

Admin Assignment 6

Where did "dynamic programming" come from?

Where did "dynamic programming"

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes.
"An interesting question is, "Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. The not using the term, leghtly: I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in this was dynamic, in the classical physical sense. It also has a very interesting property as an adjective meaning. It's impossible to use the word, dynamic, in a peiporative sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a peiporative sense. It y thinking of some commission that will possible give it a peiporative meaning. It's impossible to use the word, dynamic, in a peiporative sense. It y thinking of some commission that will possible give it a peiporative meaning. It's impossible to use the word, dynamic, in a peiporative sense. It y thinking of some commission that will possible give it a peiporative meaning. It's impossible to use the word, dynamic to the Air Congressman could object to. So I used it as an umbrella for my activities" (p. 159).

Richard Bellman On the Birth of Dynamic Programming

Stuart Dreyfus

http://www.eng.tau.ac.il/~ami/cd/or50/1526-5463-2002-50-01-0048.pdf

Dynamic programming

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

2

the subproblems are overlapping

Dynamic programming: steps

1 a) 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:

- $\hfill \square$ 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 $X = x_1, x_2, ..., x_n$ and

 $Y = y_1, y_2, ..., y_n$

What is the longest common subsequence?

5

6

2: DP solution

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

What types of subproblem solutions do we need to store?

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

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

LCS[i,j] = -	$\begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1] & \text{otherwise} \end{cases}$
	0 1 2 3 4 5 6 y _j B D C A B A
1 A	0 0

7 8

LCS[i,j] = -	$\begin{cases} 1 + LCS[i-1, j-1] \\ \max(LCS[i-1, j], LCS[i-1, j], LCS[i-1, j], LCS[i-1, j] \end{cases}$	$if x_i = y_j$ $i, j-1] otherwise$
j _i	0 1 2 3 4 5 6 y _j B D C A B A	
1 A 2 B 3 C	0 ? 0	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
9		

LCS[i,j] =	$\begin{cases} 1 + LCS[i-1, j-1] \\ \max(LCS[i-1, j], LCS[i, j-1]) \end{cases}$	$if x_i = y_j$ $j-1] otherwise$
j <u>i</u>	0 1 2 3 4 5 6 y _j B D C A B A	
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	LCS(A, B)
10		

$LCS[i,j] = \langle$	$\begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1] & \text{otherwise} \end{cases}$
j i	0 1 2 3 4 5 6 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	0 0 0 0

LCS[i,j] =	$\begin{bmatrix} 1 + LCS[i-1, j-1] \\ \max(LCS[i-1, j], LCS[i, j-1] \end{bmatrix}$	$if x_i = y_j$ otherwise
j _i	0 1 2 3 4 5 6 y _j B D C A B A	
2 B 3 C 4 B 5 D	0 0 0 0 0 0 0 0 0 0 0 ? 0 0	LCS(A, BDCA)
6 A 7 B	0 0	

$LCS[i,j] = \cdot$	$\begin{cases} 1 + LCS[i-1, j-1] \\ \max(LCS[i-1, j], LCS[i, j-1] \end{cases}$	$if x_i = y_j$ otherwise
j i	0 1 2 3 4 5 6 y _j B D C A B A	
1 A 2 B 3 C		LCS(A, BDCA)

LCS[i, j] =	$\begin{cases} 1 + LCS[i-1, j-1] \\ \max(LCS[i-1, j], LCS[i, j], LCS[i, j], LCS[i, j] \end{cases}$	
j i 0 x _i 1 A 2 B 3 C 4 B 5 D 6 A 7 B	0 1 2 3 4 5 6 y _i B D C A B A 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 2 2 0 1 1 2 2 2 2 0 1 1 2 2 ? 0 0 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 <u>i</u>	0 1 2 3 4 5 6 y _j B D C A B A		
3 C 4 B 5 D	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1		
7 B	0		

LCS[i,j] =	$\begin{cases} 1 + LCS[i-1, j-1] \\ \max(LCS[i-1, j], LCS[i, j]) \end{cases}$	$if x_i = y_j$ $j-1] otherwise$
j _ i	0 1 2 3 4 5 6 y _j B D C A B A	
0 x _i 1 A 2 B	0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 2 2	Where's the final answer?
3 C 4 B 5 D	0 1 1 222 2 0 1 1 223 3 0 1 2 223 3	
6 A 7 B	0 1 2 2 3 3 4 0 1 2 2 3 4 4	

