Dynamic Programming

David Kauchak
cs302
Spring 2012

Dynamic programming

- One of the most important algorithm tools!
- Very common interview question

- Method for solving problems where optimal solutions can be defined in terms of optimal solutions to sub-problems

  AND
- the sub-problems are overlapping

Fibonacci numbers

1, 1, 2, 3, 5, 8, 13, 21, 34, …

What is the recurrence for the n^{th} Fibonacci number?

F(n) = F(n-1) + F(n-2)

The solution for n is defined with respect to the solution to smaller problems (n-1 and n-2)

Fibonacci: a first attempt

```
FIBONACCI(n)
  if n = 1 or n = 2
      return 1
  else
      return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

Is it correct?

FIBONACCI(n)
if $n = 1$ or $n = 2$
  return 1
else
  return FIBONACCI($n-1$) + FIBONACCI($n-2$)

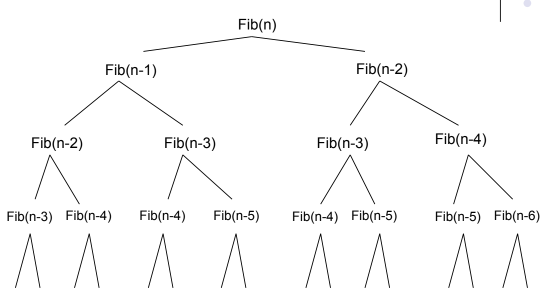
F(n) = F(n-1) + F(n-2)

Running time

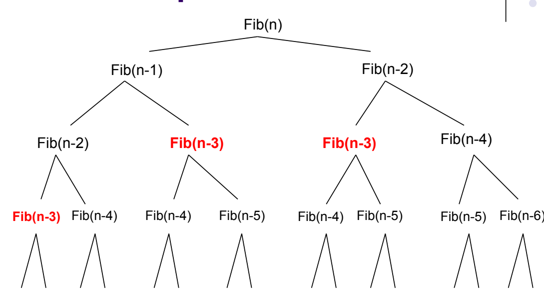
FIBONACCI(n)
if $n = 1$ or $n = 2$
  return 1
else
  return FIBONACCI($n-1$) + FIBONACCI($n-2$)

- Each call creates two recursive calls
- Each call reduces the size of the problem by 1 or 2
- Creates a full binary of depth n
- $O(2^n)$

Can we do better?



A lot of repeated work!



Identifying a dynamic programming problem

The solution can be defined with respect to solutions to subproblems

The subproblems created are overlapping, that is we see the same subproblems repeated

Creating a dynamic programming solution

Step 1: Identify a solution to the problem with respect to **smaller** subproblems

- F(n) = F(n-1) + F(n-2)

Step 2: **bottom up** - start with solutions to the smallest problems and build solutions to the larger problems

```
FIBONACCI-DP(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← fib[i − 1] + fib[i − 2]
return fib[n]
```

use an array to store solutions to subproblems

Is it correct?

```
FIBONACCI-DP(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← fib[i − 1] + fib[i − 2]
return fib[n]
```

F(n) = F(n-1) + F(n-2)

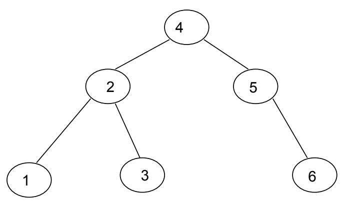
Running time?

