DYNAMIC PROGRAMMING: MORE FUN!

David Kauchak
CS 140 – Spring 2023

Admin

Assignment 5

Group 4 assignment distribution

Midterm next Thursday (2/29)

Assignment 6
- Released early next week
- Due Friday before spring break

Midterm 1

2 pages of notes

Up through 2/13 (no dynamic programming)

Made a previous midterm available (note, it was challenging)

Midterm 1 topics

Math foundations
- log properties
- properties of exponentials

Proofs by induction (weak, strong, and structural)

Big-O (theta and omega)
- Proving and disproving
- Categories and function ordering
Midterm 1 topics

Recurrences
- Generating (i.e., given a function/algorithm, write the recurrence)
- Solving: recurrence tree, substitution, master method

Sorting
- Insertion sort, Selection sort, Mergesort, Quicksort
- Runtimes, properties (in-place, stable)

Order statistics
- median/selection
- run-time

Midterm 1 topics

Data structures
- BSTs, red black trees, binary heaps, binomial heaps
- Run-times and functionality basics

Amortized analysis
- Aggregate and accounting methods

Tree counting code

Where did “dynamic programming” come from?

I spent the Fall semester of 1950 in RAND. My first task
was to find a name for a newly discovered process.
“An interesting question is: What did the name,
‘Dynamic programming’ come from?” The 1950s were not
good years for mathematical research. We had a very inter-
esting mathematician in Washington named Wilson. He was
psychologically affected by the depression. He was
free and instead of the work, research. I’m not saying
the work is bad, I’m saying it probably. He was free
as he would take. And he would get into if people
saw for example. How would you solve a problem.
Not our monge, how he

Richard Bellman On the Birth of Dynamic Programming
Stuart Dreyfus
Dynamic programming

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

AND

the subproblems are overlapping

Dynamic programming: steps

1a) optimal substructure: optimal solutions to the problem incorporate optimal solutions to related subproblems
   - convince yourself that there is optimal substructure

1b) recursive definition: use this to recursively define the value of an optimal solution

2) DP solution: describe the dynamic programming table:
   - size, initial values, order in which it’s filled in, location of solution

3) Analysis: analyze space requirements, running time

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 = x₁, x₂, …, xₙ and
Y = y₁, y₂, …, yₙ

What is the longest common subsequence?

1b: recursive solution

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

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

Assume you have a solver for smaller problems
1b: recursive solution

X = A B C B D A ?

Y = B D C A B ?

Is the last character part of the LCS?

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

1b: recursive solution

X = A B C B D A ?

Y = B D C A B ?

If they’re the same

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

If they’re different

LCS(X, Y) = LCS(X_{1...n-1}, Y)
1b: recursive solution

\[
\begin{align*}
X &= \text{A B C B D A B} \\
Y &= \text{B D C A B A}
\end{align*}
\]

If they’re different

\[
LCS(X, Y) = LCS(X_{1..m-1}, Y_{1..n-1})
\]

1b: recursive solution

\[
\begin{align*}
X &= \text{A B C B D A B} \\
Y &= \text{B D C A B A}
\end{align*}
\]

If they’re different

\[
LCS(X, Y) = LCS(X_{1..m-1}, Y_{1..n-1})
\]

2: DP solution

\[
LCS(X, Y) = \begin{cases} 
1 + LCS(X_{1..m-1}, Y_{1..n-1}) & \text{if } x_m = y_n \\
\max(LCS(X_{1..m-1}, Y), LCS(X, Y_{1..n-1})) & \text{otherwise}
\end{cases}
\]

What types of subproblem solutions do we need to store?

LCS(X_{1..j}, Y_{1..k})

two different indices
2: DP solution

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

What types of subproblem solutions do we need to store?

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

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

Need to initialize values within 1 smaller in either dimension.

How should we fill in the table?
To fill in an entry, we may need to look:
- up one
- left one
- diagonal up and left

Just need to make sure these exist.
\[
LCS[i, j] = \begin{cases} 
1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\
\max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise}
\end{cases}
\]

