

Principles and Practice of Problem Solving: Lecture 13-Dynamic Programming

Lecturer: Haiming Jin

Outline

- Dynamic Programming
 - Motivation
 - Example: Matrix-Chain Multiplication
 - Summary
 - More Examples

Limitation of Divide and Conquer

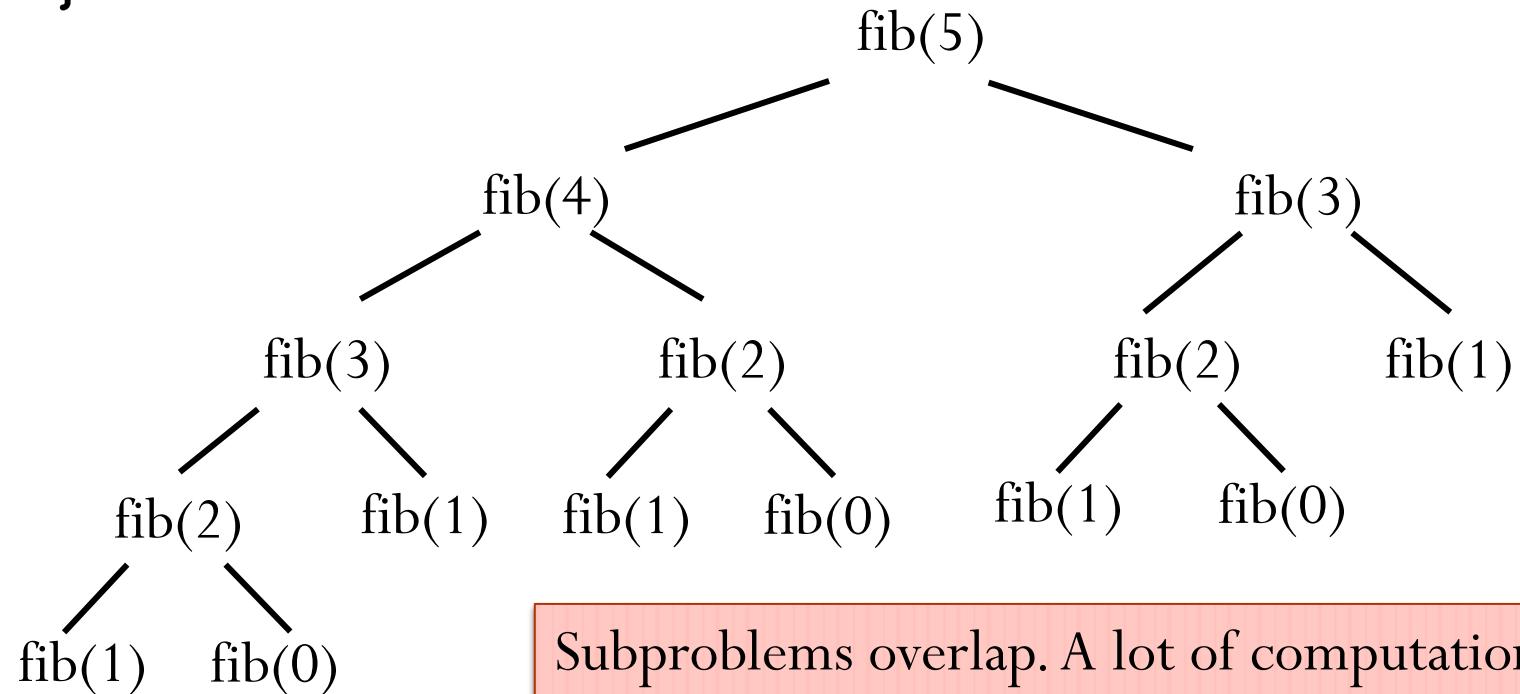
- Recursively solving subproblems can result in the same computations being repeated when the subproblems **overlap**.
- For example: computing the **Fibonacci sequence**
$$f_0 = 0; f_1 = 1; f_n = f_{n-1} + f_{n-2}, n \geq 2$$
- Divide and conquer approach:

```
int fib(int n) {  
    if(n <= 1) return n;  
    return fib(n-1)+fib(n-2);  
}
```

Fibonacci Sequence

Divide and Conquer Solution

```
int fib(int n) {  
    if(n <= 1) return n;  
    return fib(n-1)+fib(n-2);  
}
```



Subproblems overlap. A lot of computation is wasted. Time complexity is $\Omega(1.5^n)$.