```
FIBONACCI-DP(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← fib[i − 1] + fib[i − 2]
return fib[n]
```

$\Theta(n)$

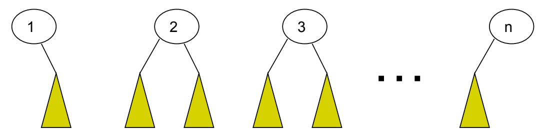
Counting binary search trees

- How many unique binary search trees can be created using the numbers 1 through n?

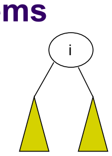


Step 1: What is the subproblem?

- Assume we have some black box solver (call it T) that can give us the answer to smaller subproblems
- How can we use the answer from this to answer our question?
- How many options for the root are there?

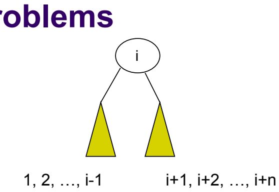


Subproblems



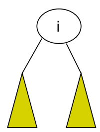
How many trees have i as the root?

Subproblems



?

Subproblems

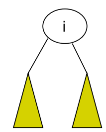


1, 2, …, i-1 i+1, i+2, …, i+n

T(i-1) ?

subproblem of
size i-1

Subproblems

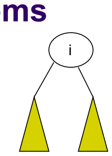


1, 2, …, i-1 i+1, i+2, …, i+n

T(i-1)

Number of trees for i+1, i+2, …, i+n
is the same as the number of trees
from 1, 2, …, n-i

Subproblems

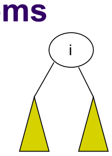


1, 2, …, i-1 i+1, i+2, …, i+n

T(i-1) T(n-i)

Given solutions for T(i-1) and T(n-i) how
many trees are there with i as the root?

Subproblems



1, 2, …, i-1 i+1, i+2, …, i+n

T(i-1) T(n-i)

T(i) = T(i-1) * T(n-i)

Step 1: define the answer with respect to subproblems

T(i) = T(i-1) * T(n-i)

$$T(n) = \sum_{i=1}^{n} T(i-1) * T(n-i)$$

```
BST-Count(n)
if n = 0
        return 1
else
        sum = 0
        for i ← 1 to n
                sum ← sum + BST-Count(i − 1) ∗ BST-Count(n − i)
return sum
```

Is there a problem?

```
BST-Count(n)
if n = 0
        return 1
else
        sum = 0
        for i ← 1 to n
                sum ← sum + BST-Count(i − 1) ∗ BST-Count(n − i)
return sum
```

As with Fibonacci, we're repeating a lot of work

Step 2: Generate a solution from the bottom-up

```
BST-Count(n)
if n = 0
        return 1
else
        sum = 0
        for i ← 1 to n
                sum ← sum + BST-Count(i − 1) ∗ BST-Count(n − i)
return sum
```

```
BST-Count-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] ∗ c[k − i]
return c[n]
```

```
BST-Count-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] ∗ c[k − i]
return c[n]
```

0 1 2 3 4 5 … n

```
BST-Count-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] ∗ c[k − i]
return c[n]
```

1 1

0 1 2 3 4 5 … n

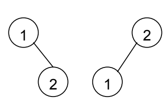
```
BST-Count-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] ∗ c[k − i]
return c[n]
```

c[0]*c[1] + c[1]*c[0]

1 1

0 1 2 3 4 5 … n

```
BST-Count-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] ∗ c[k − i]
return c[n]
```



c[0]*c[1] + c[1]*c[0]

1 1

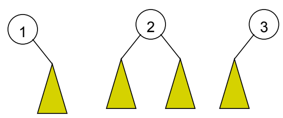
0 1 2 3 4 5 … n

```
BST-Count-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] ∗ c[k − i]
return c[n]
```

1 1 2

0 1 2 3 4 5 … n

```
BST-COUNT-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```



c[0]*c[2] + c[1]*c[1] + c[2]*c[0]

1 1 2

0 1 2 3 4 5 … n

```
BST-COUNT-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

1 1 2 5

0 1 2 3 4 5 … n

