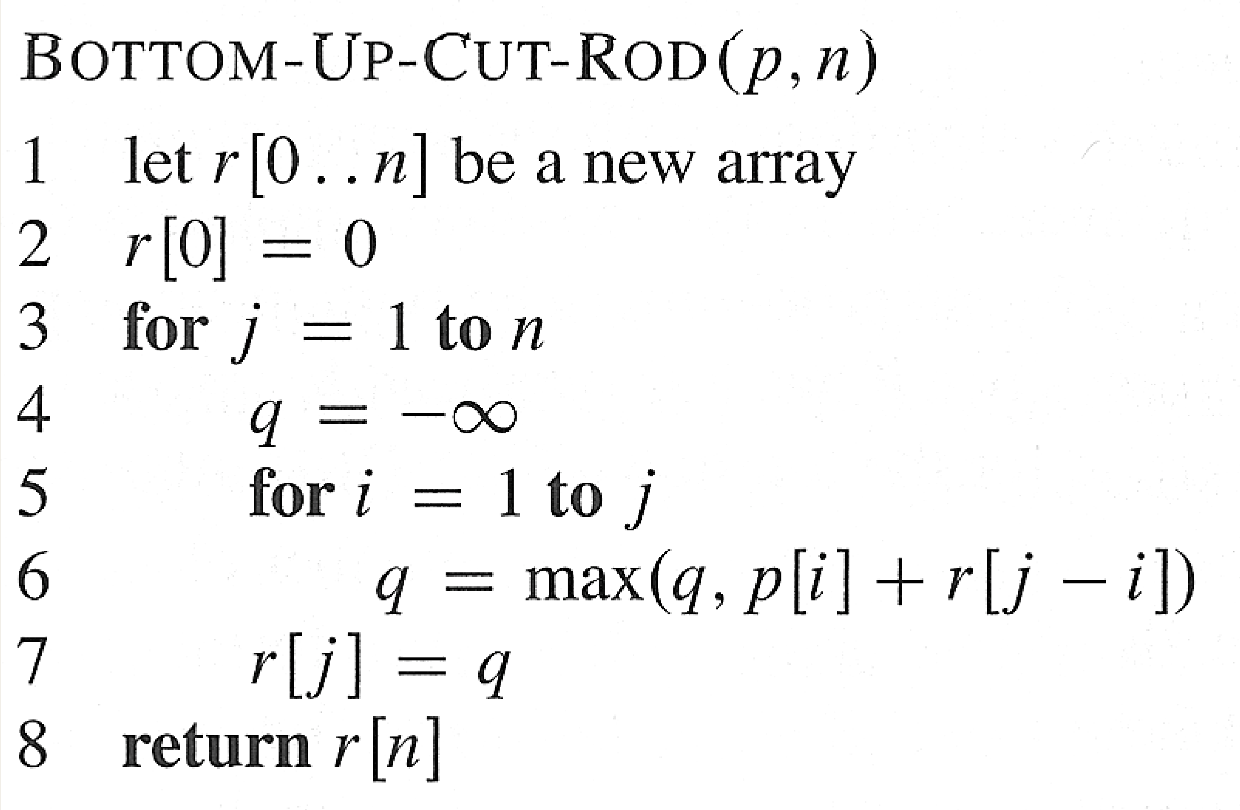
If a piece of length i is the leftmost piece cut from the rod, it generates a profit of p[i].

The remaining rod of length n-i is cut in an optimal way maximizing the profit.

