

COMP2013

Data Structures and Algorithms

Lecture 10

Dynamic Programming

Dr. Jieming Shi

Rod cutting problem



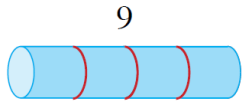
- Cut long steel rods into shorter rods and sell (cut is free)
- For different length, i inches, the price p_i is different as shown below
- Given a rod of length n inches and the price table, for $i = 1, \dots, n$,
- determine the max revenue r_n by cutting the rod and selling the pieces.
(An optimization problem)
 - You can cut 0 or many times.

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

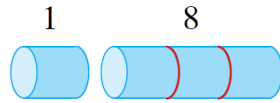
Toy example $n=4$



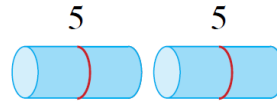
- Exhaust all possible cuts on $n=4$ inches rod



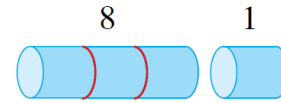
(a)



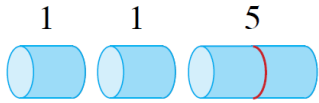
(b)



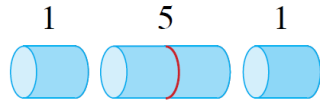
(c)



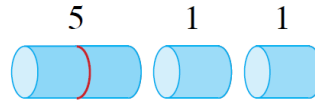
(d)



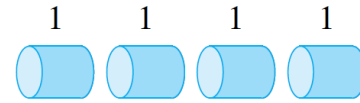
(e)



(f)



(g)



(h)

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

The solution space for n -inch rod



- At every inch from $1, \dots, n - 1$, you can independently have a choice to *cut* or *not cut*
- How many different choices in total for a rod of n inches?
 - 2^{n-1}
 - Expensive to exhaust all possible ways

Notations in the rod cutting problem



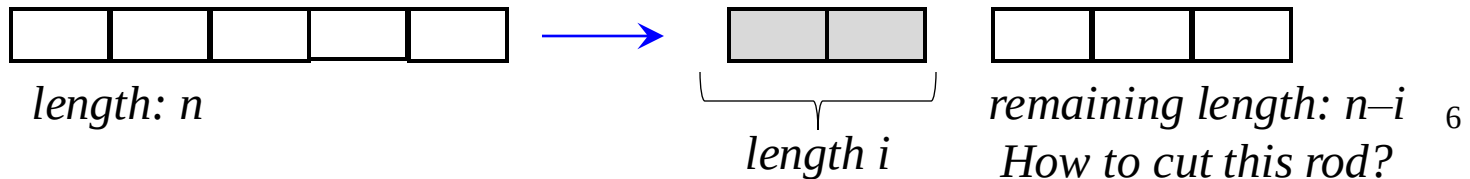
- An optimal decomposition:
 - $n = i_1 + i_2 + \dots + i_k$
 - Cut a length- n rod into k pieces, each with length $i_j, j = 1, \dots, k$:
 - $7=2+2+3$
 - Cut a length-7 rod into 3 pieces, each with length 2, 2, 3
- The *optimal* revenue of the decomposition
 - $r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}$
 - $r_7=5+5+8=18$

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

Characterize the optimization problem



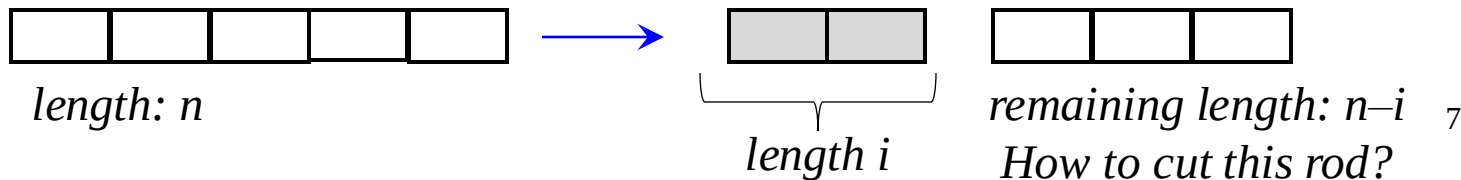
- Cut a length- n rod into two pieces with i and $n-i$ inch
 - Length- i piece will not cut anymore: price p_i
 - r_{n-i} is the optimal revenue of length $n-i$ (*a subproblem*)
 - For length- n rod: $p_i + r_{n-i}$
 - Is this the optimal revenue for length- n rod?
 - Maybe not



