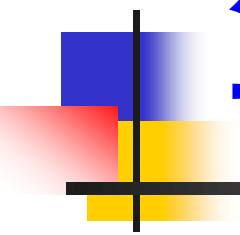


算法分析与设计

Analysis and Design of Algorithm

Lesson 07

第二章要点回顾

- 递归算法
  - 概念（阶乘、Fibonacci数列、双递归）
  - 例子（整数划分问题、Hanoi塔问题）
  - Hanoi塔算法、运行轨迹、分析时间复杂度
  - 递推方程（迭代法求解）
  - 递归的优缺点
- 分治策略
  - 基本思想、适用条件、基本步骤
  - 分治效率分析：给出了五种计算方法
  - 范例学习：二分搜索、大整数乘法、Strassen矩阵乘法、合并排序、快速排序、最近点对问题（自学）

第三章 动态规划法

Dynamic Programming

学习要点

- 理解动态规划算法的概念
- 掌握动态规划算法的基本要素
  - 最优子结构性质
  - 重叠子问题性质
- 掌握设计动态规划算法的步骤
  - 找出最优解的性质，并刻划其结构特征
  - 递归地定义最优值
  - 以自底向上的方式计算出最优值
  - 根据计算最优值时得到的信息，构造最优解
- 动态规划法应用实例

动态规划算法概述

概述

1. 最优化问题
2. 最优性原理
3. 动态规划法的设计思想

1.最优化问题

- 有n个输入（解空间），问题的解由这n个输入的一个子集组成，这个子集必须满足某些事先给定的条件，这些条件称为约束条件，满足约束条件的解称为问题的可行解。
- 满足约束条件的可行解可能不只一个，为了衡量这些可行解的优劣，事先给出一定的标准，这些标准通常以函数的形式给出，这些标准函数称为目标函数，使目标函数取得极值（极大或极小）的可行解称为最优解。
- 这类问题就称为最优化问题。

例：找零钱问题

问题： 自动售货POS机要找给顾客数量最少的现金。

分析： 假定POS机中有n张面值为$p_i(1 \leqslant i \leqslant n)$的货币，用集合$P=\{p_1, p_2, \ldots, p_n\}$表示，如果POS机需找的现金为$A$，那么，必须从$P$中选取一个最小子集$S$，使得：

$$p_i \in S \quad , \quad \sum_{i=1}^{m} p_i = A \quad (m = |S|) \qquad \text{式(1)}$$

如果用向量$X=(x_1, x_2, \ldots, x_n)$，表示$S$中所选取的货币，则：

$$x_i = \begin{cases} 1 & p_i \in S \\ 0 & p_i \notin S \end{cases} \qquad \text{式(2)}$$

例：找零钱问题

那么，POS机找的现金必须满足 $\sum_{i=1}^{n} x_i p_i = A$ 式(3)

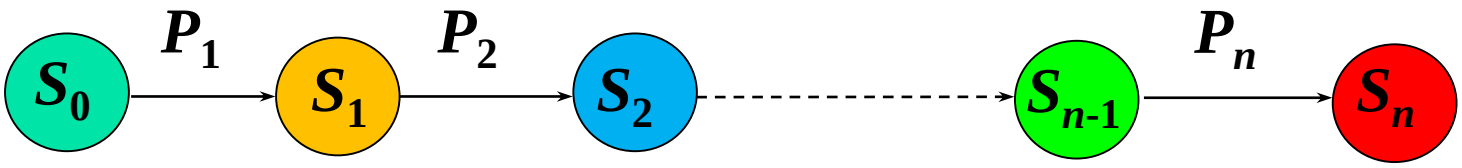
并且 $d = \min \sum_{i=1}^{n} x_i$ 式(4)

在找零钱问题中，集合*P*是该问题的**输入**，那么：

- 满足式(1)的解称为**可行解**；
- 式(2)是解的表现形式(因为向量*X*中有*n*个元素，每个元素的取值为0或1，所以，可以有2^n个不同的向量，所有这些向量的全体构成该问题的**解空间**)；
- 式(3)是该问题的**约束条件**；
- 式(4)是该问题的**目标函数**；
- 使式(4)取得极小值的解称为该问题的**最优解**。