New recurrence:

$$\mathbf{opt(j) = max}_{1 \leq i \leq j} \{\mathbf{p[i] + opt(j-i)}\} \textbf{ for } 1 \leq j \leq n$$

$r(j) = \max_{1 \le i \le j} \{p[i] + r(j-i)\}$ for $1 \le j \le n$ (r stands for opt)

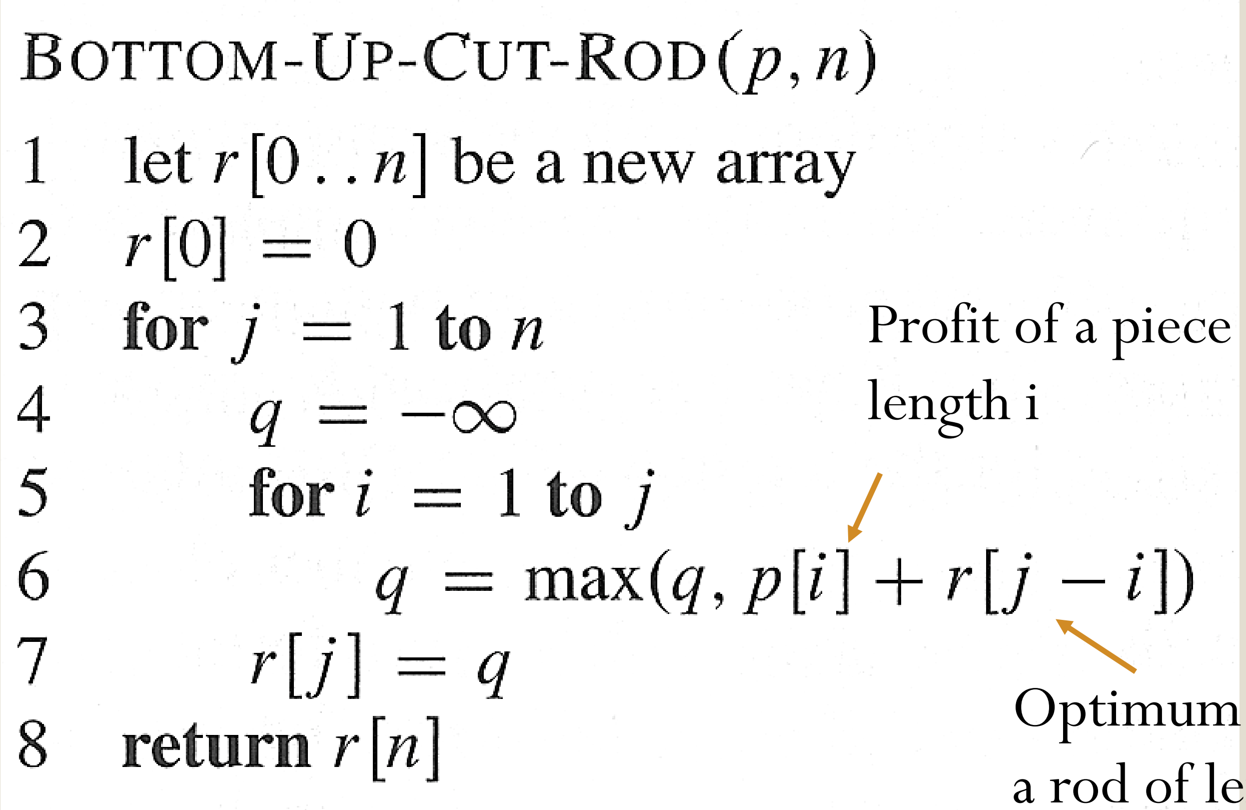


```
BOTTOM-UP-CUT-ROD(p, n)
let r[0 . . n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

Total time is $O(n^2)$ and space is $O(n)$.

See page 366 for more details.

$r(j) = \max_{1 \le i \le j} \{p[i] + r(j-i)\}$ for $1 \le j \le n$ (r stands for opt)



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

Profit of a piece of length i

Optimum solution for a rod of length j-i

Total time is $O(n^2)$ and space is $O(n)$.

See page 366 for more details.