Recursive formulation



- Cut a length- n rod into two pieces with i and $n-i$ inch
 - Length- i piece will not cut anymore: price p_i
 - r_{n-i} is the optimal revenue of length $n-i$ (*a subproblem*)
 - For length- n rod: $p_i + r_{n-i}$
 - Is this the optimal revenue for length- n rod?
 - Maybe not
- The recursive optimal solution $r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\}$



Slow recursive solver



- The recursive optimal solution $r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\}$
- Is this efficient?
 - **No**
 - **Repeated subproblems**

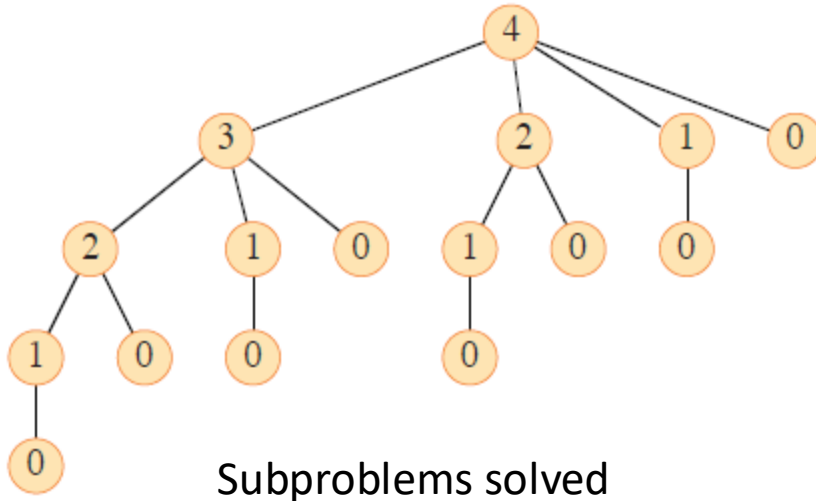
CUT-ROD(p, n)

```
1  if  $n == 0$ 
2      return 0
3   $q = -\infty$ 
4  for  $i = 1$  to  $n$ 
5       $q = \max\{q, p[i] + \text{CUT-ROD}(p, n - i)\}$ 
6  return  $q$ 
```


Slow recursive solver



- The recursive optimal solution $r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\}$
- Is this efficient?
 - **No**
 - **Repeated subproblems $O(2^n)$**



CUT-ROD(p, n)

```
1  if  $n == 0$ 
2      return 0
3   $q = -\infty$ 
4  for  $i = 1$  to  $n$ 
5       $q = \max\{q, p[i] + \text{CUT-ROD}(p, n - i)\}$ 
6  return  $q$ 
```

Dynamic programming (DP)



- Avoid solving the same subproblems repeatedly
- Solve each subproblem *only once*
 - Save the solution of the subproblem
 - Reuse the solution later when the subproblem appears again
 - Rather than recomputing
 - Use additional memory to store solutions
- Time-memory trade-off
- DP for the rod-cutting problem
 - $O(n^2)$
- *Bottom-up DP*
- Top-down DP

Bottom-up dynamic programming



- “size” of a subproblem
 - E.g., the length of a rod to cut
- Solve subproblems in size order, smallest first, storing the solution to each subproblem when it is first solved
 - In the rod-cutting problem: a subproblem of size i is smaller than the subproblem of size j , if $i < j$
- Thus, the DP procedure solves subproblems of sizes $i = 0, 1, \dots, n$, in that order
- When solving a problem of size i , there are already solutions for the smaller prerequisite problems of size $< i$