2.最优性原理

对于一个具有n个输入的最优化问题，其求解过程往往可以划分为若干个阶段，每一阶段的决策仅依赖于前一阶段的状态。从而，一个决策序列在不断变化的状态中产生。这个决策序列产生的过程称为多阶段决策过程。

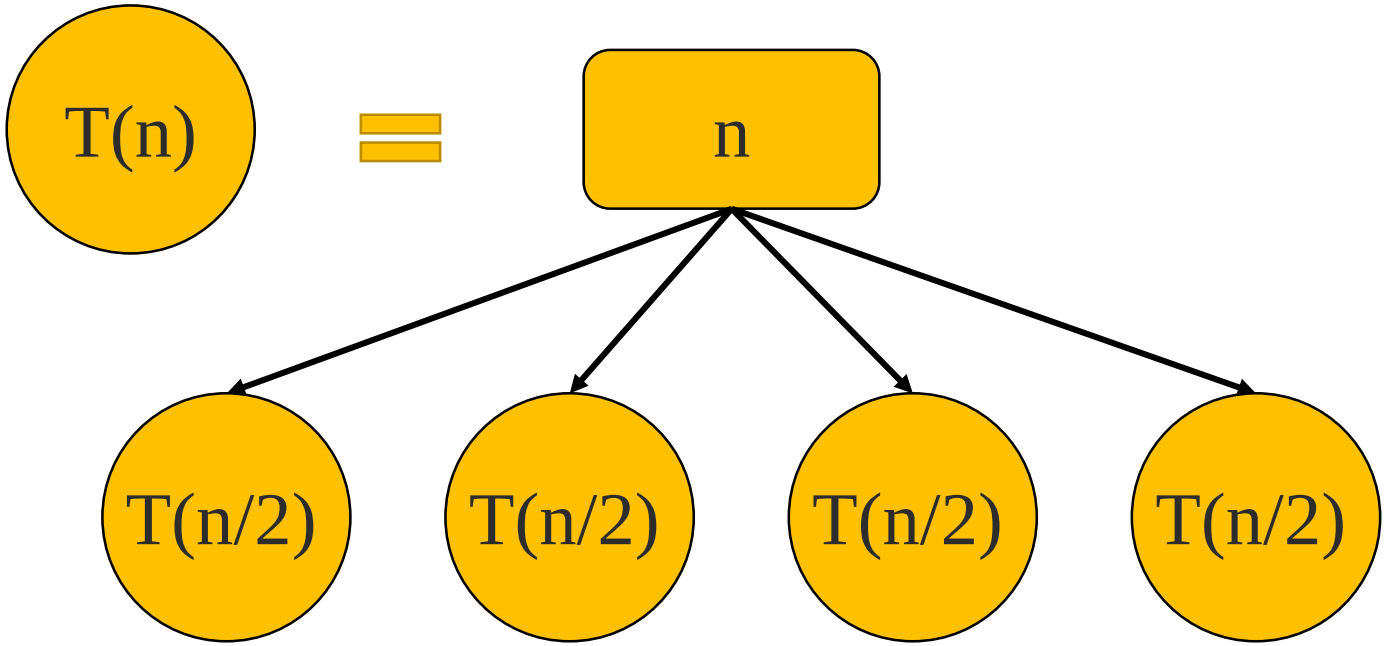


多阶段决策过程

多阶段决策过程满足最优性原理或者优化原则(Optimal Principle)：无论决策过程的初始状态和初始决策是什么，其余的决策都必须相对于初始决策所产生的当前状态，构成一个最优决策序列。 -----最优子结构性质。

