Dynamic Programming continued

David Kauchak
cs302
Spring 2012

Admin

- CS lunch Thursday after class

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?

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

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

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

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

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

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

LCS Algorithm

$\Theta(nm)$

```
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]
```

Keeping track of the solution

Our LCS algorithm only calculated the length of the LCS between X and Y

What if we wanted to know the actual sequence?

Keep track of this as well…

```
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]
```

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

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

We can follow the arrows to generate the solution

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

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

We can follow the arrows to generate the solution

BCBA

Longest increasing subsequence

Given a sequence of numbers X = $x_1, x_2, \ldots, x_n$ find the longest increasing *subsequence* $(i_1, i_2, \ldots, i_k)$, that is a subsequence where numbers in the sequence increase.

5 2 8 6 3 6 9 7

Longest increasing subsequence

Given a sequence of numbers X = $x_1, x_2, \ldots, x_n$ find the longest increasing *subsequence* $(i_1, i_2, \ldots, i_k)$, that is a subsequence where numbers in the sequence increase.

5 2 8 6 3 6 9 7

Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

Two options:
Either 5 is in the
LIS or it's not

Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

include 5

5 + LIS(8 6 3 6 9 7)

Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

include 5

5 + LIS(8 6 3 6 9 7)

What is this function exactly?

longest increasing sequence of the numbers

longest increasing sequence of the numbers starting with 8

Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

include 5

5 + LIS(8 6 3 6 9 7)

What is this function exactly?

~~longest increasing sequence of the numbers~~

This would allow for the option of sequences starting with 3 which are NOT valid!

Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

include 5

$5 + \underbrace{LIS'}(8\ 6\ 3\ 6\ 9\ 7)$

longest increasing sequence of the numbers starting with 8

Do we need to consider anything else for subsequences starting at 5?

Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

include 5

5 + LIS'(8 6 3 6 9 7)

5 + LIS'(6 3 6 9 7)

5 + LIS'(6 9 7)

5 + LIS'(9 7)

5 + LIS'(7)

Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

don't include 5

LIS(2 8 6 3 6 9 7)

Anything else?

Technically, this is fine, but now we have LIS and LIS' to worry about.

Can we rewrite LIS in terms of LIS'?

Step 1: Define the problem with respect to subproblems

$$LIS(X) = \max_i \{LIS'(i)\}$$

Longest increasing sequence for X is the longest increasing sequence starting at any element

And what is LIS' defined as (recursively)?

Step 1: Define the problem with respect to subproblems

$$LIS(X) = \max_i \{LIS'(i)\}$$

Longest increasing sequence for X
is the longest increasing sequence
starting at any element

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i\ldots n})\}$$

Longest increasing sequence starting at i

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i\ldots n})\}$$

LIS' :

5 2 8 6 3 6 9 7

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i\ldots n})\}$$

LIS' : 1

5 2 8 6 3 6 9 7

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i\ldots n})\}$$

LIS' : 1

5 2 8 6 3 6 9 7

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS' :
1 1
5 2 8 6 3 6 9 7

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS' :
1 1
5 2 8 6 3 6 9 7

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS' :
2 1 1
5 2 8 6 3 6 9 7

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS' :
3 2 1 1
5 2 8 6 3 6 9 7

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS': 2 3 2 1 1

5 2 8 6 3 6 9 7

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS': 2 2 3 2 1 1

5 2 8 6 3 6 9 7

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS': 4 2 2 3 2 1 1

5 2 8 6 3 6 9 7

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS': 3 4 2 2 3 2 1 1

5 2 8 6 3 6 9 7

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS':	3	$\boxed{4}$	2	2	3	2	1	1
	5	$\boxed{2}$	8	6	3	6	9	7

$$LIS(X) = \max_i \{LIS'(i)\}$$

Step 2: build the solution from the bottom up

```
LIS(X)
n ← LENGTH(X)
create array lis with n entries
for i ← n to 1
    max ← 1
    for j ← i + 1 to n
        if X[j] > X[i]
            if 1 + lis[j] > max
                max ← 1 + lis[j]
    lis[i] ← max
max ← 0
for i ← 1 to n
    if lis[i] > max
        max ← lis[i]
return max
```

Step 2: build the solution from the bottom up

```
LIS(X)
n ← LENGTH(X)
create array lis with n entries
for i ← n to 1
    max ← 1
    for j ← i + 1 to n
        if X[j] > X[i]
            if 1 + lis[j] > max
                max ← 1 + lis[j]
    lis[i] ← max
max ← 0
for i ← 1 to n
    if lis[i] > max
        max ← lis[i]
return max
```

start from the end (bottom)

Step 2: build the solution from the bottom up

```
LIS(X)
n ← LENGTH(X)
create array lis with n entries
for i ← n to 1
    max ← 1
    for j ← i + 1 to n
        if X[j] > X[i]
            if 1 + lis[j] > max
                max ← 1 + lis[j]
    lis[i] ← max
max ← 0
for i ← 1 to n
    if lis[i] > max
        max ← lis[i]
return max
```

$$\boxed{LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}}$$

Step 2: build the solution from the bottom up

```
LIS(X)
  n ← LENGTH(X)
  create array lis with n entries
  for i ← n to 1
        max ← 1
        for j ← i + 1 to n
              if X[j] > X[i]
                    if 1 + lis[j] > max
                          max ← 1 + lis[j]
        lis[i] ← max
  max ← 0
  for i ← 1 to n
        if lis[i] > max
              max ← lis[i]
  return max
```

$$\boxed{LIS(X) = \max_i\{LIS'(i)\}}$$

Step 2: build the solution from the bottom up

```
LIS(X)
  n ← LENGTH(X)
  create array lis with n entries
  for i ← n to 1
        max ← 1
        for j ← i + 1 to n
              if X[j] > X[i]
                    if 1 + lis[j] > max
                          max ← 1 + lis[j]
        lis[i] ← max
  max ← 0
  for i ← 1 to n
        if lis[i] > max
              max ← lis[i]
  return max
```

initialization?