Fibonacci Sequence

Iterative Solution

- We can also compute the Fibonacci sequence in iterative way:

```
int fib(int n) {  
    f[0] = 0; f[1] = 1;  
    for(i = 2 to n)  
        f[i] = f[i-1]+f[i-2];  
    return f[n];  
}
```

- Time complexity is $\Theta(n)$.

Dynamic Programming

- Used when a problem can be divided into subproblems that **overlap**.
 - Solve each subproblem **once** and store the solution in a table.
 - If a subproblem is encountered **again**, simply look up its solution in the table.
 - Reconstruct the solution to the original problem from the solutions to the subproblems.
- The more overlap the better, as this reduces the number of subproblems.
- Dynamic programming can be applied to solve **optimization problem**.

Optimization Problem

- Many problems we encounter are **optimization problems**:
 - A problem in which some function (called the **objective function**) is to be optimized (usually minimized or maximized) subject to some **constraints**.
- The solutions that satisfy the constraints are called **feasible solutions**.
- The number of feasible solutions is typically very large.
- We obtain the optimal solution by **searching** the feasible solution space.

Optimization Problem

Example

- Minimum spanning tree.
 - Objective function: the sum of all edge weights.
 - Constraints: the subgraph must be a spanning tree.

Outline

- Dynamic Programming
 - Motivation
 - Example: Matrix-Chain Multiplication
 - Summary
 - More Examples

Matrix-Chain Multiplication

- What is the cost of multiplying two matrices A and B ?
 - Suppose A is a $p \times q$ matrix and B is a $q \times r$ matrix.
 - Since the time to compute $C = AB$ is dominated by the number of **scalar multiplications**, we use the number of scalar multiplications as the complexity measure.
- $C_{ij} = \sum_{k=1}^q A_{ik}B_{kj}$.
 - We need q scalar multiplications to calculate C_{ij} .
 - C is of size $p \times r$.
- The number of scalar multiplications is pqr .

Matrix-Chain Multiplication

- Now how would you compute the multiplication of three matrices $A \times B \times C$?
 - Suppose A is of size 100×1 , B is of size 1×100 , and C is of size 100×1 .
- If we multiply as $(A \times B) \times C$, the number of scalar multiplications is 20000.
- If we multiply as $A \times (B \times C)$, the number of scalar multiplications is 200.

Matrix-Chain Multiplication

- If we want to multiply a chain of matrices $A_1 \times A_2 \times \cdots \times A_n$, where A_i is of size $p_{i-1} \times p_i$, what is the best order of multiplication to minimize the number of scalar multiplications?
- This is an optimization problem.
- It can be proved that number of different orders on n matrices is $\Omega(4^n/n^{1.5})$.
- Instead of enumerating all of the orders, can we do better to solve the optimization problem?

Matrix-Chain Multiplication

- For simplicity, define the problem of finding the optimal order to multiply $A_i \times A_{i+1} \times \cdots \times A_j$ as Q_{ij} . The minimal number of scalar multiplications is m_{ij} .
 - We ultimately want to solve Q_{1n} .

