# Data Structures and Algorithm Analysis (CSC317)

## Dynamic Programming 2

Odelia Schwartz

# Dynamic Programming

- Problems that may naively have exponential running time, but can be made polynomial (fast!)

- Dynamic: "I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying… It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense."

http://www.cs.miami.edu/home/odelia/teaching/csc317_fall19/syllabus/dy_birth.pdf

- Programming: Not programming languages; Bellman was interested in "planning and decision making."

- Main approach: hold answers to previous problems already solved in a table, to be used again without recomputing.

# Dynamic Programming so far

Main properties:

1. Overlapping subproblems (same subproblems solved over and over again)

2. Solution to big problem constructed from solutions to smaller subproblems (optimal substructure; more on later)

We'll want to contrast with other algorithmic approaches, such as divide and conquer…

# Dynamic Programming so far

Main properties:

1. Overlapping subproblems (same subproblems solved over and over again)

2. Solution to big problem constructed from solutions to smaller subproblems (optimal substructure; more on later)

To make algorithm more efficient, what did we do?

# Dynamic Programming so far

Main properties:

1. Overlapping subproblems (same subproblems solved over and over again
2. Solution to big problem constructed from solutions to smaller subproblems (optimal substructure; more on later)

To make algorithm more efficient, we either **(i) memoized** (saved solutions to smaller subproblems in a table as we recursed; "recursive solution "remembers" what results it has computed previously"); or we saved solutions to subproblems in a table **(ii) bottom-up.** These turned out equivalent.

# We did: Fibonacci Memoized and Bottom-up Dynamic Programming

See online by Galles:
https://www.cs.usfca.edu/~galles/visualization/DPFib.html

Runtime?

# Dynamic Programming Class Outline

- Examples of applications (motivation)
- Simple example to gain intuition (Fib)
- Back to applications and more examples

# Examples of applications

- Computational Biology (genome similarity)

Strings from alphabet {A, C, G, T}

Example: ACGGAT
CCGCTT

What is the Longest Common Subsequence?

Answer: 3 CGT

LCS(6,6) = 3 // length of Longest Common Subsequence

# Examples of applications

- Computational Biology (genome similarity)

  What is the Longest Common Subsequence?
  A C C C G G T C G A G T G …
  G T C G T T C G G A A T T …

  Brute force: Try all subsequences in 1st string and compare to second string…
  n=500 then 2^500 possibilities

  Pick first character or do not…
  Pick 2nd character or do not…
  Pick 3rd or do not…
  2 * 2 * 2 * 2 …. * 2 (n times)

# Longest Common Subsequence

- Formulating the recursion

- We'll try and start from the largest sequence, and then formulate the recursion for smaller subproblems

# Longest Common Subsequence

- Look at example

C C G C T T

A C G G A T

# Longest Common Subsequence

- Look at example

C C G C T T

A C G G A T

Last letter of both strings identical

What to do??

# Longest Common Subsequence

- Look at example

C C G C T T

A C G G A T

Last letter of both strings identical:
Recurse on LCS(5,5)

Solution here?

# Longest Common Subsequence

- Look at example

C C G C T T

A C G G A T

Last letter of both strings identical:
Recurse on LCS(5,5)

Solution here?
LCS(6,6) = LCS(5,5) + 1 = … 3
CCGCT T
ACGGA T

# Longest Common Subsequence

- Look at example

C C G C T C

A C G G A T

Last letter of both strings different:
What to do??

# Longest Common Subsequence

- Look at example

C C G C T C        C C G C T C

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

Last letter of both strings different:

LCS[6,6] = max(LCS[5,6], LCS(6,5]) = … 3

CCGCT        CCGCTC
ACGGAT       ACGGA

# Longest Common Subsequence

- Look at example

C C G C T C    C C G C T C

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

Last letter of both strings different:

LCS[6,6] = max(LCS[5,6], LCS(6,5]) = … 3

| | |
|---|---|
| CCGCT | CCGCTC |
| ACGGAT | ACGGA |
| = 3 CGT | = 2 CG |

# Longest Common Subsequence

- Summary so far

  Let c hold the length of the LCS

  The first string is x (indexed by i)

  Second string is y (indexed by j)

From textbook:

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

# Longest Common Subsequence

- We've structured as large subproblem composed of small subproblems
- If we know optimal solution to smaller subproblems, we can obtain optimal solution to larger subproblem

## From textbook:

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

# Longest Common Subsequence

From textbook:

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

Question: Is this recursive solution efficient?

# Longest Common Subsequence

From textbook:

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

Answer: Not efficient; only if memoized previous solutions (or build bottom-up) – just like with Fib

# Longest Common Subsequence

From textbook:

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

Question: Are there overlapping subproblems?

