数据结构与算法设计

周　可

Mail：zhke@hust.edu.cn

华中科技大学，武汉光电国家研究中心

自顶向下

自顶向下设计：

整体→局部，将系统分割成子系统和子模块，step-wise refinement

自顶向下做事：

抽象→具体，讲究一个"拆"，将大的目标落实为各项具体行动。

分治法（回顾）

VS

自底向上

自底向上设计：

简单→复杂，从系统最基础的部分着手，逐层向上构造，直到得到所要的软件系统。

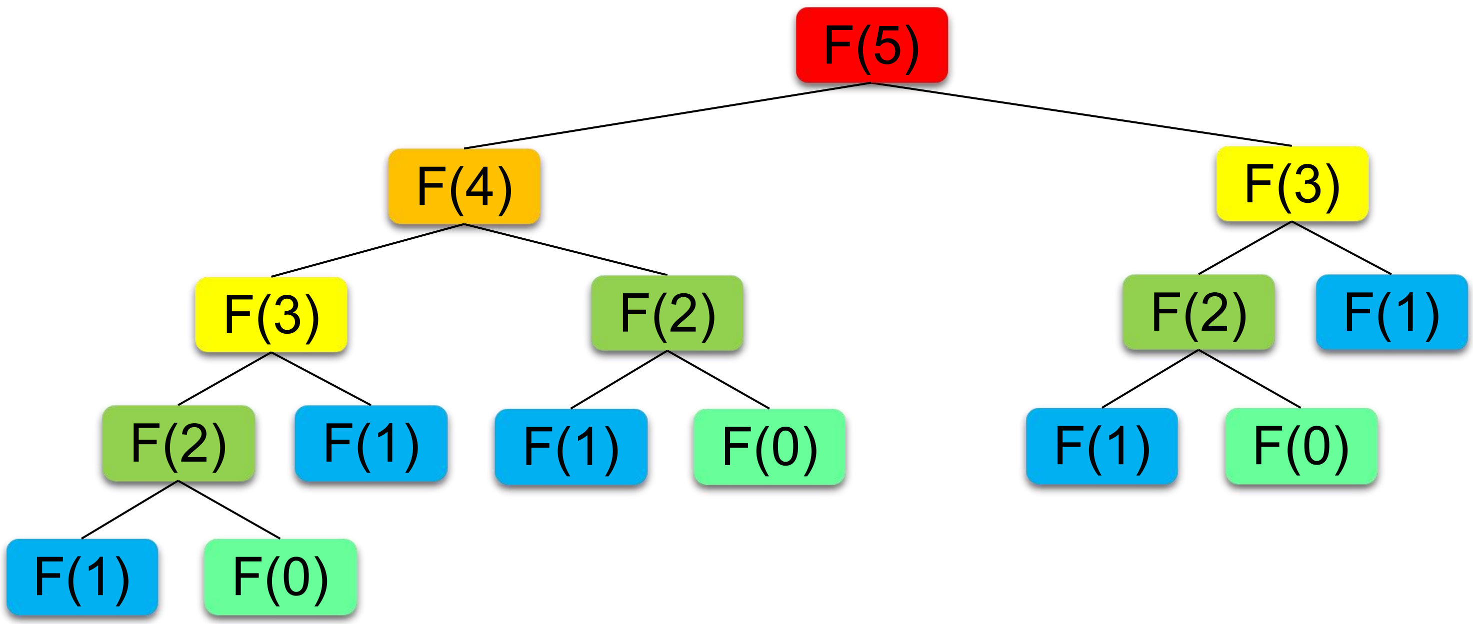
自底向上思考：

局部→整体，逐渐凝练出概览全局的整体。

动态规划

斐波拉契数

- 斐波拉契数列：1，1，2，3，5，8，13，21，34，55，……
- 递推公式： $F(n)=F(n-1)+F(n-2)$ （$n\geqslant 2$，$F(0)=1$，$F(1)=1$）
- 直接按照递推方式计算，有什么问题？**大量重复计算!**



斐波那契数

Idea

You have a large problem to solve, you can divide the problem into smaller sub-problems

(1) Solution of a sub-problem might be **inter-related** and might be **re-used** again (this is **different from Divide & Conquer**).

(2) So it is better to store those smaller solutions somewhere.

Key idea：Space for Time

Dynamic programming

- Matrix Chain Multiplication
- Introduction of Dynamic Programming
- Longest Common Subsequence

Example 1. Matrix Chain Multiplication

Given n matrices $M_1, M_2, \ldots, M_n$, compute the product $M_1M_2M_3\ldots M_n$, where M_i has dimension $\mathbf{d_{i-1}}$ **x** $\mathbf{d_i}$ (i.e., with d_{i-1} rows and d_i columns), for $i = 1,\ldots,n$.