```
BST-COUNT-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

1 1 2 5 …

0 1 2 3 4 5 … n

Running time?

```
BST-COUNT-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

$\Theta(n^2)$

Longest common subsequence (LCS)

For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

ABA?

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

ABA

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

ACA?

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

ACA

Longest common subsequence (LCS)

- For a sequence X = $x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices ($i_1, i_2, \ldots, i_k$) where $1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

DCA?

Longest common subsequence (LCS)

- For a sequence X = $x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices ($i_1, i_2, \ldots, i_k$) where $1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

~~DCA~~

Longest common subsequence (LCS)

- For a sequence X = $x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices ($i_1, i_2, \ldots, i_k$) where $1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

AADAA?

Longest common subsequence (LCS)

- For a sequence X = $x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices ($i_1, i_2, \ldots, i_k$) where $1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

AADAA

LCS problem

Given two sequences X and Y, a **common subsequence** is a subsequence that occurs in both X and Y

Given two sequences X = $x_1, x_2, \ldots, x_n$ and Y = $y_1, y_2, \ldots, y_n$, *What is the longest common subsequence?*

X = A B C B D A B

Y = B D C A B A

LCS problem

- Given two sequences X and Y, a **common subsequence** is a subsequence that occurs in both X and Y
- Given two sequences X = $x_1, x_2, \ldots, x_n$ and Y = $y_1, y_2, \ldots, y_n$, What is the longest common subsequence?

X = A B C B D A B

Y = B D C A B A

Step 1: Define the problem with respect to subproblems

X = A B C B D A B

Y = B D C A B A

Step 1: Define the problem with respect to subproblems

X = A B C B D A ?

Y = B D C A B ?

Is the last character part of the LCS?

Step 1: Define the problem with respect to subproblems

X = A B C B D A ?

Y = B D C A B ?

Two cases: either the characters are the same or they're different

Step 1: Define the problem with respect to subproblems

X = $\boxed{\text{A B C B D A}}$A

LCS

Y = $\boxed{\text{B D C A B}}$A

The characters are part of the LCS

What is the recursive relationship?

If they're the same

$$LCS(X,Y) = LCS(X_{1...n-1}, Y_{1...m-1}) + x_n$$

Step 1: Define the problem with respect to subproblems

X = $\boxed{\text{A B C B D A}}$B

LCS

Y = $\boxed{\text{B D C A B A}}$

If they're different

$$LCS(X,Y) = LCS(X_{1...n-1}, Y)$$

Step 1: Define the problem with respect to subproblems

X = $\boxed{\text{A B C B D A B}}$

LCS

Y = $\boxed{\text{B D C A B}}$A

If they're different

$$LCS(X,Y) = LCS(X, Y_{1...m-1})$$

Step 1: Define the problem with respect to subproblems

X = A B C B D A B

Y = B D C A B A

?

X = A B C B D A B

Y = B D C A B A

If they're different

Step 1: Define the problem with respect to subproblems

X = A B C B D A B

Y = B D C A B A

$$LCS(X,Y) = \begin{cases} 1 + LCS(X_{1\ldots n-1}, Y_{1\ldots m-1}) & \text{if } x_n = y_m \\ \max(LCS(X_{1\ldots n-1}, Y), LCS(X, Y_{1\ldots m-1})) & \text{otherwise} \end{cases}$$

(for now, let's just worry about counting the length of the LCS)

Step 2: Build the solution from the bottom up

$$LCS(X,Y) = \begin{cases} 1 + LCS(X_{1\ldots n-1}, Y_{1\ldots m-1}) & \text{if } x_n = y_m \\ \max(LCS(X_{1\ldots n-1}, Y), LCS(X, Y_{1\ldots m-1})) & \text{otherwise} \end{cases}$$

What types of subproblem solutions do we need to store?

$LCS(X_{1\ldots j}, Y_{1\ldots k})$

two different indices

Step 2: Build the solution from the bottom up

$$LCS(X,Y) = \begin{cases} 1 + LCS(X_{1\ldots n-1}, Y_{1\ldots m-1}) & \text{if } x_n = y_m \\ \max(LCS(X_{1\ldots n-1}, Y), LCS(X, Y_{1\ldots m-1})) & \text{otherwise} \end{cases}$$

What types of subproblem solutions do we need to store?

$LCS(X_{1\ldots j}, Y_{1\ldots k})$

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \textit{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i							
1	A							
2	B							
3	C							
4	B							
5	D							
6	A							
7	B							

For Fibonacci and tree counting, we had to initialize some entries in the array. Any here?

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \textit{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Need to initialize values within 1 smaller in either dimension.

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \textit{if } x_i = y_j \\ \boxed{\max(LCS[i-1, j], LCS[i, j-1]) \quad \text{otherwise}} \end{cases}$$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	?					
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS(A, B)

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \textit{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0					
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

$$LCS[i,j] = \begin{cases} \boxed{1 + LCS[i-1, j-1] \quad \text{if } x_i = y_j} \\ \max(LCS[i-1, j], LCS[i, j-1] \quad \text{otherwise} \end{cases}$$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	?		
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS(A, BDCA)

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1] & \text{otherwise} \end{cases}$$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1		
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS(A, BDCA)

$$LCS[i,j] = \begin{cases} \boxed{1 + LCS[i-1, j-1] \quad \text{if } x_i = y_j} \\ \max(LCS[i-1, j], LCS[i, j-1] \quad \text{otherwise} \end{cases}$$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	?	
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCAB)

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1] & \text{otherwise} \end{cases}$$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCAB)

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i		j: 0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Where's the final answer?

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0,0] ← 0
for i ← 1 to m
        c[i,0] ← 0
for j ← 1 to n
        c[0,j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i,j] ← 1 + c[i-1, j-1]
                elseif c[i-1, j] > c[i, j-1]
                        c[i,j] ← c[i-1, j]
                else
                        c[i,j] ← c[i, j-1]
return c[m,n]
```