Matrix-Chain Multiplication

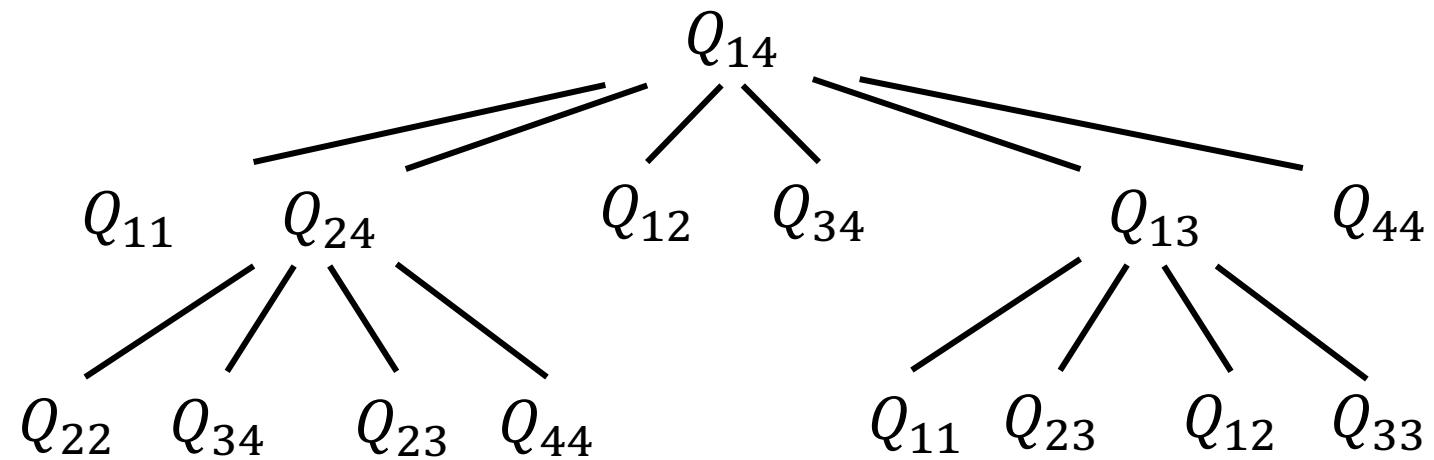
- Suppose in the optimal order for $A_i \times \cdots \times A_j$, the **last** multiplication is $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$.
- Then the order of computing $A_i \times \cdots \times A_k$ in the **optimal** order of computing $A_i \times \cdots \times A_j$ must be an **optimal** order to compute $A_i \times \cdots \times A_k$.
 - Why?
 - If not, then we copy and paste the better order \rightarrow we have a better order for computing $A_i \times \cdots \times A_j$!
 - Similar conclusion for computing $A_{k+1} \times \cdots \times A_j$.
- If we know k , we can divide the problem Q_{ij} into two smaller instances: Q_{ik} and $Q_{(k+1)j}$.

Matrix-Chain Multiplication

- Assume we have known the minimum number of scalar multiplications for Q_{ik} and $Q_{(k+1)j}$ as m_{ik} and $m_{(k+1)j}$.
 - Then $m_{ij} = m_{ik} + m_{(k+1)j} + p_{i-1}p_kp_j$.
- **However, we don't know k !** We need to consider all possible divisions, i.e., all $i \leq k \leq j - 1$.
- Thus, in order to solve Q_{ij} , we need to consider all subproblems Q_{ik} and $Q_{(k+1)j}$, for all $i \leq k \leq j - 1$.
 - $m_{ij} = \min_{i \leq k \leq j-1} (m_{ik} + m_{(k+1)j} + p_{i-1}p_kp_j)$

Matrix-Chain Multiplication

- In summary, we can divide the problem into subproblems of the same form.



Many subproblems are overlapped.

Matrix-Chain Multiplication

- The straightforward recursive algorithm has exponential time complexity.
 - However, it will encounter each subproblem many times in different branches of the tree.
- The total number of different subproblems is not exponential.
 - They are Q_{ij} , for $1 \leq i \leq j \leq n$.
 - The total number is $n(n + 1)/2$.
- Instead, we use a tabular, bottom-up approach.

Matrix-Chain Multiplication

Bottom-up Approach

- Apply the recursive relation:

$$m_{ij} = \min_{i \leq k \leq j-1} (m_{ik} + m_{(k+1)j} + p_{i-1}p_kp_j)$$

- Initial situation $m_{11} = m_{22} = \dots = m_{nn} = 0$.
- In the first round, we compute $m_{12}, m_{23}, \dots, m_{(n-1)n}$.
- In the second round, we compute $m_{13}, m_{24}, \dots, m_{(n-2)n}$.
- So on and so forth. In the l -th round, we compute $m_{1(l+1)}, m_{2(l+2)}, \dots, m_{(n-l)n}$.
- Finally, we compute m_{1n} .
- To obtain the multiplication order, we also record the partition k which gives the minimal m_{ij} as s_{ij} .