Fact 1. Given matrices A with dimension p x q and B with dimension q x r, multiplication AB takes pqr scalar multiplications.

Objective?——To compute $M_1M_2M_3\ldots M_n$ with the minimum number of scalar multiplications.

Example 1. Matrix Chain Multiplication

Problem: Parenthesize the product $M_1M_2 \ldots M_n$ in a way to minimize the number of scalar multiplications.

Example. M_1 --- 20 x 10

M_2 --- 10 x 50

M_3 --- 50 x 5

M_4 --- 5 x 30

$(((M_1M_2)M_3)M_4)$ --- 18000 multiplications

$(M_1((M_2M_3)M_4))$ --- 10000 multiplications

$((M_1M_2)(M_3M_4))$ --- 47500 multiplications

$\boxed{((M_1(M_2M_3))M_4) \text{ --- 6500 multiplications}}$

$(M_1(M_2(M_3M_4)))$ --- 28500 multiplications

Example 1. Matrix Chain Multiplication

Problem: Parenthesize the product $M_1M_2\ldots M_n$ in a way to minimize the number of scalar multiplications.

However, exhaustive search is not efficient. Let P(n) be the number of alternative parenthesizations of n matrices.

$P(n) = 1,$ if $n=1$

$P(n) = \sum_{k=\mathbf{1} \text{ to } \mathbf{n-1}} P(k)P(n-k)$, if $n \geq 2$

$$\left.\begin{array}{l} M_1 M_2 \ldots M_{n-1} M_n \\ \quad\cdots \\ M_1 M_2 \ldots M_{n-1} M_n \end{array}\right\} \textbf{n-1种}$$

Example 1. Matrix Chain Multiplication

So let's use dynamic programming.

Let m_{ij} be the number of multiplications performed using an optimal parenthesization of $M_iM_{i+1}\ldots M_{j-1}M_j$.

$$M_iM_{i+1}\ldots M_k \mid M_{k+1}\ldots M_{j-1}M_j$$

- $m_{ii} = 0$
- $m_{ij} = \min_k\{\mathbf{m_{ik}} + \mathbf{m_{k+1,j}} + d_{i-1}d_kd_j,\ 1 \le i \le k < j \le n\}$

Example 1. Matrix chain multiplication

M_1 --- 20 x 10

M_2 --- 10 x 50

M_3 --- 50 x 5

M_4 --- 5 x 30

$M_1M_2M_3M_4$

i \ j	1	2	3	4
1	0	10000	**3500**	
2		0	**2500**	4000
3			0	7500
4				0

Round 2

Round 1

Round 0

$$m_{13} \left\{ \begin{array}{l} M_1 | M_2 M_3 \\ M_1 M_2 | M_3 \end{array} \right.$$

Dynamic programming is **bottom-up**!

- $m_{ii} = 0$
- $m_{ij} = \min_k\{m_{ik} + m_{k+1,j} + d_{i-1}d_kd_j,\ 1 \le i \le k < j \le n\}$

Example 1. Matrix chain multiplication

i \ j	1	2	3	4
1	0	10000	3500	**6500**
2		0	2500	4000
3			0	7500
4				0

m[1,4] contains the value of the optimal solution.

$$m_{14} \left\{ \begin{array}{l} M_1 \mid M_2 M_3 M_4 \\ M_1 M_2 \mid M_3 M_4 \\ M_1 M_2 M_3 \mid M_4 \end{array} \right.$$

- $m_{ii} = 0$
- $m_{ij} = \min_k \{ m_{ik} + m_{k+1,j} + d_{i-1} d_k d_j,\ 1 \le i \le k < j \le n \}$

Example 1. Matrix chain multiplication

m[1,4] contains the value of the optimal solution.

i \ j	1	2	3	4
1	0	10000	3500	**6500**
2		0	2500	4000
3			0	7500
4				0

Round 3

Round 2

Round 1

Round 0

$$m_{14} \left\{ \begin{array}{l} M_1 \mid M_2M_3M_4 \\ M_1M_2 \mid M_3M_4 \\ M_1M_2M_3 \mid M_4 \end{array} \right.$$

- $m_{ii} = 0$
- $m_{ij} = \min_k\{m_{ik} + m_{k+1,j} + d_{i-1}d_kd_j, 1 \le i \le k < j \le n\}$

```
MATRIX-CHAIN-ORDER(p)
n = p.length - 1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i = 1 to n
    m[i,i] = 0
for l = 2 to n                          // l is the chain length
    for i = 1 to n - l + 1
        j = i + l - 1
        m[i,j] = ∞
        for k = i to j - 1
            q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
            if q < m[i,j]
                m[i,j] = q
                s[i,j] = k
return m and s
```