Bottom-up dynamic programming



- From small to large problems, non-recursive

BOTTOM-UP-CUT-ROD(p, n)

```
1  let  $r[0:n]$  be a new array      // will remember solution values in  $r$ 
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$                 // for increasing rod length  $j$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$             //  $i$  is the position of the first cut
6           $q = \max\{q, p[i] + r[j - i]\}$ 
7       $r[j] = q$                   // remember the solution value for length  $j$ 
8  return  $r[n]$ 
```

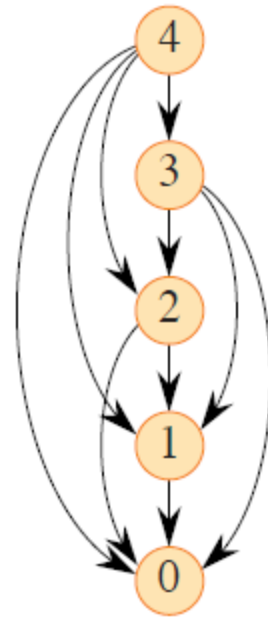
i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$		1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1	5	8	10	13	17	18	22	25	30

Solution of subproblems

Subproblem graphs: Interpret relationship of subproblems



- Rod length $n = 4$
- Bottom-up from small to large subproblems
- You solve the smaller subproblems y adjacent to a given subproblem x before you solve subproblem x



Construct a solution



- So far given a length- n rod, we only get the max revenue we can get by cutting it
- How to cut it? How many pieces to cut?

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$		1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Construct a solution



EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
1  let  $r[0:n]$  and  $s[1:n]$  be new arrays
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$            // for increasing rod length  $j$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$        //  $i$  is the position of the first cut
6          if  $q < p[i] + r[j - i]$ 
7               $q = p[i] + r[j - i]$ 
8               $s[j] = i$      // best cut location so far for length  $j$ 
9       $r[j] = q$              // remember the solution value for length  $j$ 
10 return  $r$  and  $s$ 
```

$s[j]$ remember the best cut location for length j

Print-Cut-Rod-Solution($p, 10$)

Output is 10

Print-Cut-Rod-Solution($p, 7$)

Output is 1, 6

PRINT-CUT-ROD-SOLUTION(p, n)

