Where did "dynamic programming" come from?

"I spent the fall quarter of 1958 at RAND. My first task was to find a name for an important decision process.

"An interesting question is, "Where did the name dynamic programming come from?"

"The RAND project was a real success for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was interested in solving the major problem of planning and management in the Air Force. He wanted a name that was keyworded -- a name that would be easily remembered and used..."" - Richard Bellman

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 $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_m$

What is the longest common subsequence?

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?

1b: recursive solution

$X = A\,B\,C\,B\,D\,A\,B$

$Y = B\,D\,C\,A\,B\,A$

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

Is the last character part of the LCS?

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

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

**X = A B C B D A B**

**Y = B D C A B A**

If they're different

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

---

2: DP solution

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

What types of subproblem solutions do we need to store?

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

two different indices

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

\[ \text{LCS}(X_{1..i}, Y_{1..j}) \]

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

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

<table>
<thead>
<tr>
<th>i</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>j</td>
<td>x_i</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

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.

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

\[
\begin{array}{ccccccc}
\hline
i & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
j & y_j & B & D & C & A & B & A \\
\hline
0 & x_i & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & A & 0 & & & & & \\
2 & B & 0 & & & & & \\
3 & C & 0 & & & & & ? \\
4 & B & 0 & & & & & - up one \\
5 & D & 0 & & & & & - left one \\
6 & A & 0 & & & & & - diagonal up and left \\
7 & B & 0 & & & & & Just need to make sure these exist \\
\hline
\end{array}
\]

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

\[
\begin{array}{ccccccc}
\hline
i & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
j & y_j & B & D & C & A & B & A \\
\hline
0 & x_i & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & A & 0 & ? & & & & \\
2 & B & 0 & & & & & \\
3 & C & 0 & & & & & \\
4 & B & 0 & & & & & \\
5 & D & 0 & & & & & \\
6 & A & 0 & & & & & \\
7 & B & 0 & & & & & \\
\hline
\end{array}
\]

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

\[
\begin{array}{ccccccc}
\hline
i & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
j & y_j & B & D & C & A & B & A \\
\hline
0 & x_i & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & A & 0 & 0 & & & & \\
2 & B & 0 & & & & & \\
3 & C & 0 & & & & & \\
4 & B & 0 & & & & & \\
5 & D & 0 & & & & & \\
6 & A & 0 & & & & & \\
7 & B & 0 & & & & & \\
\hline
\end{array}
\]

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

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

*Where's the final answer?*
### The Algorithm

**LCS-LENGTH(X, Y)**

1. \( m \leftarrow \text{length}(X) \)
2. \( n \leftarrow \text{length}(Y) \)
3. \( c[0, 0] \leftarrow 0 \)
4. for \( i \leftarrow 1 \) to \( m \)
5. \( c[i, 0] \leftarrow 0 \)
6. for \( j \leftarrow 1 \) to \( n \)
7. \( c[0, j] \leftarrow 0 \)
8. for \( i \leftarrow 1 \) to \( m \)
9. for \( j \leftarrow 1 \) to \( n \)
10. if \( x_i = y_j \)
11. \( c[i, j] \leftarrow 1 + c[i-1, j-1] \)
12. else \( c[i, j] \leftarrow \max(c[i-1, j], c[i, j-1]) \)
13. else
14. \( c[i, j] \leftarrow c[i, j-1] \)
15. return \( c[m, n] \)

---

### The Algorithm

**LCS-LENGTH(X, Y)**

1. \( m \leftarrow \text{length}(X) \)
2. \( n \leftarrow \text{length}(Y) \)
3. \( c[0, 0] \leftarrow 0 \)
4. for \( i \leftarrow 1 \) to \( m \)
5. \( c[i, 0] \leftarrow 0 \)
6. for \( j \leftarrow 1 \) to \( n \)
7. \( c[0, j] \leftarrow 0 \)
8. for \( i \leftarrow 1 \) to \( m \)
9. for \( j \leftarrow 1 \) to \( n \)
10. if \( x_i = y_j \)
11. \( c[i, j] \leftarrow 1 + c[i-1, j-1] \)
12. else \( c[i, j] \leftarrow \max(c[i-1, j], c[i, j-1]) \)
13. else
14. \( c[i, j] \leftarrow c[i, j-1] \)
15. return \( c[m, n] \)
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 c(i, 0) = 0
6 for j = 1 to n
7 c(0, j) = 0
8 for i = 1 to m
9 for j = 1 to n
10 if x_i = y_j
11 c(i, j) = 1 + c(i-1, j-1)
12 else if c(i-1, j) > c(i, j-1)
13 c(i, j) = c(i-1, j)
14 else if c(i, j-1) > c(i-1, j)
15 c(i, j) = c(i, j-1)
16 return c(m, n)

Fill in the matrix
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?

\[
LCS[i, j] = \begin{cases} 
1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\
\max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise}
\end{cases}
\]
How do we generate the solution from this?