Example

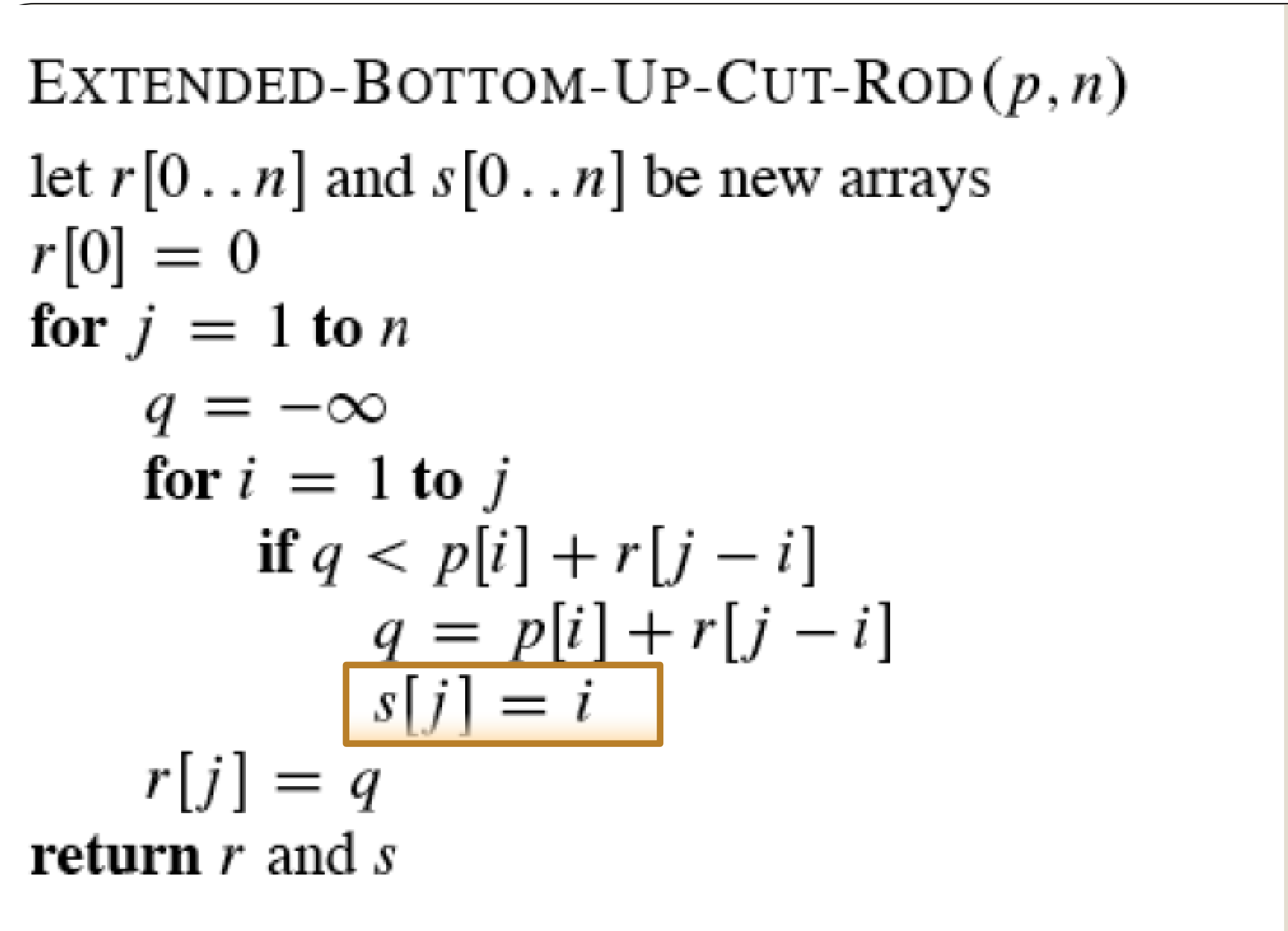
n=5

	1	2	3	4	5
p	**3**	**5**	**10**	**12**	**14**
opt (r)	**3**	**6**	**10**	**13**	**16**

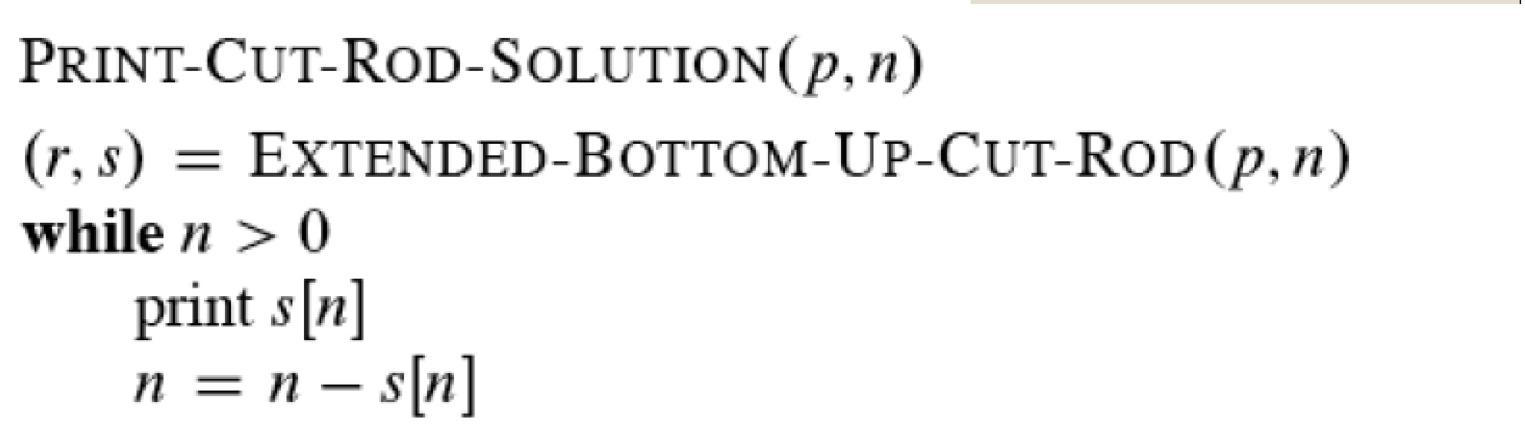
How to record where the cuts are made?

Use an arrays to record which index **k** resulted in the maximum for opt(j)

- Needs some adjusting of indices to generate cut positions



```
EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
let $r[0 \twodots n]$ and $s[0 \twodots n]$ be new arrays
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        if $q < p[i] + r[j - i]$
            $q = p[i] + r[j - i]$
            $\boxed{s[j] = i}$
    $r[j] = q$
return $r$ and $s$
```



```
PRINT-CUT-ROD-SOLUTION$(p, n)$
$(r, s) =$ EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