```
1  ( $r, s$ ) = EXTENDED-BOTTOM-UP-CUT-ROD( $p, n$ )
2  while  $n > 0$ 
3      print  $s[n]$            // cut location for length  $n$ 
4       $n = n - s[n]$          // length of the remainder of the rod
```

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$		1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10



- For *optimization* problems: minimize or maximize something
- DP applies when the subproblems overlap
 - Reduce the repeated computation of same subproblem
 - Solve each subproblem only once and store the answer for repeated usage

Two key ingredients



- Two key ingredients that an optimization problem must have in order for dynamic programming to apply:
 - **Optimal substructure**
 - if an optimal solution to the problem contains within its optimal solutions to subproblems.
 - **Overlapping subproblems.**
- Dynamic programming builds an optimal solution to the problem from optimal solutions to subproblems.

Two key ingredients



- *Optimal substructure varies across problem domains*
- how many subproblems an optimal solution to the original problem uses
 - In the rod-cutting problem, one subproblem of size $n - i$ for the problem of size n
- how many choices you have in determining which subproblem(s) to use in an optimal solution
 - n choices for i to find the subproblem yielding an optimal solution

Two key ingredients



- Overlapping subproblems:
 - The same subproblem solved many times

Four steps in DP

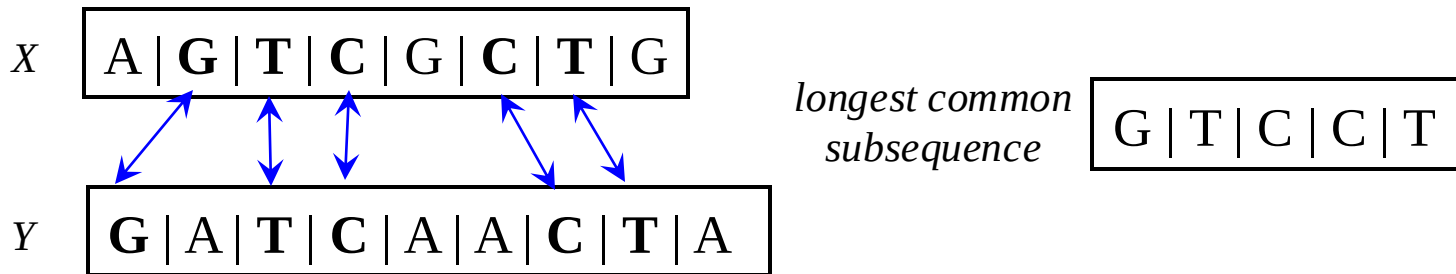


- 1. Characterize the structure of an optimal solution.
- 2. Recursively define the value of an optimal solution.
 - (But not solve it by recursion if you use bottom-up DP)
- 3. Compute the value of an optimal solution, typically in a bottom-up fashion.
- 4. Construct an optimal solution from computed information.

Longest common subsequence (LCS)



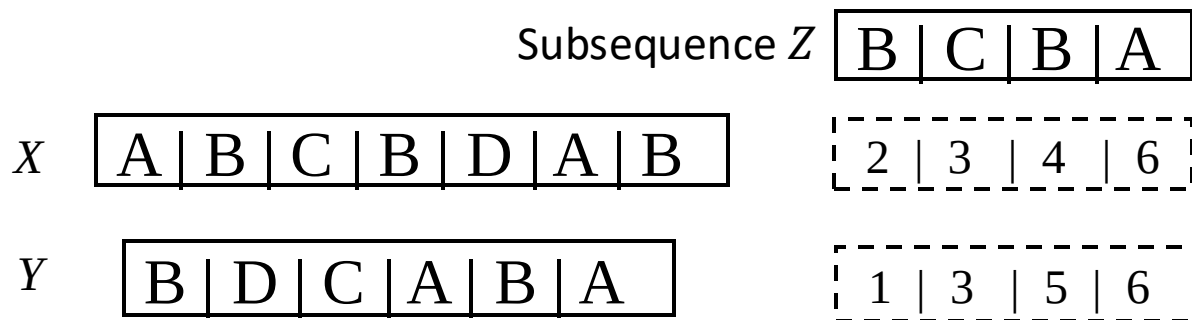
- Problem background
 - A string is a sequence of characters, can be expressed as an array $X = \langle x_1, x_2, \dots, x_m \rangle$
 - Application: Given DNA strings X and Y , how do we measure the similarity between them?
 - Goal: Find the **longest** “common subsequence” between X and Y
 - The common characters must appear in the same order, but not necessarily consecutive
 - The longer the more similar



LCS problem



- Given sequences $X = \langle x_1, x_2, \dots, x_m \rangle$, $Y = \langle y_1, y_2, \dots, y_n \rangle$
- $Z = \langle z_1, z_2, \dots, z_k \rangle$ is a **subsequence** of X if there exists an increasing integer sequence $\langle i_1, i_2, \dots, i_k \rangle$ such that $x_{i_j} = z_j$ for all $j = 1, 2, \dots, k$
- **Longest common subsequence (LCS)**
 - Find a *common* subsequence of both X and Y
 - The subsequence has the *maximum* length



◆ Which one is LCS?

◆ $\langle B, C, A \rangle$

◆ $\langle B, C, B, A \rangle$

◆ $\langle B, D, A, B \rangle$

A brute-force approach



- Enumerate all subsequences of X , and check if each subsequence is also a subsequence of Y , to find LCS
- How many subsequences exist in X with length m ?
 - 2^m (This is just for X , similar for Y)
- Resort to dynamic programming
 - *Problem size* is related to the number of characters in sequences

Step 1: Characterize the optimal substructure



- (We are now trying to formulate subproblems)
- Given the sequences $X = \langle x_1, x_2, \dots, x_m \rangle$, $Y = \langle y_1, y_2, \dots, y_n \rangle$
 - X and Y are also denoted as X_m and Y_n
- Assume $\mathbf{Z} = \langle z_1, z_2, \dots, z_k \rangle$ be the **LCS** of X and Y
 - $\mathbf{Z}$ is also denoted as $\mathbf{Z}_k$
- What can the last characters in X and Y tell you?
 - Consider these cases:
 - Case 1. $x_m = y_n$
 - Case 2. $x_m \neq y_n$ and $x_m \neq z_k$