Recursion tree on the board…

# Longest Common Subsequence

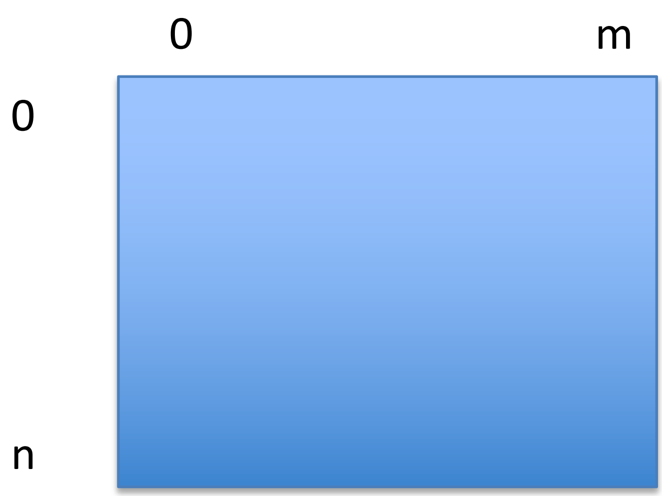
Dynamic Programming solution:

Needs a table. In Fib length n.
Here??

# Longest Common Subsequence

Dynamic Programming solution:

- Define table c[0..m, 0..n]
  n = x.length (of first subsequence)
  m = y.length (of second subsequence)



# Longest Common Subsequence

Animation by Galles:

https://www.cs.usfca.edu/~galles/visualization/DPLCS.html

Bottom-up: we impose order
Memoized: order imposed by recursion

# Longest Common Subsequence

Dynamic Programming solution:

- Main approach: Either memoize solutions to subproblems not yet computed, or compute solutions to subproblems bottom-up
- We'll see that runtime is $\Theta(mn)$.
  mn subproblems
  constant computation each
- We'll write out Bottom-up (memoized as assignment)

# Longest Common Subsequence

Main properties that allow DP:

- Overlapping subproblems
- Solution to big problem constructed from solutions to smaller subproblem (optimal)

# Bottom-up LCS from book

```
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 LCS from book

Runtime: $\Theta(mn)$.

Size of table (mn)
Times constant operations per table entry (up to 3!)

# Bottom-up LCS from book

Example on the board…

# Bottom-up LCS from book

## Printing result

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

# DP so far

- Problems that naively can appear exponential time
- But via recursion and memoization, or bottom-up filling a table, become polynomial
- Main idea: Save solutions to subproblems in a table that can later be accessed

# DP so far

- Fibonacci:
  - number of subproblems = table size
  - for each subproblem, look at how many choices of previous subproblems

- LCS:
  - number of subproblems = table size
  - for each subproblem, look at how many choices of previous subproblems

# DP so far

- Fibonacci: $\Theta(n)$
  - number of subproblems = table size: n
  - for each subproblem, look at how many choices of previous subproblems? 2

- LCS: $\Theta(mn)$
  - number of subproblems = table size: n x m
  - for each subproblem, look at how many choices of previous subproblems? Up to 3

# Another DP example

- Rod-cutting problem
- First DP problem in the book…
- Table size n but may have up to n choices…

# Rod cutting problem

We are given prices $p_i$ for each rod of length $i$

| 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 |

Question: We are given a rod of length n, and want to maximize revenue, by cutting up the rod into pieces and selling each of the pieces.

# Rod cutting problem

## Example: 4 inch rod. Best solution?
## We'll first list all solutions…

1. Cut into 2 pieces length 2:

$$p_2 + p_2 = 5 + 5 = 10$$

2. Cut into 4 pieces length 1:

$$p_1 + p_1 + p_1 + p_1 = 1 + 1 + 1 + 1 = 4$$

3-4. Cut into 2 pieces, length 1 and length 3 (or vice versa length 3 and then 1):

$$p_1 + p_3 = 1 + 8 = 9; p_3 + p_1 = 8 + 1 = 9$$

5. Keep length 4:

$$p_4 = 9$$

6-8: Cut into 3 pieces, length 1, 1, and 2 (any order):

$$p_1 + p_1 + p_2 = 7; p_2 + p_1 + p_1 = 7; p_1 + p_2 + p_1 = 7$$

# Rod cutting problem

## Example: 4 inch rod. Best solution?
## We'll first list all solutions…

1. Cut into 2 pieces length 2:

$$p_2 + p_2 = 5 + 5 = 10$$

2. Cut into 4 pieces length 1:

$$p_1 + p_1 + p_1 + p_1 = 1 + 1 + 1 + 1 = 4$$

3-4. Cut into 2 pieces, length 1 and length 3 (or vice versa length 3 and then 1):

$$p_1 + p_3 = 1 + 8 = 9; p_3 + p_1 = 8 + 1 = 9$$

5. Keep length 4:

$$p_4 = 9$$

6-8: Cut into 3 pieces, length 1, 1, and 2 (any order):

$$p_1 + p_1 + p_2 = 7; p_2 + p_1 + p_1 = 7; p_1 + p_2 + p_1 = 7$$

# Rod cutting problem

Total: 8 cases for n=4 $(=2^{n-1})$. We can slightly reduce by always requiring cuts in non-decreasing order. But still a lot!

Note: We've computed a brute force solution; all possibilities for this simple small example. But we want more optimal solution!

# Rod cutting problem

Will Divide and Conquer work?

Maybe, but need to think about how to combine solutions…

On the board… length 8, conquer each 4;
Best solution 10+10=20
But dividing into 6 and 2 yields 17+5=22 better!

# Rod cutting problem One solution


- Cut rod into length i and n-i
- Recurse on n-i

# Rod cutting problem One solution


- Cut rod into length i and n-i
- Recurse on n-i

# Rod cutting problem

We'll define:

a. Maximum revenue for log of size n: $r_n$
(this is the solution we want to find)

b. Revenue (price) for single log of length i: $p_i$

Example: If we cut log into length i and n-i:

Revenue: $p_i + r_{n-i}$

(this can be seen as recursing on n-i)

# Rod cutting problem

Many possible choices of i…

$$r_n = \max \begin{Bmatrix} p_1 + r_{n-1} \\ p_2 + r_{n-2} \\ \ldots \\ p_n + r_0 \end{Bmatrix}$$

Size 1, recurse on n-1

Size 2, recurse on n-2

Size n, recurse on nothing

# Rod cutting problem

Recursive solution…