while $n > 0$
    print $s[n]$
    $n = n - s[n]$
```

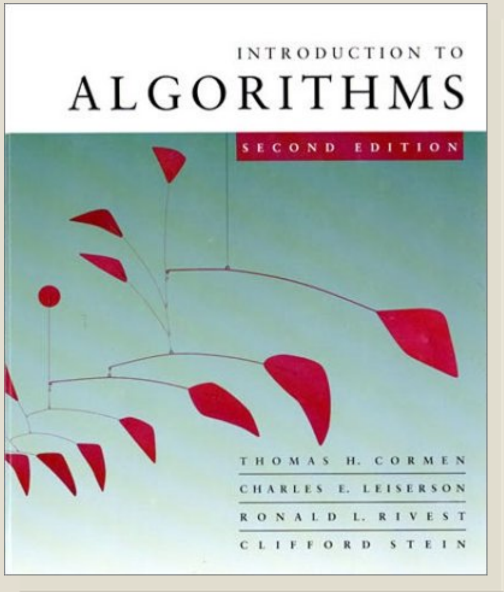
Dynamic Programming Problems

1) Non-Adjacent Selection
2) Rod Cutting
3) Weighted Selection

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

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

- Longest Common Subsequence
- Optimal substructure
- Overlapping subproblems
- Sequence alignment (Edit Dist.)

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

Problem 4: *Longest Common Subsequence (LCS)*

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

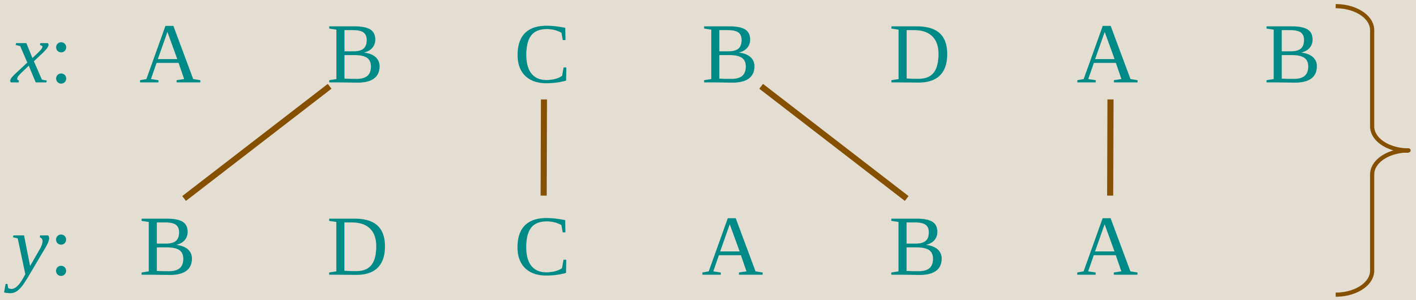
y: B D C A B A

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

Recursive formulation

Theorem.