Where's the final answer?
### The algorithm

**LCS(X, Y)**

1. \(m \leftarrow \text{length}(X)\)
2. \(n \leftarrow \text{length}(Y)\)
3. \(c(0,0) \leftarrow 0\)
4. for \(i \leftarrow 1 \text{ to } m\)
5. \(c(i,0) \leftarrow 0\)
6. for \(j \leftarrow 1 \text{ to } n\)
7. \(c(0,j) \leftarrow 0\)
8. for \(i \leftarrow 1 \text{ to } m\)
9. for \(j \leftarrow 1 \text{ to } n\)
10. \(c(i,j) \leftarrow \begin{cases} 
                    c(i-1,j-1) + 1 & \text{if } \Delta(x_i,y_j) = +1 \\
                    c(i-1,j) & \text{if } \Delta(x_i,y_j) = 0 \\
                    c(i,j-1) & \text{if } \Delta(x_i,y_j) = -1 \\
                    c(i,j) & \text{if } \Delta(x_i,y_j) = 0 \\
                   \end{cases} \)
11. return \(c(m,n)\)

Base case initialization:

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

LCS-LENGTH(X, Y)
1 m ← length[X]
2 n ← length[Y]
3 c(0,0) ← 0
4 for i ← 1 to m
5 for j ← 1 to n
6 if x_i = y_j
7 c(i,j) ← c(i-1, j-1) + 1
8 else
9 c(i,j) ← max(c(i-1, j), c(i, j-1))
10 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?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x_i</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>B</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LCS(ABCB, BDCABA)

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x_i</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>B</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Rod splitting**

**Input:** a length \( n \) and a table of prices for \( i = 1, 2, \ldots, m \)

**Output:** maximum revenue obtainable by cutting up the rod and selling the pieces

**Example:**

<table>
<thead>
<tr>
<th>length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

We can follow the arrows to generate the solution.
Rod splitting

Input: a length \( n \) and a table of prices for \( i = 1, 2, \ldots, m \)

Output: maximum revenue obtainable by cutting up the rod and selling the pieces

Example:

<table>
<thead>
<tr>
<th>length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

What is the best way if you have 13 units of rod?

1a: optimal substructure

Prove: optimal solutions to the problem incorporate optimal solutions to related subproblems

<table>
<thead>
<tr>
<th>length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

What would a solution look like?

What would a subproblem solution look like?

\( \{l_1, l_2, l_3, \ldots, l_m\} \) where \( \sum_{i=1}^{m} l_i \leq n \)
1a: optimal substructure

Prove: optimal solutions to the problem incorporate optimal solutions to related subproblems.

Assume: \( (l_1, l_2, l_3, ..., l_m) \) is a solution to \( n \), but \( (l_2, l_3, ..., l_m) \) is not a solution to \( n - l_1 \).

If that were the case, then some solution to \( n - l_1 \) exists where the sum of the prices of the lengths is greater than that for \( (l_2, l_3, ..., l_m) \).

We could add \( l_1 \) to this subproblem solution and get a better solution to the \( n \) problem... contradiction.

Proof by contradiction:

1b: recursive solution

What should be the first cut?

What are the options?

length | i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
price | \( p_i \) | 1 | 3 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 28

n

What should be the first cut?

How much is left?

length | i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
price | \( p_i \) | 1 | 3 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 28

n

n

cut 1

price 1
1b: recursive solution

<table>
<thead>
<tr>
<th>length</th>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>$p_i$</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

Which one should we choose?

Pretend like we have a solver (R) that gives us the answer to suproblems.

What would R take as input and return?

R(x) \rightarrow price for best set of cuts of length x

(coULD structure it with the actual cuts, but focusing on just the price is easier for now)
### 1b: Recursive Solution

<table>
<thead>
<tr>
<th>Length</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>17</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>24</td>
</tr>
<tr>
<td>10</td>
<td>28</td>
</tr>
</tbody>
</table>

What's the best we can do with this cut?

1 + \( R(n-1) \)

### 1b: Recursive Solution

<table>
<thead>
<tr>
<th>Length</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>17</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>24</td>
</tr>
<tr>
<td>10</td>
<td>28</td>
</tr>
</tbody>
</table>

What's the best we can do with this cut?

3 + \( R(n-2) \)
1b: recursive solution

What should be the first cut?

\[
R(n) = \max_{i \in n-1} \{ p_i + R(n - l_i) \}
\]

2: DP solution (from the bottom-up)

What are the smallest possible subproblems?

To calculate \( R(n) \), what are all the subproblems we need to calculate? This is the “table”.

How should we fill in the table?

Where will the answer be?

\[
R(n) = \max_{i \in n-1} \{ p_i + R(n - l_i) \}
\]

What are the smallest possible subproblems?

\( R(0) = 0 \), \( R(\cdot) \) not possible
To calculate $R(n)$, what are all the subproblems we need to calculate? This is the “table”. $R(0) \ldots R(n)$

Note: This is filling in a table for all possible integer lengths from 1 to $n$.