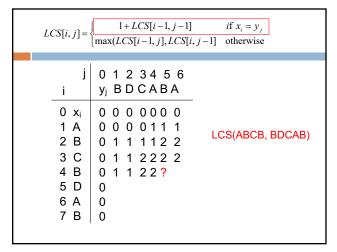
15 16

$LCS[i, j] = \begin{cases} 1 + LC \\ \max(LCS[i]) \end{cases}$
$ \begin{array}{c ccccccccccccccccccccccccccccccccccc$

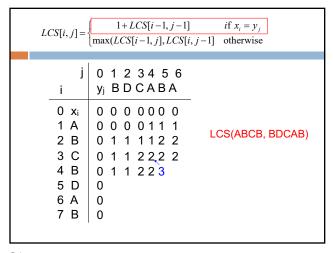
LCS	$[i,j] = \begin{cases} 1 + LCS[i-1] \\ \max(LCS[i-1,j], \end{cases}$	$[i, j-1]$ if $x_i = y_j$ LCS[i, j-1] otherwise
	0 1 2 3 4 5 6 y _j BDCABA	Space requirements: ⊖(nm)
1 A 2 B 3 C 4 B 5 D 6 A	0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 2 2 0 1 1 2 2 2 2 0 1 1 2 2 3 3 0 1 2 2 2 3 3 0 1 2 2 3 3 4 0 1 2 2 3 4 4	Running time: ⊖(nm)

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

What if we wanted to know the actual sequence?



19 20



1 + LCS[i-1,j-1] $if x_i = y_j$ $LCS[i,j] = \begin{cases} 1 + LCS[i-1,j], LCS[i,j-1] & \text{otherwise} \end{cases}$ 0 1 2 3 4 5 6 $y_j BDCABA$ 0 0 0 0 0 0 $0 x_i$ 1 A 0 0 0 0 1 1 1 LCS(ABCB, BDCABA) 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

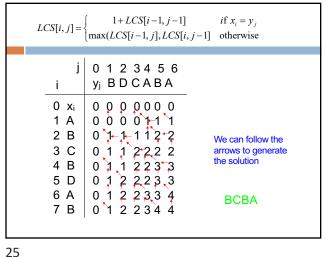
22

21

LCS[i,j] =	$\begin{cases} 1 + LCS[i-1, j-1] \\ \max(LCS[i-1, j], LCS[i-1, j], LCS[i-1, j], LCS[i-1, j], LCS[i-1, j], LCS[i-1, j], LCS[i-1, j-1] \end{cases}$	
j i 0 x _i 1 A 2 B 3 C 4 B 5 D 6 A 7 B	· · · = = =	LCS(ABCB, BDCABA)

1 + LCS[i-1, j-1]LCS[i,j] = $\max(LCS[i-1,j],LCS[i,j-1]$ otherwise 0 1 2 3 4 5 6 $y_j BDCABA$ i $0 x_i$ 000000 0 0 0 0 1 1 1 1 A 0 1-1-112-2 0 1 1 2-2 2 2 B How do we 3 C generate the 4 B 0 1 1 2 2 3 3 solution from this? 5 D 0 1 2 2 2 3 3 0 1 2 2 3 3 4 6 A 7 B 0 1 2 2 3 4 4

23 24



Rod splitting

Input: a length n and a table of prices for i = 1, 2, ... mOutput: maximum revenue obtainable by cutting up the rod and selling the pieces

Example:

26

length i 1 2 3 4 5 price p_i 1 5 8 9 10 17 17 20 24 30

1 a: optimal substructure

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

length i 1 2 3 4 5 price p_i 1 5 8 9 10 17 17 20 24 30

What would a solution look like?

1a: optimal substructure

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

length i 1 2 3 4 5 6 7 price p_i 1 5 8 9 10 17 17 20 24 30

 $\{l_1, l_2, l_3, \dots, l_m\}$ where $\sum_{i=1}^{m} l_i \leq n$

What would a subproblem solution look like?

27 28

1 a: optimal substructure

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

$$\{l_1, l_2, l_3, \dots, l_m\}$$
 where $\sum_{i=1}^{m} l_i \leq n$

$$\{l_2, l_3, \dots, l_m\}$$
 where $\sum_{i=2}^m l_i \le n - l_1$

1a: optimal substructure

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

Proof by contradiction:

Assume: $\{l_1,l_2,l_3,\dots,l_m\}$ is a solution to n, but $\{l_2,l_3,\dots,l_m\}$ is **not** a solution to $n-l_1$

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

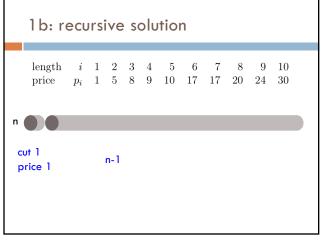
We could add l_1 to this subproblem solution and get a better solution to the n problem... contradiction