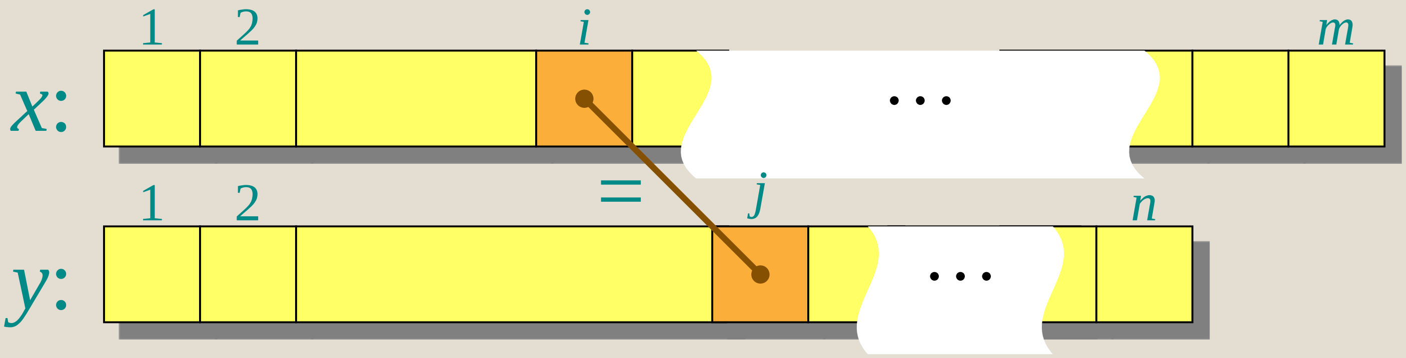
$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

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

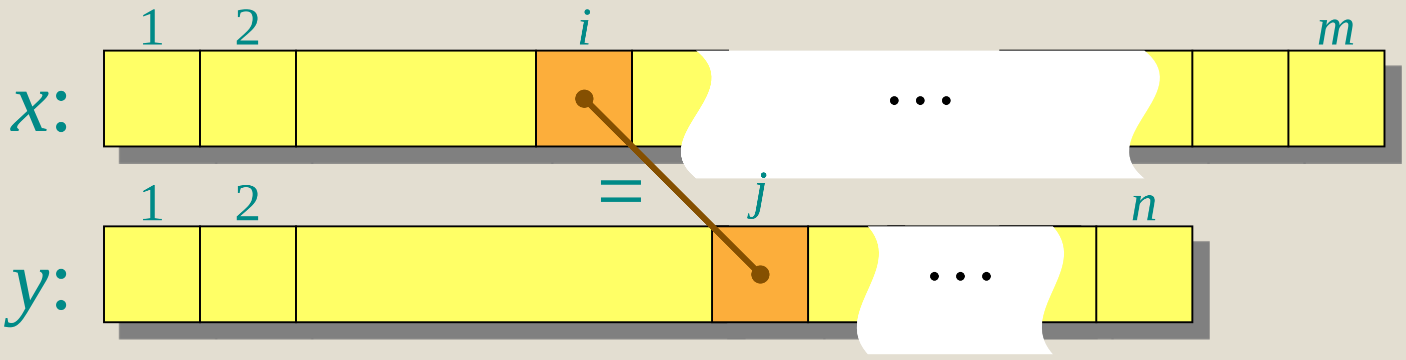


Recursive formulation

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

Suppose w is a longer CS of $x[1 \,.\,.\, i\text{–}1]$ and $y[1 \,.\,.\, j\text{–}1]$, that is, $|w| > k\text{–}1$. 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 $\big|w \,\|\, z[k]\big| > k$. 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 $\left|w \,\|\, z[k]\right| > k$. 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. □

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