Matrix-Chain Multiplication

Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

		m_{ij}	1	2	j	3	4
			1	2		3	4
			1	2		3	4
i		1	0				
i		2	—	0			
i		3	—	—	0		
i		4	—	—	—	0	

		s_{ij}	1	2	j	3	4
			1	2		3	4
			1	2		3	4
i		1	—				
i		2	—	—			
i		3	—	—	—		
i		4	—	—	—	—	

Matrix-Chain Multiplication

Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

		m_{ij}	1	2	j	3	4
			1	2		3	4
			1	2		3	4
i	1	0					
	2	—	0				
	3	—	—	0			
	4	—	—	—	0		

		s_{ij}	1	2	j	3	4
			1	2		3	4
			1	2		3	4
i	1	—					
	2	—	—				
	3	—	—	—			
	4	—	—	—	—		

$$\begin{aligned}m_{i(i+1)} &= m_{ii} + m_{(i+1)(i+1)} + p_{i-1}p_ip_{i+1} \\&= p_{i-1}p_ip_{i+1}\end{aligned}$$

Matrix-Chain Multiplication

Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

		m_{ij}	1	2	j	3	4
			1	2		3	4
			1	2		3	4
i	1	0	100				
	2	—	0	10			
	3	—	—	0	200		
	4	—	—	—	0		

		s_{ij}	1	2	j	3	4
			1	2		3	4
			1	2		3	4
i	1	—	1				
	2	—	—	2			
	3	—	—	—	3		
	4	—	—	—	—		

$$\begin{aligned}m_{i(i+1)} &= m_{ii} + m_{(i+1)(i+1)} + p_{i-1}p_ip_{i+1} \\&= p_{i-1}p_ip_{i+1}\end{aligned}$$

Matrix-Chain Multiplication

Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

		j	1	2	3	4
		i	1	2	3	4
m_{ij}		1	0	100		
	1	—	0	10		
	2	—	—	0	200	
	3	—	—	—	0	
	4	—	—	—	—	0

		j	1	2	3	4
		i	1	2	3	4
s_{ij}		1	—	1		
	1	—	—	2		
	2	—	—	—	3	
	3	—	—	—	—	
	4	—	—	—	—	—

$$m_{i(i+2)} = \min\{m_{ii} + m_{(i+1)(i+2)} + p_{i-1}p_ip_{i+2},$$

$$m_{i(i+1)} + m_{(i+2)(i+2)} + p_{i-1}p_{i+1}p_{i+2}\}$$

Matrix-Chain Multiplication

Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

		m_{ij}	1	2	j	3	4
			1	2		3	4
			1	2		3	4
i	1	0	100				
	2	—	0	10			
	3	—	—	0	200		
	4	—	—	—	0		

		s_{ij}	1	2	j	3	4
			1	2		3	4
			1	2		3	4
i	1	—	1				
	2	—	—	2			
	3	—	—	—	3		
	4	—	—	—	—		

$$m_{13} = \min\{m_{11} + m_{23} + p_0 p_1 p_3,$$

$$m_{12} + m_{33} + p_0 p_2 p_3\} = \min\{20, 200\}$$

Matrix-Chain Multiplication

Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

		m_{ij}	1	2	j	3	4
			1	2		3	4
i		1	0	100	20		
		2	—	0	10		
3	—	—	0	200			
4	—	—	—	0			

		s_{ij}	1	2	j	3	4
			1	2		3	4
i		1	—	1	1		
		2	—	—	2		
3	—	—	—	—	3		
4	—	—	—	—	—		

$$m_{13} = \min\{m_{11} + m_{23} + p_0 p_1 p_3,$$

$$m_{12} + m_{33} + p_0 p_2 p_3\} = \min\{20, 200\}$$

Matrix-Chain Multiplication

Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

		m_{ij}	1	2	j	3	4
			1	2		3	4
		i	1	2		3	4
	1		0	100	20		
	2		—	0	10		
	3		—	—	0	200	
	4		—	—	—	0	

		s_{ij}	1	2	j	3	4
			1	2		3	4
		i	1	2		3	4
	1		—	1	1		
	2		—	—	2		
	3		—	—	—	3	
	4		—	—	—	—	

$$m_{24} = \min\{m_{22} + m_{34} + p_1 p_2 p_4,$$

$$m_{23} + m_{44} + p_1 p_3 p_4\} = \min\{400, 30\}$$

Matrix-Chain Multiplication

Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

		i	j	
		1	2	3
i	1	0	100	20
	2	—	0	10
	3	—	—	0
	4	—	—	—

		i	j	
		1	2	3
i	1	—	1	1
	2	—	—	2
	3	—	—	—
	4	—	—	—

$$m_{24} = \min\{m_{22} + m_{34} + p_1 p_2 p_4,$$

$$m_{23} + m_{44} + p_1 p_3 p_4\} = \min\{400, 30\}$$

Matrix-Chain Multiplication

Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

		m_{ij}	1	2	j	3	4
			1	2	3	4	
i		1	0	100	20		
		2	—	0	10	30	
i		3	—	—	0	200	
		4	—	—	—	0	

		s_{ij}	1	2	j	3	4
			1	2	3	4	
i		1	—	1	1		
		2	—	—	2	3	
i		3	—	—	—	3	
		4	—	—	—	—	

$$m_{i(i+3)} = \min_{i \leq k \leq i+2} (m_{ik} + m_{(k+1)(i+3)} + p_{i-1}p_k p_{(i+3)})$$