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>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>30</td>
</tr>
</tbody>
</table>
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>30</td>
</tr>
</tbody>
</table>

What should be the first cut?
What are the options?

49

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>30</td>
</tr>
</tbody>
</table>

cut 1
price 1

50

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>30</td>
</tr>
</tbody>
</table>

cut 2
price 3

51

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>30</td>
</tr>
</tbody>
</table>

cut 1
price 1

52
1b: recursive solution

<table>
<thead>
<tr>
<th>length</th>
<th>i 1 2 3 4 5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>p_i 1 3 8 9 10 17 17 20 24 30</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) → 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)

cut 1 price 1

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

1 + R(n-1)
What’s the best we can do with this cut?

What should be the first cut?

\[
R(n) = \max_{2 \leq i \leq n} (p_i + R(n - i))
\]
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?

$$R(n) = \max_{i \in \mathbb{Z}^+} \{p_i + R(n - li)\}$$

Note: This is filling in a table for all possible integer lengths from 1 to $n$. The dependencies are on smaller values.
2: DP solution (from the bottom-up)

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

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 \( l_i \leq j \)

\[ p = p_i + r[j-l_i] \]

if \( p > \max \)

\[ \max = p \]

\[ r[j] = \max \]

return \( r[n] \)

---

Space requirements:
\( \Theta(n) \)

Running time:
\( \Theta(nm) \)

---

2: DP solution

DP-Rod-Splitting(n)

\[ r[0] = 0 \]

for \( j = 1 \) to \( n \)

\[ \max = 0 \]

for \( i = 1 \) to \( m \)

if \( l_i \leq j \)

\[ p = p_i + r[j-l_i] \]

if \( p > \max \)

\[ \max = p \]

\[ r[j] = \max \]

return \( r[n] \)

---

Space requirements: \( \Theta(n) \)

Running time: \( \Theta(nm) \)
0-1 Knapsack problem

0-1 Knapsack – A thief robbing a store finds m items worth $v_1, v_2, \ldots, v_m$ dollars and weight $w_1, w_2, \ldots, w_m$ 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 they want to maximize value?

Repetition is allowed, that is you can take multiple copies of any item.

1b: recursive solution

\[ K(w) = \max_{i=1, \ldots, m} \{ K(w - w_i) + v_i \} \]

2: DP solution (from the bottom-up)

\[ K(w) = \max_{i=1, \ldots, m} \{ K(w - w_i) + v_i \} \]

What are the smallest possible subproblems?

$K(0) = 0$

To calculate $K(w)$, what are all the subproblems we need to calculate? This is the “table”: $K(0) \ldots K(W)$

How should we fill in the table? $K(1) \rightarrow K(W)$

Where will the answer be? $K(W)$

3: Analysis

\[ K(w) = \max_{i=1, \ldots, m} \{ K(w - w_i) + v_i \} \]

What are the smallest possible subproblems? $K(0) = 0$

To calculate $K(w)$, what are all the subproblems we need to calculate? This is the “table”: $K(0) \ldots K(W)$

How should we fill in the table? $K(0) \rightarrow K(W)$

Where will the answer be? $K(W)$

Space requirements: $\Theta(W)$

Running time: $\Theta(Wm)$
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(s)
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[0] = 1
2 f[1] = 1
3 for i = 2 to n
4 f[i] = ∞
5 return Fibonacci-Memoized(n)

// Fib-Lookup(n)
1 if f[i] < ∞
2 return f[i]
3 x = Fib-Lookup(n - 1) + 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 to denote uncalculated
Memoized fibonacci

```python
Fibonacci(n):
    if n = 1 or n = 2
        return 1
    else
        return Fibonacci(n - 1) + Fibonacci(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)