Running time?

```
LIS(X)
  n ← LENGTH(X)
  create array lis with n entries
  for i ← n to 1
        max ← 1
        for j ← i + 1 to n
              if X[j] > X[i]
                    if 1 + lis[j] > max
                          max ← 1 + lis[j]
        lis[i] ← max
  max ← 0
  for i ← 1 to n
        if lis[i] > max
              max ← lis[i]
  return max
```

$\Theta(n^2)$

Another solution

- Can we use LCS to solve this problem?

5 2 8 6 3 6 9 7

LCS

2 3 5 6 6 7 8 9

Another solution

- Can we use LCS to solve this problem?

5 2 8 6 3 6 9 7

LCS

2 3 5 6 6 7 8 9

Memoization

- Sometimes it can be a challenge to write the function in a bottom-up fashion
- Memoization:
  - Write the recursive function top-down
  - Alter the function to check if we've already calculated the value
  - If so, use the pre-calculate value
  - If not, do the recursive call(s)

Memoized fibonacci

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

```
FIBONACCI-MEMOIZED(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← ∞
return FIB-LOOKUP(n)
```

```
FIB-LOOKUP(n)
if fib[n] < ∞
        return fib[n]
x ← FIB-LOOKUP(n − 1) + FIB-LOOKUP(n − 2)
if x < fib[n]
        fib[n] ← x
return fib[n]
```

Memoized fibonacci

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

```
FIBONACCI-MEMOIZED(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← ∞
return FIB-LOOKUP(n)
```

```
FIB-LOOKUP(n)
if fib[n] < ∞
        return fib[n]
fib[n] ← FIB-LOOKUP(n − 1) + FIB-LOOKUP(n − 2)
return fib[n]
```

Memoized fibonacci

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

```
Fibonacci-Memoized(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
    fib[i] ← ∞
return Fib-Lookup(n)
```

Use ∞ to denote uncalculated

```
Fib-Lookup(n)
if fib[n] < ∞
    return fib[n]
x ← Fib-Lookup(n − 1) + Fib-Lookup(n − 2)
if x < fib[n]
    fib[n] ← x
return fib[n]
```

Memoized fibonacci

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

```
Fibonacci-Memoized(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
    fib[i] ← ∞
return Fib-Lookup(n)
```

What else could we use besides an array?

Use ∞ to denote uncalculated

```
Fib-Lookup(n)
if fib[n] < ∞
    return fib[n]
x ← Fib-Lookup(n − 1) + Fib-Lookup(n − 2)
if x < fib[n]
    fib[n] ← x
return fib[n]
```

Memoized fibonacci

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

```
Fibonacci-Memoized(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
    fib[i] ← ∞
return Fib-Lookup(n)
```

```
Fib-Lookup(n)
if fib[n] < ∞
    return fib[n]
x ← Fib-Lookup(n − 1) + Fib-Lookup(n − 2)
if x < fib[n]
    fib[n] ← x
return fib[n]
```

Check if we already calculated the value

Memoized fibonacci

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

```
Fibonacci-Memoized(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
    fib[i] ← ∞
return Fib-Lookup(n)
```

```
Fib-Lookup(n)
if fib[n] < ∞
    return fib[n]
x ← Fib-Lookup(n − 1) + Fib-Lookup(n − 2)
if x < fib[n]
    fib[n] ← x
return fib[n]
```

calculate the value

Memoized fibonacci

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

```
Fibonacci-Memoized(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← ∞
return Fib-Lookup(n)
```

```
Fib-Lookup(n)
if fib[n] < ∞
        return fib[n]
x ← Fib-Lookup(n − 1) + Fib-Lookup(n − 2)
if x < fib[n]
        fib[n] ← x
return fib[n]
```

store the value

Memoization

- Pros
  - Can be more intuitive to code/understand
  - Can be memory savings if you don't need answers to all subproblems
- Cons
  - Depending on implementation, larger overhead because of recursion (though often the functions are tail recursive)

Quick summary

- Step 1: Define the problem with respect to subproblems
  - We did this for divide and conquer too. What's the difference?
  - You can identify a candidate for dynamic programming if there is **overlap** or **repeated work** in the subproblems being created
- Step 2: build the solution from the bottom up
  - Build the solution such that the subproblems referenced by larger problems are already solved
  - Memoization is also an alternative

0-1 Knapsack problem

- **0-1 Knapsack** – A thief robbing a store finds n items worth $v_1, v_2, .., v_n$ dollars and weight $w_1, w_2, \ldots, w_n$ pounds, where v_i and w_i are integers. The thief can carry at most W pounds in the knapsack. Which items should the thief take if he/she wants to maximize value?
- Repetition is allowed, that is you can take multiple copies of any item

$$K(w) = \max_{i: w_i \le w} \{K(w - w_i) + v_i\}$$