 - Case 3. $x_m \neq y_n$ and $y_n \neq z_k$

Step 1: Characterize the optimal substructure



- Case 1. If $x_m = y_n$
 - The last character in Z (i.e., z_k) must be same as x_m and y_n
 - *Subproblem:*
 - $Z_{k-1} = \langle z_1, z_2, \dots, z_{k-1} \rangle$ is an LCS of the sequences $X_{m-1} = \langle x_1, x_2, \dots, x_{m-1} \rangle$, $Y_{n-1} = \langle y_1, y_2, \dots, y_{n-1} \rangle$
 - If we can find LCS Z_{k-1} for X_{m-1} and Y_{n-1} , we just append z_k (e.g., A) to get LCS Z_k for X_m and Y_n

X

...	A
-----	---

Y

...	A
-----	---

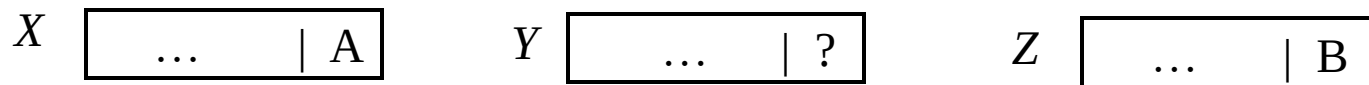
Z

...	A
-----	---

Step 1: Characterize the optimal substructure



- Case 2. $x_m \neq y_n$ and $x_m \neq z_k$
- The last character in X , x_m , does not contribute to the last character of Z , z_k
- *Subproblem* (How to make Z the longest possible?):
 - $Z_k = \langle z_1, z_2, \dots, z_k \rangle$ is an LCS of the remaining sequences $X_{m-1} = \langle x_1, x_2, \dots, x_{m-1} \rangle$,
 $Y = \langle y_1, y_2, \dots, y_n \rangle$



- Case 3. $x_m \neq y_n$ and $y_n \neq z_k$
- The last character in Y , y_n , does not contribute to the last character of Z , z_k
- *Subproblem*
 - $Z_k = \langle z_1, z_2, \dots, z_k \rangle$ is an LCS of the remaining sequences $X = \langle x_1, x_2, \dots, x_m \rangle$,
 $Y_{n-1} = \langle y_1, y_2, \dots, y_{n-1} \rangle$



Step 2. Recursive optimal solution



- Step 1 implies that we can examine subproblems to find the LCS of X_n and Y_n
- Step 2:
 - For X_i and Y_j , with length i and j respectively ($i = 0, 1, \dots, m$, and $j = 0, 1, \dots, n$)
 - If x_i and y_j are the same,
 - Find the LCS of X_{i-1} and Y_{j-1} and append x_i to get the LCS of X_i and Y_j
 - If x_i and y_j are different,
 - Find the LCS_1 of X_i and Y_{j-1}
 - Find the LCS_2 of X_{i-1} and Y_j
 - The longer sequence between LCS_1 and LCS_2 is the LCS of X_i and Y_j

Step 2. Recursive optimal solution



- We describe the LCS problem in terms of the length of LCS
- Let $c[i, j]$ (scalar) be the *length* of LCS between X_i and Y_j , where $i = 0, 1, \dots, m$, and $j = 0, 1, \dots, n$

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max \{c[i, j - 1], c[i - 1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Step 3: Computing the length of an LCS



- Based on the recursive formulation in Step 2, we could trivially write a recursive algorithm
 - Inefficient since a subproblem may be computed repeatedly
 - You can analyze the running time by solving the recurrence

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max \{c[i, j - 1], c[i - 1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Step 3: Computing the length of an LCS



- Store the solution of all possible solutions
 - $c[i, j]$
 - $i = 0, 1, \dots, m$, and $j = 0, 1, \dots, n$
 - What data structure? (*Using DP, you need to design proper structure to store solutions*)
- Store LCS length $c[i, j]$ in **row-major order**, in $m \times n$ matrix (table)
- $b[1:m, 1:n]$ matrix to help in constructing an optimal solution (For Step 4)

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max \{c[i, j - 1], c[i - 1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Step 3: Computing the length of an LCS



- The figure shows two matrices
 - $c[i, j]$: LCS length (numbers)
 - $b[i, j]$: indicator to construct LCS (arrows)
- Solve problems in a *bottom-up* way
 - From *small-size* problems to *large-size* problems (i.e., from smaller i and j to larger i and j)
 - Each subproblem is computed only once
 - Construct b and c matrices

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

Step 3: Computing the length of an LCS



- $c[i, j]$: the numbers
- $b[i, j]$: the arrows

LCS-LENGTH(X, Y, m, n)