Where will the answer be? $R(n)$
2: DP solution

DP-Rod-Splitting(n)

r[0] = 0
for j = 1 to n
    max = 0
    for i = 1 to m
        if l[i] ≤ j
            p = p[i] + r[j - l[i]]
        if p > max
            max = p
    r[j] = max
return r[n]

Space requirements?
Running time?

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
1 if n = 1 or n = 2
2 return 1
3 else
4 return Fibonacci(n - 1) + Fibonacci(n - 2)

Fibonacci-Memoized(s)
1 \( f[i] \) = \( f[i] \)
2 if \( f[i] \) == \( \infty \)
3 for \( i \) = 2 to \( n \)
4 \( f[i] \) = \( f[i] \)
5 return Fib-Lookup(s)

Fib-Lookup(s)
1 if \( f[i] \) < \( \infty \)
2 return \( f[i] \)
3 \( x \) = \( \text{Fib-Lookup}(n - 1) + \text{Fib-Lookup}(n - 2) \)
4 if \( x \) < \( f[i] \)
5 \( f[i] \) = \( x \)
6 return \( f[i] \)

What else could we use besides an array? Use \( \infty \) to denote uncalculated

Memoized fibonacci

// Fibonacci
1 if n = 1 or n = 2
2 return 1
3 else
4 return Fibonacci(n - 1) + Fibonacci(n - 2)

Fibonacci-Memoized(s)
1 \( f[i] \) = \( f[i] \)
2 if \( f[i] \) == \( \infty \)
3 for \( i \) = 2 to \( n \)
4 \( f[i] \) = \( f[i] \)
5 return Fib-Lookup(s)

Fib-Lookup(s)
1 if \( f[i] \) < \( \infty \)
2 return \( f[i] \)
3 \( x \) = \( \text{Fib-Lookup}(n - 1) + \text{Fib-Lookup}(n - 2) \)
4 if \( x \) < \( f[i] \)
5 \( f[i] \) = \( x \)
6 return \( f[i] \)

Memoized fibonacci

// Fibonacci
1 if n = 1 or n = 2
2 return 1
3 else
4 return Fibonacci(n - 1) + Fibonacci(n - 2)

Fibonacci-Memoized(s)
1 \( f[i] \) = \( f[i] \)
2 if \( f[i] \) == \( \infty \)
3 for \( i \) = 2 to \( n \)
4 \( f[i] \) = \( f[i] \)
5 return Fib-Lookup(s)

Fib-Lookup(s)
1 if \( f[i] \) < \( \infty \)
2 return \( f[i] \)
3 \( x \) = \( \text{Fib-Lookup}(n - 1) + \text{Fib-Lookup}(n - 2) \)
4 if \( x \) < \( f[i] \)
5 \( f[i] \) = \( x \)
6 return \( f[i] \)

Memoized fibonacci

// Fibonacci
1 if n = 1 or n = 2
2 return 1
3 else
4 return Fibonacci(n - 1) + Fibonacci(n - 2)

Fibonacci-Memoized(s)
1 \( f[i] \) = \( f[i] \)
2 if \( f[i] \) == \( \infty \)
3 for \( i \) = 2 to \( n \)
4 \( f[i] \) = \( f[i] \)
5 return Fib-Lookup(s)

Fib-Lookup(s)
1 if \( f[i] \) < \( \infty \)
2 return \( f[i] \)
3 \( x \) = \( \text{Fib-Lookup}(n - 1) + \text{Fib-Lookup}(n - 2) \)
4 if \( x \) < \( f[i] \)
5 \( f[i] \) = \( x \)
6 return \( f[i] \)

Memoized fibonacci

// Fibonacci
1 if n = 1 or n = 2
2 return 1
3 else
4 return Fibonacci(n - 1) + Fibonacci(n - 2)

Fibonacci-Memoized(s)
1 \( f[i] \) = \( f[i] \)
2 if \( f[i] \) == \( \infty \)
3 for \( i \) = 2 to \( n \)
4 \( f[i] \) = \( f[i] \)
5 return Fib-Lookup(s)

Fib-Lookup(s)
1 if \( f[i] \) < \( \infty \)
2 return \( f[i] \)
3 \( x \) = \( \text{Fib-Lookup}(n - 1) + \text{Fib-Lookup}(n - 2) \)
4 if \( x \) < \( f[i] \)
5 \( f[i] \) = \( x \)
6 return \( f[i] \)
Memoized fibonacci

```
Memoized fibonacci(n)
1 if n == 1 or n == 2
2 return 1
3 else
4 return Memoized(n-1) + Memoized(n-2)
```

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)