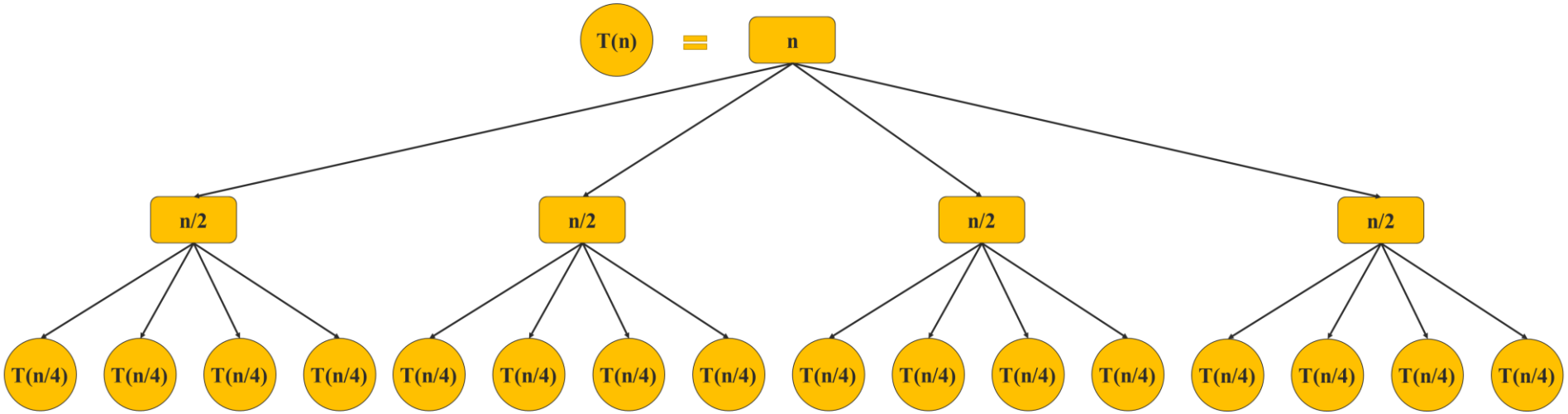
3.算法总体思想

- 动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题



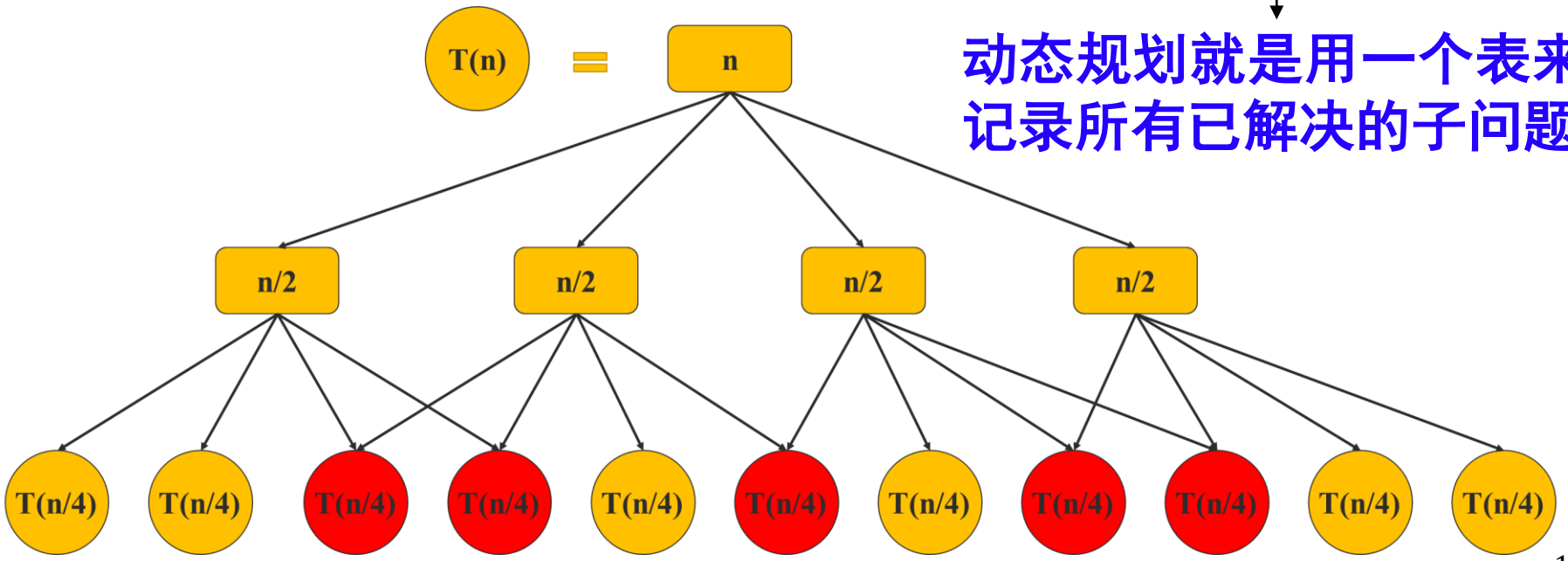
3.算法总体思想

- **与分治区别：经分解得到的子问题往往不是互相独立的。