Dynamic programming

- Matrix Chain Multiplication
- Introduction of Dynamic Programming
- Longest Common Subsequence

动态规划原理

动态规划（Dynamic Programming，DP）是运筹学的一个分支，是求解决策过程最优化问题的数学方法。与分治方法相似，都是<u>通过组合子问题的解来求解原问题。</u>

Q：什么情况下用动态规划方法求解问题？

A：适合应用动态规划方法求解的最优化问题应该具备的<u>两个要素：最优子结构（Optimal substructure）</u>和<u>子问题重叠（Overlapping subproblems）</u>。

Dynamic-programming hallmark #1

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

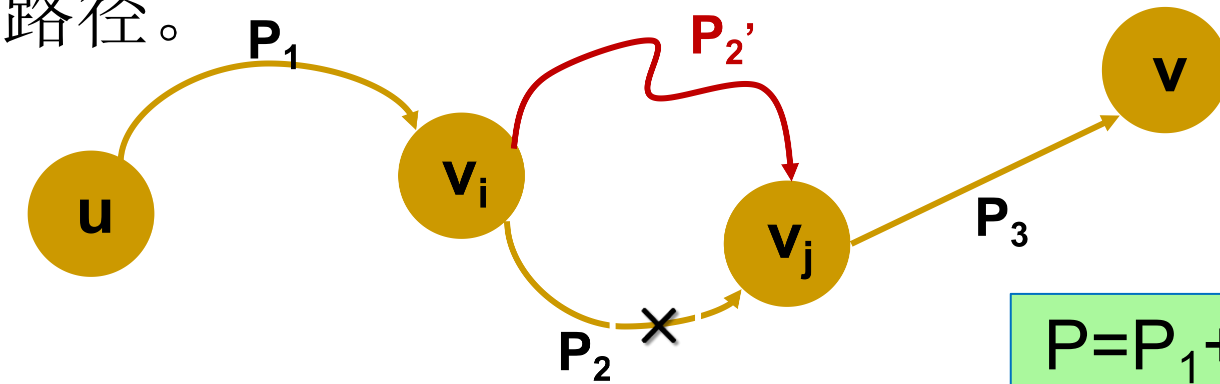
无权最短路径问题：

给定一个有向图G=(V,E)和两个顶点u,v∈V，找到一条从u到v的边数最少的路径。

最短路径问题具有最优子结构的性质吗？如何证明？

讨论：最短路径的性质

给定一个包含n个顶点的带权有向图G=(V,E)，假定δ(u,v)=<u=v_1, …, v_i, …, v_j, …,v_k=v>是从u到v的最短路径。



P=P_1+P_2+P_3最短

- δ(u,v)具有什么性质？

<v_i, …, v_j>是从v_i到v_j的最短路径。

- 如何证明？**反证法！** “Cut-Paste”

假设P_2'＜ P_2, 则

(P ' =P_1+P_2'+P_3) ＜ (P=P_1+P_2+P_3)

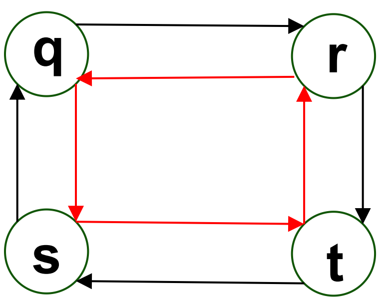
Contradiction!

无权最长路径问题：

给定一个有向图G=(V,E)和两个顶点u,v∈V，找到一条从u到v的边数最多的路径。

无权最长路径问题具有最优子结构的性质吗?

NO!



q→r→t是从q到t的最长简单路径，

q→r是从q到r的最长简单路径吗？

r→t是从r到t的最长简单路径吗？

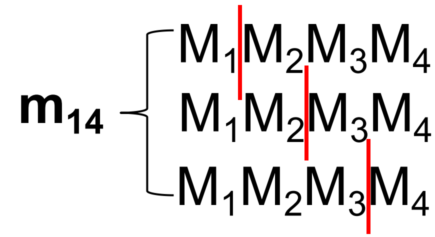
Dynamic-programming hallmark #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