Matrix-Chain Multiplication

Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

		m_{ij}	1	2	j	3	4
			1	2	3	4	
i		1	0	100	20		
		2	—	0	10	30	
i		3	—	—	0	200	
		4	—	—	—	0	

		s_{ij}	1	2	j	3	4
			1	2	3	4	
i		1	—	1	1		
		2	—	—	2	3	
i		3	—	—	—	3	
		4	—	—	—	—	

$$\begin{aligned}m_{14} &= \min_{1 \leq k \leq 3} (m_{1k} + m_{(k+1)4} + p_0 p_k p_4) \\&= \min\{230, 2300, 220\}\end{aligned}$$

Matrix-Chain Multiplication

Example

- $n = 4, A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

		i	j	
		1	2	3
i	1	0	100	20
	2	—	0	10
	3	—	—	0
	4	—	—	—

Optimal Value

		i	j	
		1	2	3
i	1	—	1	1
	2	—	—	2
	3	—	—	—
	4	—	—	—

$$\begin{aligned}m_{14} &= \min_{1 \leq k \leq 3} (m_{1k} + m_{(k+1)4} + p_0 p_k p_4) \\&= \min\{230, 2300, 220\}\end{aligned}$$

Matrix-Chain Multiplication

Constructing an Optimal Order

- We can construct an optimal order based on the records s_{ij} .

```
Print_Order(s, i, j) {
    if(i == j) cout << "Ai";
    else {
        cout << "(";
        Print_Order(s, i, sij);
        cout << "*";
        Print_Order(s, sij+1, j);
        cout << ")";
    }
}
```

- Initial call is **Print_Order(s, 1, n)**;

Matrix-Chain Multiplication

Example

- Construct an optimal order
 - $n = 4, A_1 \times A_2 \times A_3 \times A_4$.
 - $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20$.

s_{ij}	1	2	j	3	4
1	—	1	1	3	
2	—	—	2	3	
3	—	—	—	3	
4	—	—	—	—	

$$s_{14} = 3 \rightarrow A_1 \times A_2 \times A_3 \times A_4 = (A_1 \times A_2 \times A_3) \times A_4$$

$$s_{13} = 1 \rightarrow A_1 \times A_2 \times A_3 = A_1 \times (A_2 \times A_3)$$

$$s_{23} = 2 \rightarrow A_2 \times A_3 = A_2 \times A_3$$

$$A_1 \times A_2 \times A_3 \times A_4 = (A_1 \times (A_2 \times A_3)) \times A_4$$

Matrix-Chain Multiplication

Time Complexity

- Get the minimum number of scalar multiplications:
 - We need to obtain all m_{ij} and s_{ij} , for $1 \leq i \leq j \leq n$.
 - $O(n^2)$ records
 - Each m_{ij} is the minimum of $O(n)$ terms.
 - Total time complexity is $O(n^3)$.
- Obtain the optimal order:
 - $O(n)$

Matrix-Chain Multiplication

Summary

- Matrix-chain multiplication is an optimization problem.
- The solution is based on **dynamic programming**.
 - The original problem can be divided into same subproblems that **overlap**.
 - Each subproblem is solved once and stored in a table.
 - If a subproblem is encountered again, simply look up its solution in the table.
 - Reconstruct the solution to the original problem from the solutions to the subproblems.

Outline

- Dynamic Programming
 - Motivation
 - Example: Matrix-Chain Multiplication
 - Summary
 - More Examples

Dynamic Programming for Optimization

- There are two key ingredients that an optimization problem must have in order for dynamic programming to apply:
 - Optimal substructure;
 - Overlapping subproblems.