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0,0] ← 0
for i ← 1 to m
        c[i,0] ← 0
for j ← 1 to n
        c[0,j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i,j] ← 1 + c[i-1, j-1]
                elseif c[i-1, j] > c[i, j-1]
                        c[i,j] ← c[i-1, j]
                else
                        c[i,j] ← c[i, j-1]
return c[m,n]
```

Base case initialization

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0,0] ← 0
for i ← 1 to m
        c[i,0] ← 0
for j ← 1 to n
        c[0,j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i,j] ← 1 + c[i-1, j-1]
                elseif c[i-1, j] > c[i, j-1]
                        c[i,j] ← c[i-1, j]
                else
                        c[i,j] ← c[i, j-1]
return c[m,n]
```

Fill in the matrix

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0, 0] ← 0
for i ← 1 to m
        c[i, 0] ← 0
for j ← 1 to n
        c[0, j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i, j] ← 1 + c[i − 1, j − 1]
                elseif c[i − 1, j] > c[i, j − 1]
                        c[i, j] ← c[i − 1, j]
                else
                        c[i, j] ← c[i, j − 1]
return c[m, n]
```

The algorithm

LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0, 0] ← 0
for i ← 1 to m
c[i, 0] ← 0
for j ← 1 to n
c[0, j] ← 0
for i ← 1 to m
for j ← 1 to n
if x_i = y_i
c[i, j] ← 1 + c[i − 1, j − 1]
elseif c[i − 1, j] > c[i, j − 1]
c[i, j] ← c[i − 1, j]
else
c[i, j] ← c[i, j − 1]
return c[m, n]

The algorithm

LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0, 0] ← 0
for i ← 1 to m
c[i, 0] ← 0
for j ← 1 to n
c[0, j] ← 0
for i ← 1 to m
for j ← 1 to n
if x_i = y_i
c[i, j] ← 1 + c[i − 1, j − 1]
elseif c[i − 1, j] > c[i, j − 1]
c[i, j] ← c[i − 1, j]
else
c[i, j] ← c[i, j − 1]
return c[m, n]

Running time?

LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0, 0] ← 0
for i ← 1 to m
c[i, 0] ← 0
for j ← 1 to n
c[0, j] ← 0
for i ← 1 to m
for j ← 1 to n
if x_i = y_i
c[i, j] ← 1 + c[i − 1, j − 1]
elseif c[i − 1, j] > c[i, j − 1]
c[i, j] ← c[i − 1, j]
else
c[i, j] ← c[i, j − 1]
return c[m, n]

Θ(nm)