- 递归算法效率很低，是因为反复求解相同子问题。
- 动态规划思想：对每个子问题只求解一次，将结果保存起来，是典型的time-memory trade-off。

以矩阵链乘法为例，说明子问题重叠性质：



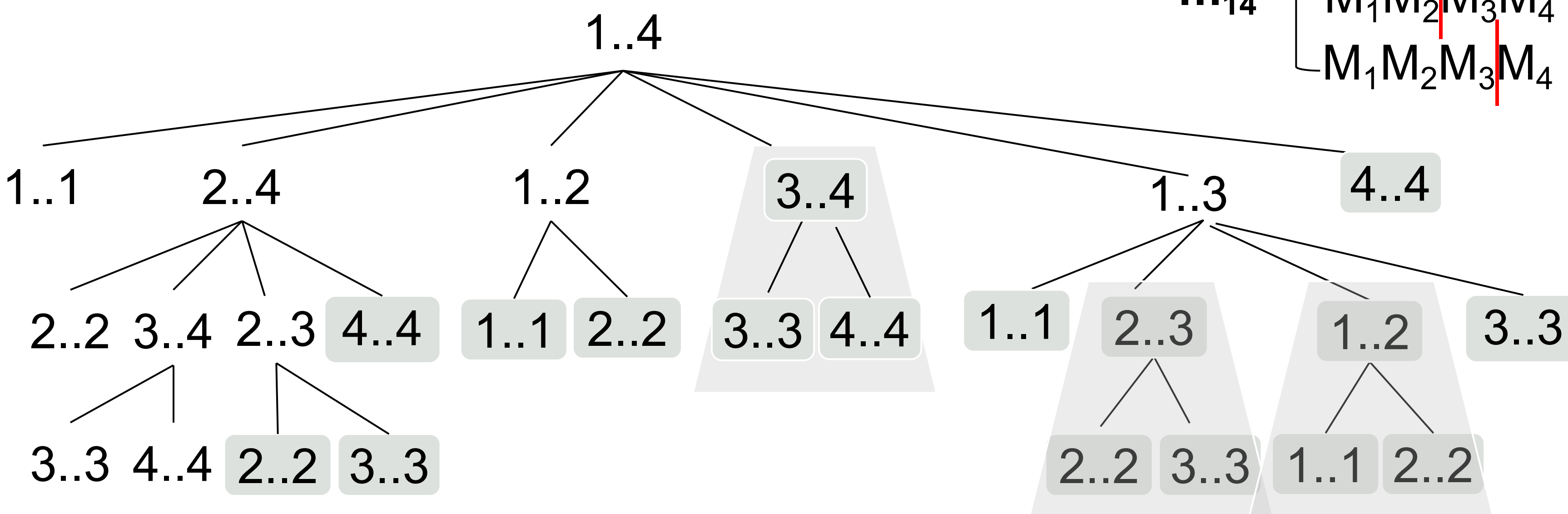


图15-7 RECURSIVE-MATRIX-CHAIN(p，1，4)所产生的递归调用树

- 采用深度优先搜索（DFS）描述<u>*带备忘机制的自顶向下动态规划算法*</u>处理子问题图的顺序。
- 朴素递归算法→<u>*指数时间(Ω (2^n)) （推导P220）*</u>
- 动态规划算法→<u>*多项式时间(O (n^3))*</u>