Optimal Substructure

- An optimal solution to the problem contains **within it optimal solutions to subproblems.**
 - In matrix-chain multiplication, the optimal order on calculating $A_i \times \cdots \times A_j$ that splits the product between A_k and A_{k+1} contains within it optimal solutions to the problem of ordering $A_i \times \cdots \times A_k$ and $A_{k+1} \times \cdots \times A_j$.
- You can show optimal substructure property by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction.

Overlapping Subproblems

- A recursive algorithm for the problem solves the same subproblems **over and over**, rather than always generating new subproblems.
 - E.g., subproblems of matrix-chain multiplication overlap.
 - In contrast, a problem for which a divide-and-conquer approach is suitable usually generates **brand-new** problems at each step of the recursion.
- Dynamic-programming algorithms take advantage of overlapping subproblems by
 - solving each subproblem once ...
 - ... and then storing the solution in a table where it can be looked up when needed.

Designing a Dynamic-Programming Algorithm

1. Characterize **the structure** of an optimal solution.
 - Usually, we need to define a **general** problem.
2. **Recursively** define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a **bottom-up** fashion.
4. Construct an optimal solution from computed information.

Outline

- Dynamic Programming
 - Motivation
 - Example: Matrix-Chain Multiplication
 - Summary
 - More Examples

Longest Common Subsequence

- **Problem:** Given 2 sequences, $X = \langle x_1, \dots, x_m \rangle$ and $Y = \langle y_1, \dots, y_n \rangle$, find a common subsequence whose length is maximum.

springtime
| | | | |
printing

ncaa tournament
| | | |
north carolina

basketball
| |
krzyzewski

Subsequence need not be consecutive, but must be in order.

Naïve Algorithm

- For every subsequence of X , check whether it's a subsequence of Y .
- **Time:** $\Theta(n2^m)$.
 - 2^m subsequences of X to check.
 - Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, for second, and so on.

Optimal Substructure

Theorem

Let $Z = \langle z_1, \dots, z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .
 2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y .
 3. or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1} .

Notation:

prefix $X_i = \langle x_1, \dots, x_i \rangle$ is the first i letters of X .

This says what any longest common subsequence must look like;
do you believe it?

Optimal Substructure

Theorem

Let $Z = \langle z_1, \dots, z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .
 2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y .
 3. or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1} .

Proof: (case 1: $x_m = y_n$)

Any sequence Z' that does not end in $x_m = y_n$ can be made longer by adding $x_m = y_n$ to the end.

Therefore,

- (1) longest common subsequence (LCS) Z must end in $x_m = y_n$.
 - (2) Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1} , and
 - (3) there is no longer CS of X_{m-1} and Y_{n-1} , or Z would not be an LCS.

Optimal Substructure

Theorem

Let $Z = \langle z_1, \dots, z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .
 2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y .
 3. or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1} .

Proof: (case 2: $x_m \neq y_n$, and $z_k \neq x_m$)

Since Z does not end in x_m ,

- (1) Z is a common subsequence of X_{m-1} and Y , and
 - (2) there is no longer CS of X_{m-1} and Y , or Z would not be an LCS.