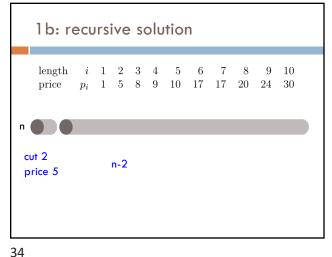
29

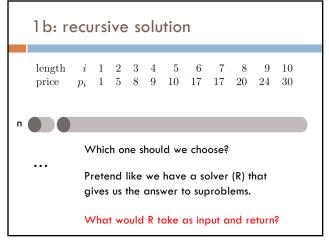
30

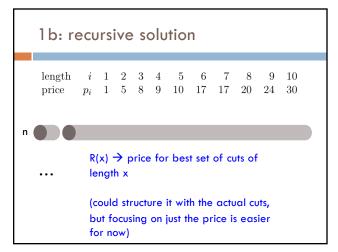
length i 1 2 3 4 5 6 7 8 9 10 price p_i 1 5 8 9 10 17 17 20 24 30 What should be the first cut? What are the options?

31 32

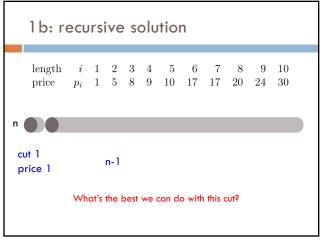


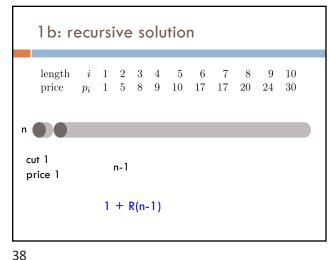


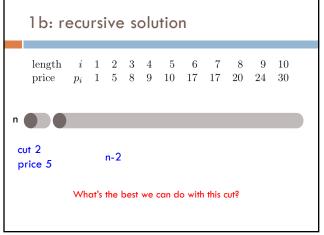


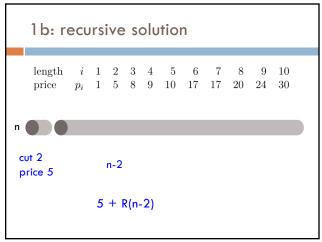


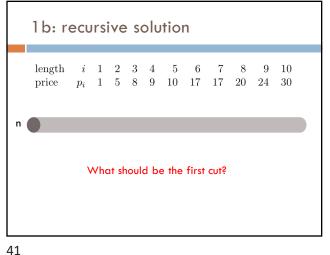
35 36

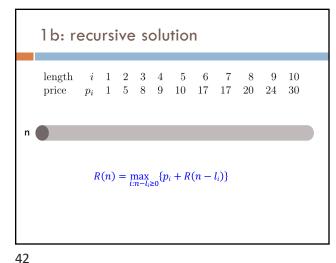












2: DP solution (from the bottom-up)

$$R(n) = \max_{i:n-l_i \ge 0} \{p_i + R(n-l_i)\}$$

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?

2: DP solution (from the bottom-up)

$$R(n) = \max_{i:n-l_i \ge 0} \{p_i + R(n-l_i)\}\$$

What are the smallest possible subproblems? R(0) = 0, R(-i) not possible

2: DP solution (from the bottom-up)

$$R(n) = \max_{i:n-l_i \ge 0} \{p_i + R(n-l_i)\}\$$

To calculate R(n), what are all the subproblems we need to calculate? This is the "table". R(0)...R(n)

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

2: DP solution (from the bottom-up)

$$R(n) = \max_{i:n-l_i \ge 0} \{p_i + R(n-l_i)\}\$$

How should we fill in the table? $R(0) \rightarrow R(n)$

The dependencies are on smaller values

45

46

48

2: DP solution (from the bottom-up)

```
R(n) = \max_{i:n-l_i \ge 0} \{p_i + R(n-l_i)\}
```

Where will the answer be? R(n)

2: DP solution

```
\begin{aligned} & \text{DP-Rod-Splitting(n)} \\ & r[0] = 0 \\ & \text{for } j = 1 \, to \, n \\ & \text{max} = 0 \\ & \text{for } i = 1 \, to \, m \\ & \text{if } l_i \leq j \\ & p = p_i + r[j - l_i] \\ & \text{if } p > \text{max} \\ & \text{max} = p \end{aligned}
r[j] = \text{max}
```