矩阵链乘法的递归求解

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j−1
    q = RECURSIVE-MATRIX-CHAIN (p, i, k)
        + RECURSIVE-MATRIX-CHAIN (p, k+1, j)
        + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

```
MEMOIZED-MATRIX-CHAIN (p)

n = p.length - 1
let m[1..n, 1..n] be a new table
for i = 1 to n
    for j = i to n
        m[i, j] = ∞
return LOOKUP-CHAIN(m, p, 1, n)
```

```
LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j-1
        q = LOOKUP-CHAIN(m, p, i, k)
            + LOOKUP-CHAIN(m, p, k+1, j) + p_{i-1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

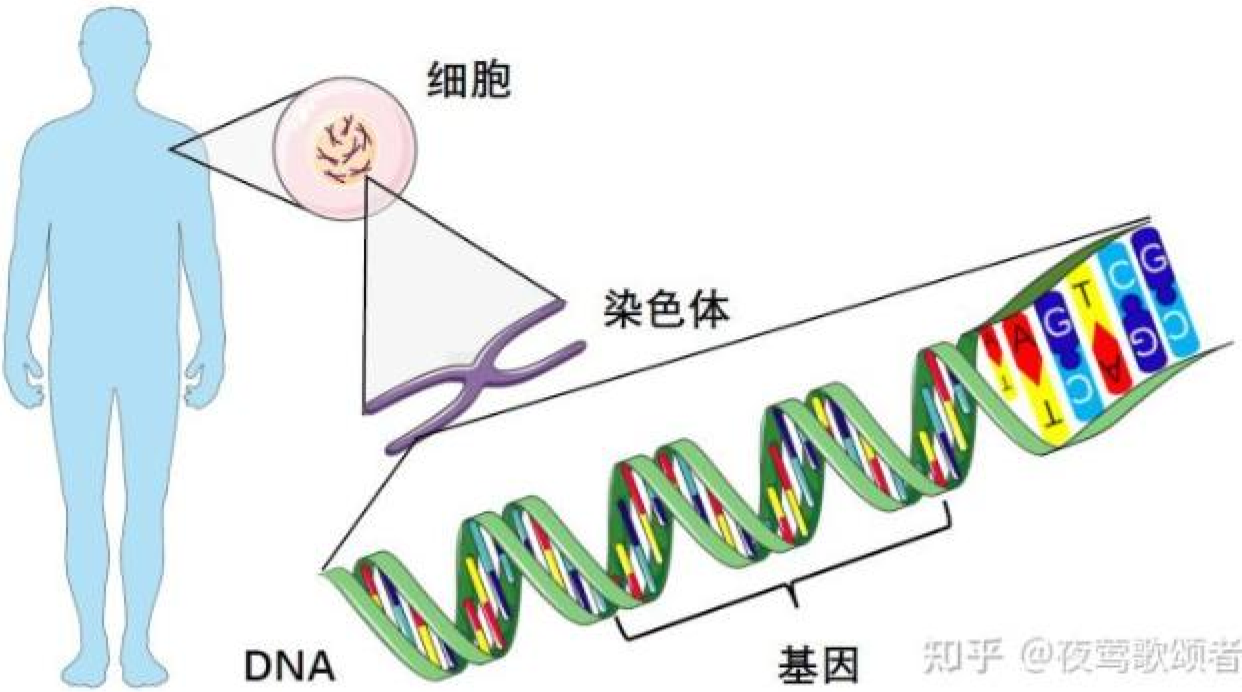
小结

适合应用动态规划方法求解的最优化问题应该具备的两个要素：最优子结构（Optimal substructure）和子问题重叠（Overlapping subproblems）。

Dynamic programming

- Matrix Chain Multiplication
- Introduction of Dynamic Programming
- Longest Common Subsequence

Longest Common Subsequence (LCS)



问题：如何比较两个**DNA**串的相似度？

<u>直接比较、转换操作、子串法</u>

Longest Common Subsequence

- Given two sequences $x[1 \,..\, m]$ and $y[1 \,..\, n]$, find a longest subsequence common to them both.

“a” *not* “the”

x: A B C B D A B

y: B D C A B A

BCBA = LCS(x, y)

functional notation, but not a function

Q: Can you find another LCS in this case?

A: BCAB is another LCS, so longest common subsequence is not unique!

Brute-force LCS algorithm

Check every subsequence of $x[1 \mathinner{..} m]$ to see if it is also a subsequence of $y[1 \mathinner{..} n]$.

