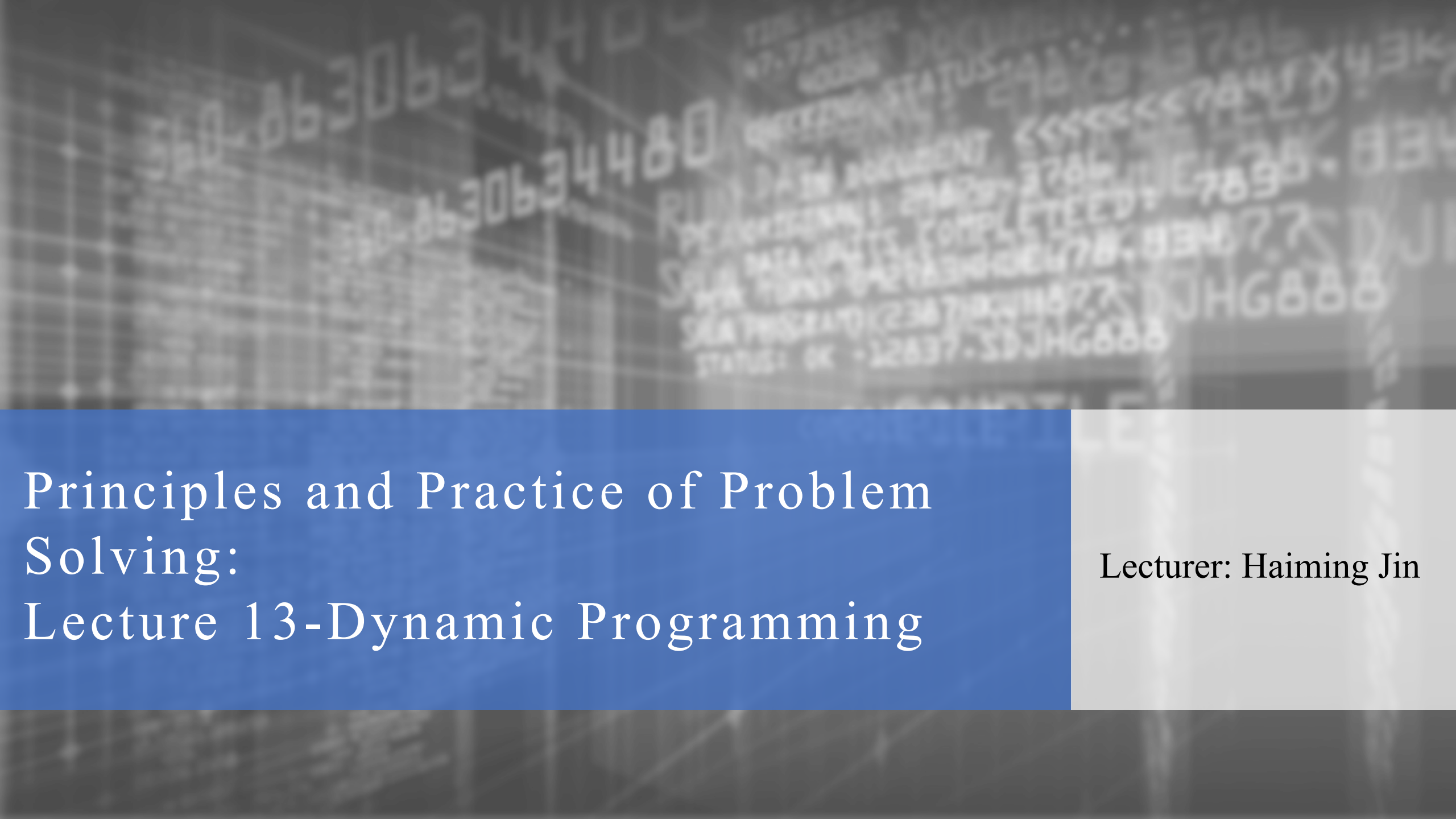


Principles and Practice of Problem Solving: Lecture 13-Dynamic Programming

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**

$$f_0 = 0;\ \ f_1 = 1;\ \ f_n = f_{n-1} + f_{n-2}, n \geq 2$$

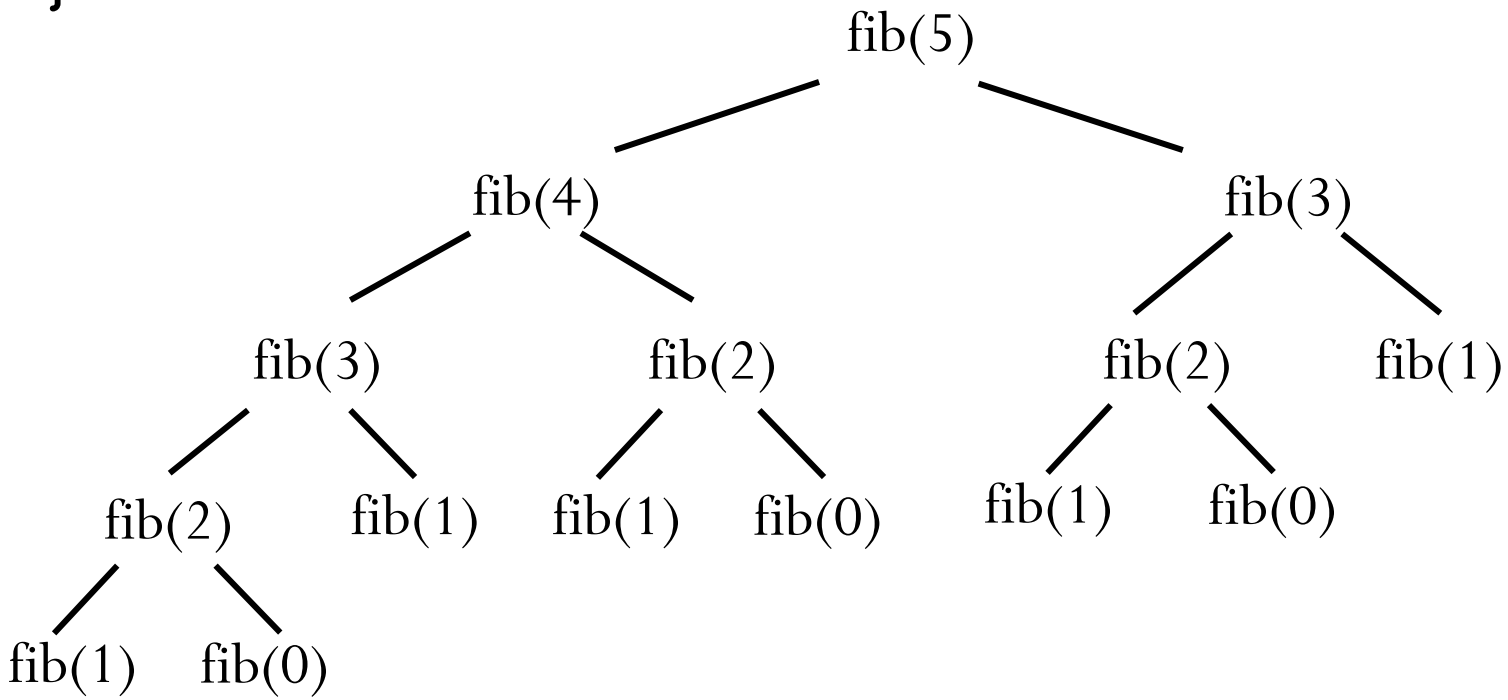
- Divide and conquer approach:

```
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);
}
```



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.

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

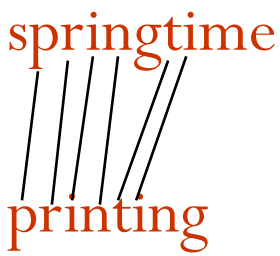
1. Characterize **the structure** of an optimal solution.
   - Usually, we need to define a **general** problem.
2. **Recursively** define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a **bottom-up** fashion.
4. Construct an optimal solution from computed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