```
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
```

# Rod cutting problem

Recursive solution…

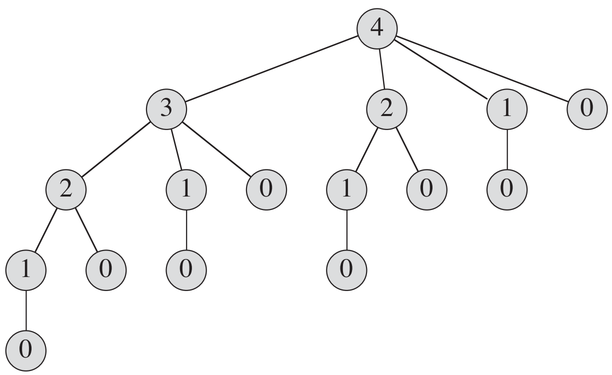
```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

Why is this so slow?

# Rod cutting problem

## Recursive solution… why is this so slow?

Cut-rod calls itself repeatedly with the same parameter values. We can see by plotting a tree:




- Node label = size of subproblem called on
- Can see by eye that many subproblems called repeatedly. We call this a problem with subproblem overlap.
- Number of nodes exponential in n ( $2^n$ ); therefore exponential number of calls to Cut-Rod

Leaves: each possible way of cutting rod; either cut or not at each position 2^(n-1)

# Rod cutting problem: memoized solution

Step 1: Initialization:

```
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)
```

Creates array for holding memoized results, and initialized to minus infinity. Then calls the main auxiliary function

# Rod cutting problem: memoized DP

Step 2: The main auxiliary function, which goes through the lengths, computes answers to subproblems and memoizes if subproblem not yet encountered:

```
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
```

# Rod cutting problem: Bottom-up DP

```
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]
```

# Rod cutting problem: Bottom-up DP

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

Lines 1-2 check if value already known or memoized; Lines 3-7 compute the maximal revenue if it has not already been memoized, and line 8 saves it.

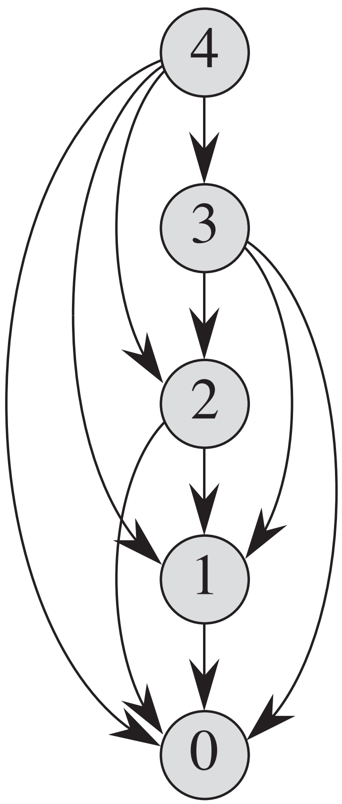
<u>Run time: For both top-down and bottom-up versions:</u>

$O(n^2)$

Easiest to see for bottom-up version: doubly-nested for loop.

# Rod cutting problem

- We can also view graph form; reduce previous tree that included all subproblems repeatedly…




- Each vertex represents subproblem of given size
- Vertex label = subproblem size
- Edge from x to y: We need a solution to subproblem y when solving subproblem x
- Runtime equal to number of edges $O(n^2)$
- Runtime a combination of number of items in the table (n) and work per item (n). The work per item is due to the max operation (needed even if the table is filled and we just take values from the table) is proportional to n, as in the number of edges in the graph