Analysis

- Checking $= O(n)$ time per subsequence.
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time $= O(n2^m)$

$=$ exponential time.

Towards a better algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $|s|$.

Strategy: Consider ***prefixes*** of x and y.

- Define $c[i, j] = |\mathrm{LCS}(x[1 \,..\, i], y[1 \,..\, j])|$.
- Then, $c[i, j] = |\mathrm{LCS}(x, y)|$.

-Define table c[-,-]

-c[i,j] stores the length of an LCS of the sequences X[1..i] and Y[1..j].

$c[i,j] = 0,$ if $i=0$ or $j=0$

$c[i,j] = c[i-1,j-1] + 1,$ if $i,j>0$ and $x_i = y_j$

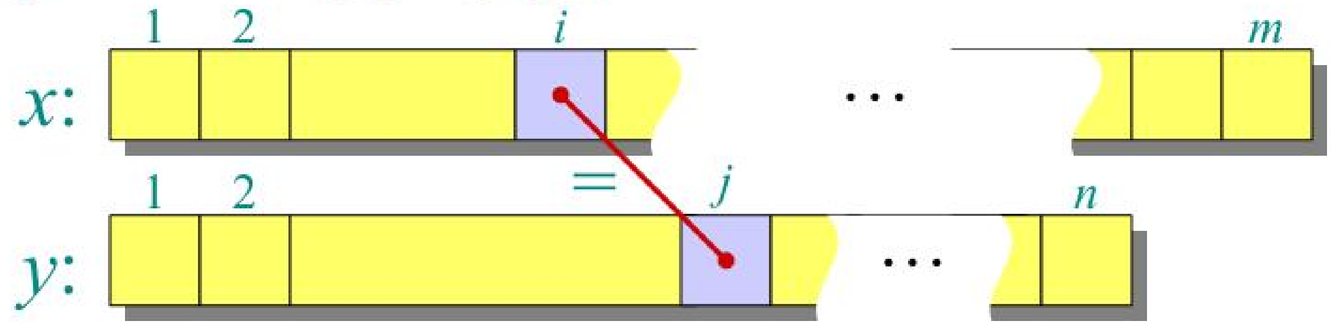
$c[i,j] = \max\{ c[i-1,j], c[i,j-1] \},$ if $i,j>0$ and $x_i \neq y_j$

Recursive formulation

Theorem.

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

Proof. Case $x[i] = y[j]$:



Let $z[1 \,.\,.\, k] = \text{LCS}(x[1 \,.\,.\, i], y[1 \,.\,.\, j])$, where $c[i, j] = k$. Then, $z[k] = x[i]$, or else z could be extended. Thus, $z[1 \,.\,.\, k-1]$ is CS of $x[1 \,.\,.\, i-1]$ and $y[1 \,.\,.\, j-1]$.

Proof (continued)

Claim: $z[1\,..\,k-1] = \mathrm{LCS}(x[1\,..\,i-1], y[1\,..\,j-1])$.
Suppose w is a longer CS of $x[1\,..\,i-1]$ and $y[1\,..\,j-1]$, that is, $|w| > k-1$. Then, ***cut and paste***: $w \,||\, z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1\,..\,i]$ and $y[1\,..\,j]$ with $\left| w \,||\, z[k] \right| > k$. Contradiction, proving the claim.