47

3: Analysis

```
\begin{aligned} & \text{DP-Rod-Splitting(n)} \\ & r[0] = 0 \\ & \text{for } j = 1 \, to \, n \\ & \text{max} = 0 \\ & \text{for } i = 1 \, to \, m \\ & \text{if } l_i \leq j \\ & p = pi + r[j - li] \\ & \text{if } p > \text{max} \\ & \text{max} = p \end{aligned} \text{Running time?} \text{return } r[n]
```

3: Analysis

50

```
\begin{array}{lll} \operatorname{DP-Rod-Splitting}(\mathbf{n}) & & & & \\ r[0] = 0 & & & & \\ \operatorname{for} j = 1 \operatorname{to} n & & & \\ \operatorname{max} = 0 & & & \\ \operatorname{for} i = 1 \operatorname{to} m & & & \\ \operatorname{if} l_i \leq j & & & \\ p = pi + r[j - li] & & \\ \operatorname{if} p > \operatorname{max} & & \\ \operatorname{max} = p & \\ \\ r[j] = \operatorname{max} & & \\ \end{array}
```

49

0-1 Knapsack problem

0-1 Knapsack – A thief robbing a store finds m items worth $v_1, v_2, ..., v_m$ dollars and weight

 $w_1, w_2, ..., w_m$ 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

la: optimal substructure

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

Proof by contradiction:

Assume: $\{i_1,i_2,i_3,\dots,i_k\}$ is a solution to W but $\{i_2,i_3,\dots,i_k\}$ is **not** a solution to W - W_{i_1}

Then some solution to $W-w_{i_1}$ exists, $\{i'_{2},i'_{3},\ldots,i'_{n}\}$ where the sum of the values of the items is greater than for $\{i_{2},i_{3},\ldots,ik\}$

We could create a solution $\{i_1,i'2,i'3,\dots,i'_n\}$ to the original problem that has more value... contradiction

51 52

1b: recursive solution

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

2: DP solution (from the bottom-up)

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

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) ... K(W)

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

Where will the answer be? K(W)

53

54

3: Analysis

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

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) \dots K(W)

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

Where will the answer be? K(W)

Space requirements: ⊖(W)

Running time: ⊖(Wm)

Memoization

Sometimes it can be a challenge to write the function in a bottom-up fashion $% \label{eq:controller}%$

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)

55

```
\begin{tabular}{l|l} \textbf{Memoized fibonacci} \\ \hline & Fibonacci(n) \\ \hline & 1 & \textbf{if } n=1 \ \text{or } n=2 \\ \hline & 2 & \textbf{return } 1 \\ \hline & 3 & \textbf{else} \\ \hline & & & \textbf{return Fibonacci}(n-1) + \textbf{Fibonacci}(n-2) \\ \hline & & \\ \hline
```

```
Memoized fibonacci
              FIBONACCI(n)
              1 \quad \text{if } n=1 \text{ or } n=2 \\
              \begin{array}{ccc} & \text{of } n=2\\ 2 & \text{return 1}\\ 3 & \text{else} \end{array}
                          return Fibonacci(n-1) + Fibonacci(n-2)
                                                              What else could we use
              Fibonacci-Memoized(n)
                                                             besides an array?
              1 \quad fib[1] \leftarrow 1
             Use ∞ to denote
                                                                                uncalculated
              Fib-Lookup(n)
              1 \quad \text{if } fib[n] < \infty
              \begin{array}{ccc} & & \text{return } fib[n] \\ 2 & & \text{return } fib[n] \\ 3 & x \leftarrow \text{Fib-Lookup}(n-1) + \text{Fib-Lookup}(n-2) \end{array}
              6 return fib[n]
```

```
Memoized fibonacci
                 FIBONACCI(n)
                 1 \quad \text{if } n=1 \text{ or } n=2 \\
                               return 1
                 3 else
                                return Fibonacci(n-1) + Fibonacci(n-2)
                 Fibonacci-Memoized(n)
                 1 \quad fib[1] \leftarrow 1
                 2 \quad fib[2] \leftarrow 1
                 3 for i \leftarrow 3 to n
                        fib[i] \leftarrow \infty
                 5 return Fib-Lookup(n)
                 Fib-Lookup(n)
                                                                                      Check if we already
                 \begin{array}{c|c} 1 & \text{if } fib[n] < \infty \\ 2 & \text{return } fib[n] \\ \hline 3 & x \leftarrow \text{Fib-Lookup}(n-1) + \text{Fib-Lookup}(n-2) \\ \end{array} 
                                                                                      calculated the value
                  \begin{array}{ccc} 4 & \textbf{if} & x < fib[n] \\ 5 & & fib[n] \leftarrow x \end{array} 
                 6 return fib[n]
```

59 60

Memoization

Pros

- $lue{}$ 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)