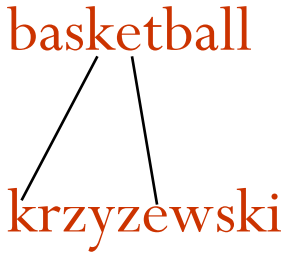
Outline

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

Longest Common Subsequence

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





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 $Z = \langle z_1, \ldots, z_k \rangle$ 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 $X_i = \langle x_1, \ldots, x_i \rangle$ 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 $Z = \langle z_1, \ldots, z_k \rangle$ 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 $Z = \langle z_1, \ldots, z_k \rangle$ 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

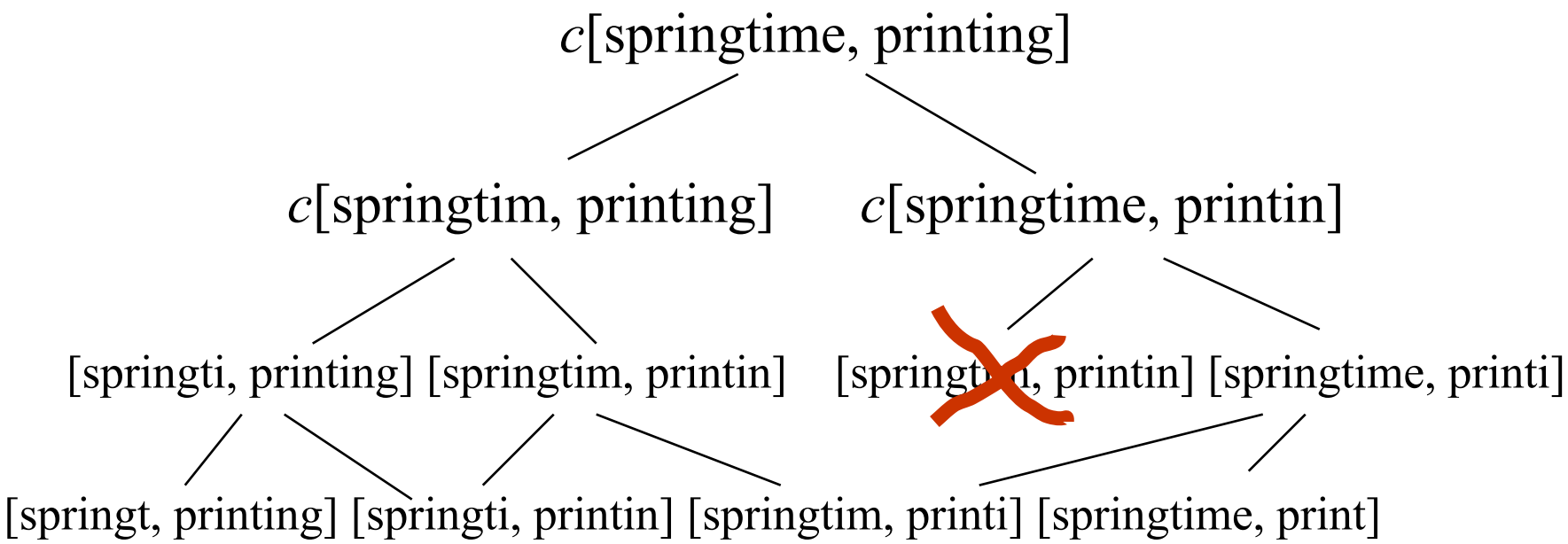
- Define $c[i, j]$ = length of LCS of X_i and Y_j .
- We want $c[m,n]$.

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

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 ← length[X]
2.  n ← length[Y]
3.  for i ← 1 to m
4.      do c[i, 0] ← 0
5.  for j ← 0 to n
6.      do c[0, j ] ← 0
7.  for i ← 1 to m
8.       do for j ← 1 to n
9.           do if x_i = y_j
10.                then c[i, j ] ← c[i−1, j−1] + 1
11.                     b[i, j ] ← "↖"
12.           else if c[i−1, j ] ≥ c[i, j−1]
13.                then c[i, j ] ← c[i−1, j ]
14.                     b[i, j ] ← "↑"
15.                else c[i, j ] ← c[i, j−1]
16.                     b[i, j ] ← "←"
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 ] = "↖"
4.     then PRINT-LCS(b, X, i−1, j−1)
5.          print x_i
6.     elseif b[i, j ] = "↑"
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 $K = k_1 < k_2 < \cdots < k_n$ 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 Progrmaming