Thus, $c[i-1, j-1] = k-1$, which implies that $c[i, j] = c[i-1, j-1] + 1$.

Other cases are similar.

Dynamic-programming hallmark #1

Optimal substructure
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If $z = \mathrm{LCS}(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.

Recursive algorithm for LCS

$$
\begin{aligned}
&\text{LCS}(x, y, i, j)\\
&\quad \textbf{if } x[i] = y[j]\\
&\qquad \textbf{then } c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1\\
&\qquad \textbf{else } c[i, j] \leftarrow \max\{\text{LCS}(x, y, i-1, j),\\
&\qquad\qquad\qquad\qquad\qquad\quad \text{LCS}(x, y, i, j-1)\}
\end{aligned}
$$

Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

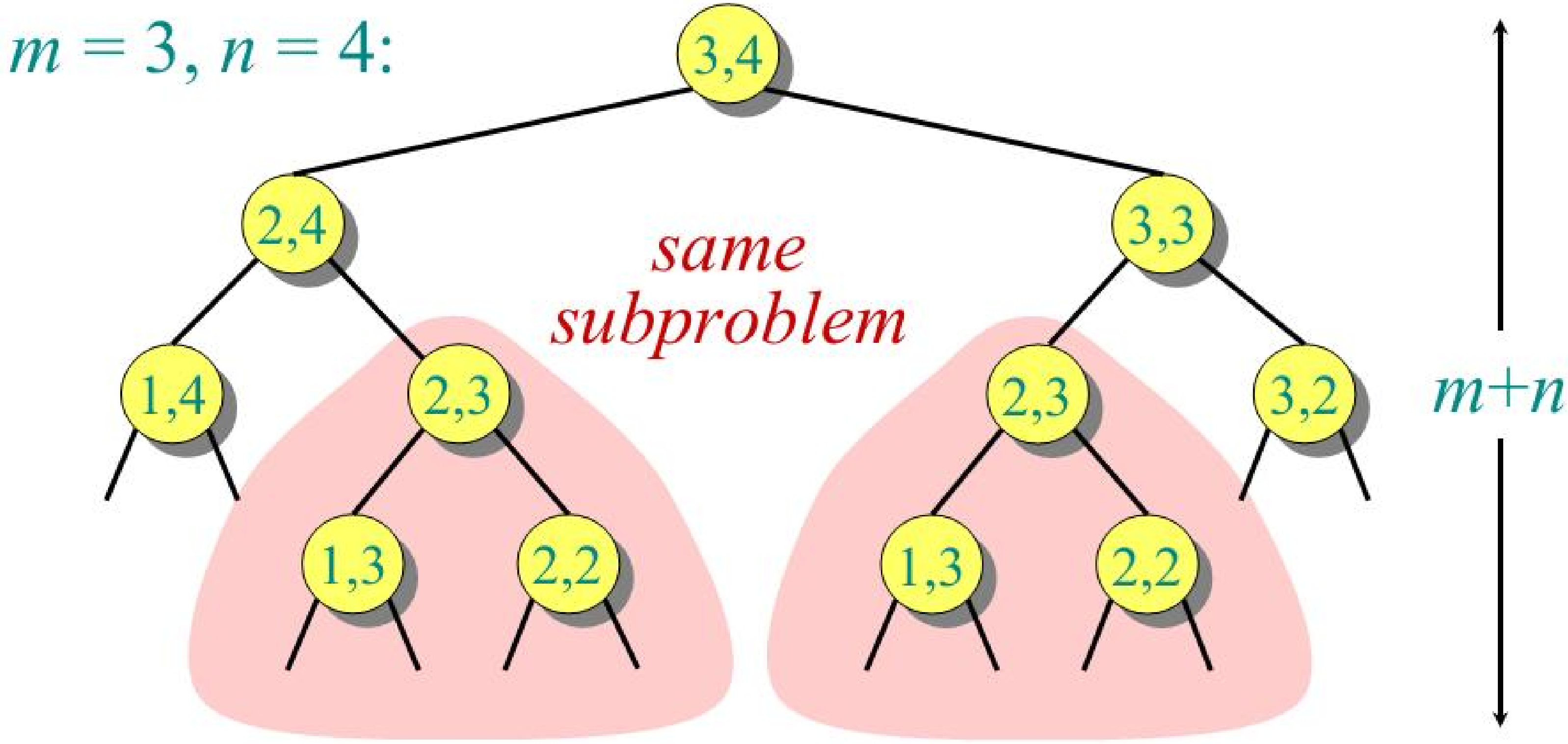
Dynamic-programming hallmark #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths m and n is only mn.

Recursion tree



Height $= m + n$
but we're solving subproblems already solved!

Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

LCS(x, y, i, j)
  if $c[i, j] \neq$ NIL
    then if $x[i] = y[j]$
      then $c[i, j] \leftarrow$ LCS$(x, y, i–1, j–1) + 1$
      else $c[i, j] \leftarrow \max\{LCS(x, y, i–1, j)$,
          LCS$(x, y, i, j–1)\}$

same as before

Time $= \Theta(mn) =$ constant work per table entry.
Space $= \Theta(mn)$.

LCS Example

X=<A,C,B,D>

Y=<B,A,C,D>