```

1  let  $b[1 : m, 1 : n]$  and  $c[0 : m, 0 : n]$  be new tables
2  for  $i = 1$  to  $m$ 
3       $c[i, 0] = 0$ 
4  for  $j = 0$  to  $n$ 
5       $c[0, j] = 0$ 
6  for  $i = 1$  to  $m$            // compute table entries in row-major order
7      for  $j = 1$  to  $n$ 
8          if  $x_i == y_j$ 
9               $c[i, j] = c[i - 1, j - 1] + 1$       Case 1
10              $b[i, j] = \nwarrow$ 
11          elseif  $c[i - 1, j] \geq c[i, j - 1]$       Case 2 and 3
12               $c[i, j] = c[i - 1, j]$ 
13               $b[i, j] = \uparrow$ 
14          else  $c[i, j] = c[i, j - 1]$ 
15               $b[i, j] = \leftarrow$ 
16  return  $c$  and  $b$ 
    
```

In $O(mn)$

		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
i	0	x_i							
	1	A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
	2	B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
	3	C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
	4	B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
	5	D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
	6	A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
	7	B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

Step 4: Construct an LCS



- $c[i, j]$: the numbers
- $b[i, j]$: the arrows

In $O(m+n)$

PRINT-LCS(b, X, i, j)

```

1  if  $i == 0$  or  $j == 0$ 
2      return // the LCS has length 0
3  if  $b[i, j] == \nwarrow$ 
4      PRINT-LCS( $b, X, i - 1, j - 1$ )
5      print  $x_i$  // same as  $y_j$ 
6  elseif  $b[i, j] == \uparrow$ 
7      PRINT-LCS( $b, X, i - 1, j$ )
8  else PRINT-LCS( $b, X, i, j - 1$ )
    
```

		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
i	x_i								
0			0	0	0	0	0	0	0
1	A		0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2	B		0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3	C		0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4	B		0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5	D		0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6	A		0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7	B		0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4



- For *optimization* problems: minimize or maximize something
- DP applies when the subproblems overlap
 - Reduce the repeated computation of same subproblem
 - Solve each subproblem only once and store the answer for repeated usage

Two key ingredients



- Two key ingredients that an optimization problem must have in order for dynamic programming to apply:
 - **Optimal substructure**
 - if an optimal solution to the problem contains within its optimal solutions to subproblems.
 - **Overlapping subproblems.**
- Dynamic programming builds an optimal solution to the problem from optimal solutions to subproblems.



- 1. Characterize the structure of an optimal solution.
- 2. Recursively define the value of an optimal solution.
 - (But not solve it by recursion)
- 3. Compute the value of an optimal solution, typically in a bottom-up fashion.
- 4. Construct an optimal solution from computed information.

A methodology, not a specific method

Thank You