Recursion for Length

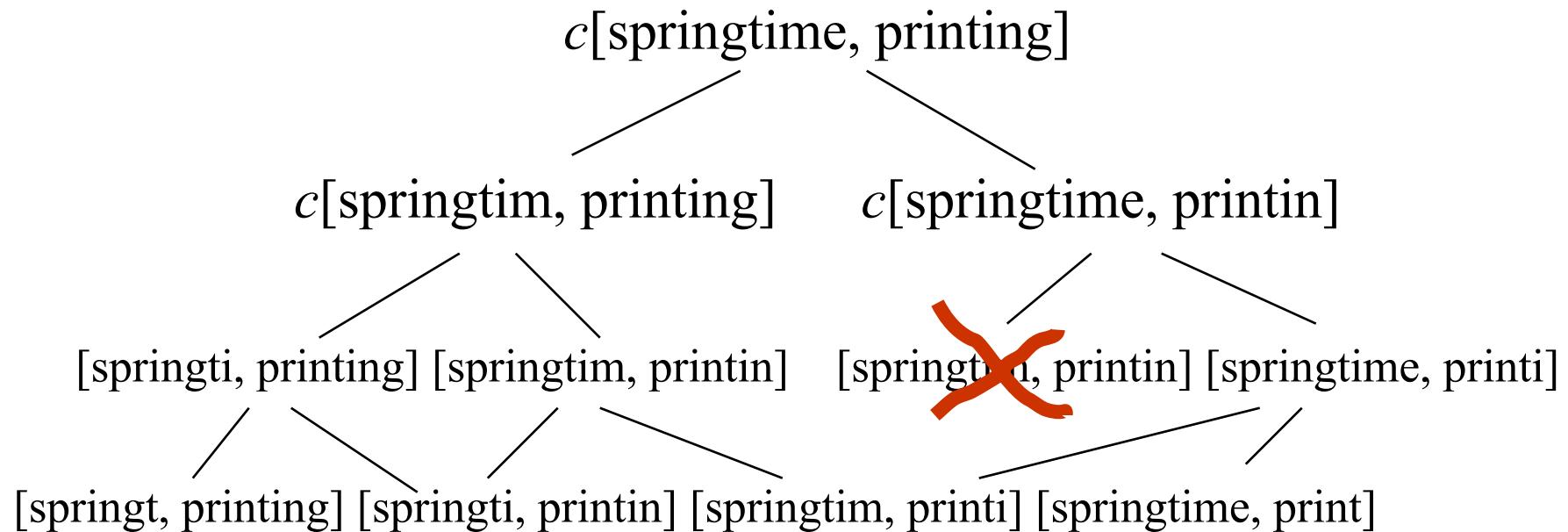
- Define $c[i, j] = \text{length of LCS of } X_i \text{ and } Y_j$.
- We want $c[m, n]$.

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

This gives a recursive algorithm and solves the problem.
But does it solve it well?

Recursive Solution

$$c[\alpha, \beta] = \begin{cases} 0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,} \\ c[prefix\alpha, prefix\beta] + 1 & \text{if } \text{end}(\alpha) = \text{end}(\beta), \\ \max(c[prefix\alpha, \beta], c[\alpha, prefix\beta]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta). \end{cases}$$



Computing the length of an LCS

LCS-LENGTH (X, Y)

```
1.  $m \leftarrow \text{length}[X]$ 
2.  $n \leftarrow \text{length}[Y]$ 
3. for  $i \leftarrow 1$  to  $m$ 
   4.   do  $c[i, 0] \leftarrow 0$ 
5. for  $j \leftarrow 0$  to  $n$ 
   6.   do  $c[0, j] \leftarrow 0$ 
7. for  $i \leftarrow 1$  to  $m$ 
   8.   do for  $j \leftarrow 1$  to  $n$ 
      9.     do if  $x_i = y_j$ 
         10.        then  $c[i, j] \leftarrow c[i-1, j-1] + 1$ 
            11.         $b[i, j] \leftarrow "$ ↖“
         12.        else if  $c[i-1, j] \geq c[i, j-1]$ 
            13.          then  $c[i, j] \leftarrow c[i-1, j]$ 
            14.           $b[i, j] \leftarrow "$ ↑“
         15.        else  $c[i, j] \leftarrow c[i, j-1]$ 
            16.           $b[i, j] \leftarrow "$ ←“
17. return  $c$  and  $b$ 
```

- Keep track of $c[a,b]$ in a table of nm entries:

- top/down
- bottom/up

$b[i, j]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j .

$c[m,n]$ contains the length of an LCS of X and Y .

Time: $O(mn)$

Constructing an LCS

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

1. **if** $i = 0$ or $j = 0$
2. **then return**
3. **if** $b[i, j] = \nwarrow$
4. **then** PRINT-LCS($b, X, i-1, j-1$)
5. print x_i
6. **elseif** $b[i, j] = \uparrow$
7. **then** PRINT-LCS($b, X, i-1, j$)
8. **else** PRINT-LCS($b, X, i, j-1$)

- Initial call is PRINT-LCS (b, X, m, n).
- When $b[i, j] = \nwarrow$, we have extended LCS by one character. So LCS = entries with \nwarrow in them.
- Time: $O(m+n)$

More problems

Optimal BST: Given sequence $K = k_1 < k_2 < \dots < k_n$ of n sorted keys, with a search probability p_i for each key k_i , build a binary search tree (BST) **with minimum expected search cost**.

Minimum convex decomposition of a polygon,

Hydrogen placement in protein structures, ...

References and Readings

- Thomas L. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, “Introduction to Algorithms” (3rd Edition)
 - Chapter 15 Dynamic Programming