```
m ← length[X]
n ← length[Y]
For i = 1 to m
    do c[i,0] ← 0
For j = 0 to n
    do c[0,j] ← 0
For i = 1 to m
    for j = 1 to n
        if x_i = y_j
            then c[i,j]←c[i-1,j-1]+1
            else if c[i-1,j] ≥ c[i,j-1]
                then c[i,j] ← c[i-1,j]
                else c[i,j] ← c[i,j-1]
```

i \ j	0	B 1	A 2	C 3	D 4
0 0	0	0	0	0	0
A 1	0				
C 2	0				
B 3	0				
D 4	0				

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 \\ \max\{c[i-1,j], c[i,j-1]\} \end{cases}$$

LCS Example

X=<A,C,B,D>

Y=<B,A,C,D>

m ← length[X]
n ← length[Y]
For i = 1 to m
    do c[i,0] ← 0
For j = 0 to n
    do c[0,j] ← 0
For i = 1 to m
    for j = 1 to n
        if $x_i = y_j$
            then c[i,j]←c[i-1,j-1]+1
        else if c[i-1,j] ≥ c[i,j-1]
            then c[i,j] ← c[i-1,j]
        else c[i,j] ← c[i,j-1]

i \ j	0	B 1	A 2	C 3	D 4
0	0	0	0	0	0
A 1	0	0	**1**	1	1
C 2	0				
B 3	0				
D 4	0				

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 \\ \max\{c[i-1,j],\ c[i,j-1]\} \end{cases}$$

X=<A,C,B,D>

Y=<B,A,C,D>

LCS Example

m ← length[X]
n ← length[Y]
For i = 1 to m
    do c[i,0] ← 0
For j = 0 to n
    do c[0,j] ← 0
For i = 1 to m
    for j = 1 to n
        if $x_i = y_j$
            then c[i,j]←c[i-1,j-1]+1
        else if c[i-1,j] ≥ c[i,j-1]
            then c[i,j] ← c[i-1,j]
        else c[i,j] ← c[i,j-1]

i \ j	0	B 1	A 2	C 3	D 4
0	0	0	0	0	0
A 1	0	0	**1**	1	1
C 2	0	0	1	**2**	2
B 3	0	**1**	1	2	2
D 4	0	1	1	2	**3**

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 \\ \max\{c[i-1,j],\, c[i,j-1]\} \end{cases}$$

What is the LCS?

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

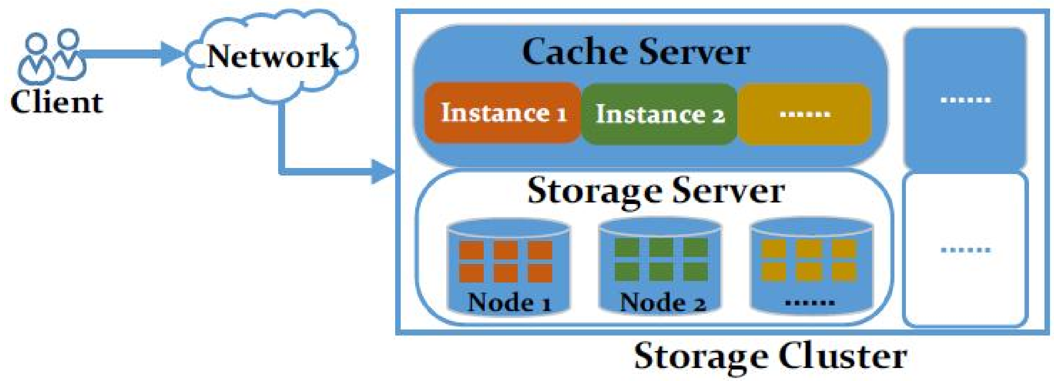
Space = $\Theta(mn)$.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

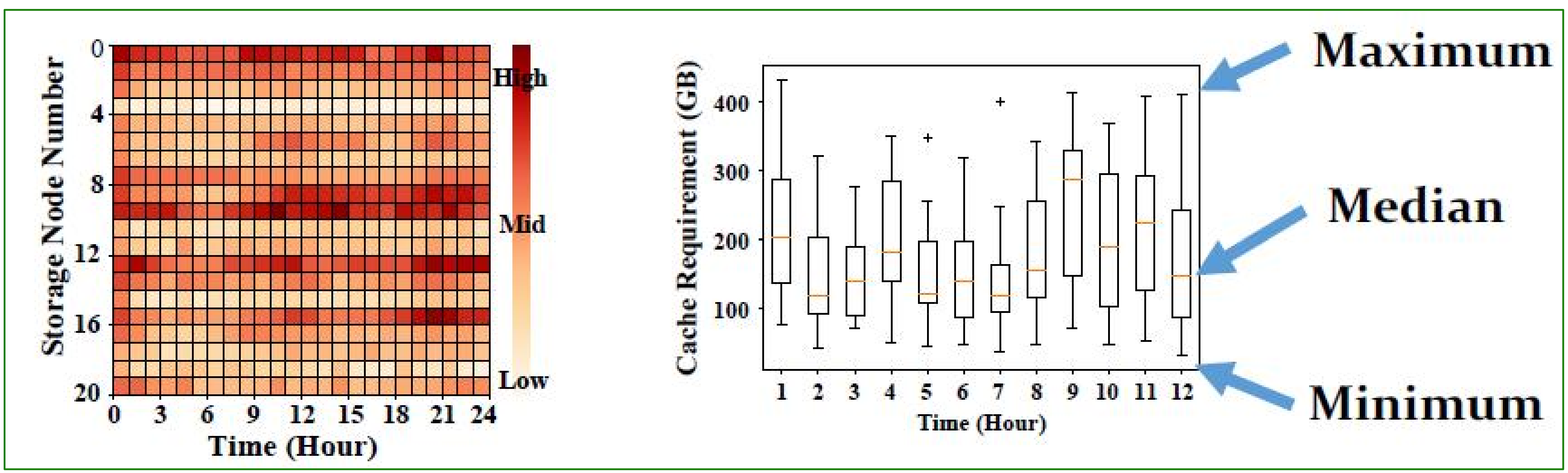
BCBA ! **BDAB ?**

拓展：Application of DP

一项国际最新研究 (腾讯CBS系统的缓存优化）为例：



Cloud Block System



Research Motivation

拓展：Application of DP

OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems

Online Cache Modeling

$O(\log n) \rightarrow O(1)$

- Obtain the miss *ratio curve*, which indicates the miss ratio corresponding to different cache sizes.

Optimization Target Defining

- Define an optimization target.

Searching for Optimal Configuration

DP

- Based on the cache modeling and defined target mentioned above, our OSCA searches for the optimal configuration scheme.

作业

1）15.2-1

2）15.4-1

Thank You!

Q&A