不同子问题的数目常常只有多项式量级。在用分治法求解时，有些子问题被重复计算了许多次。**



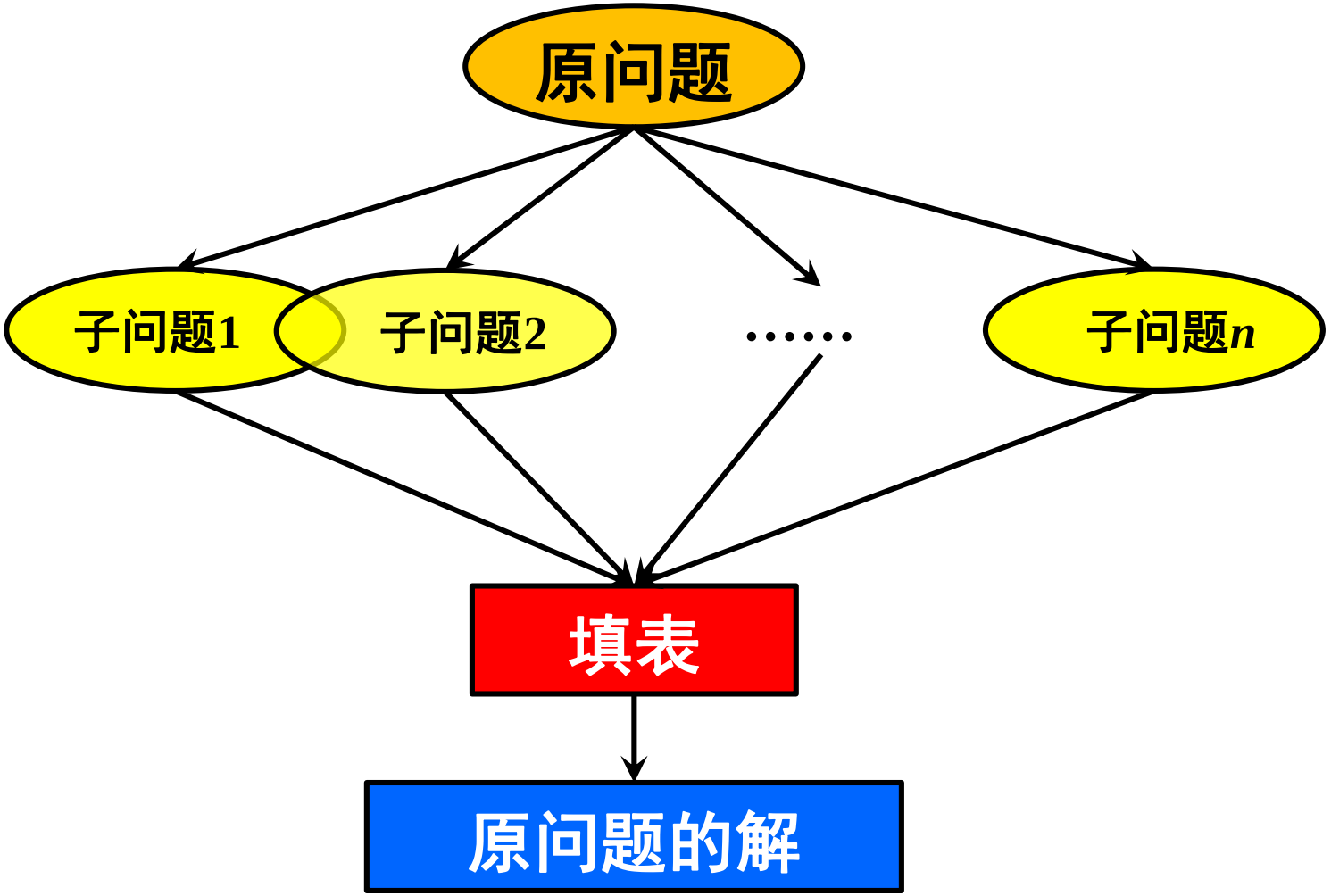
3.算法总体思想

- 用分治法求解，子问题的数目常有多项式量级，有些子问题被重复计算了多次，如果能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法。

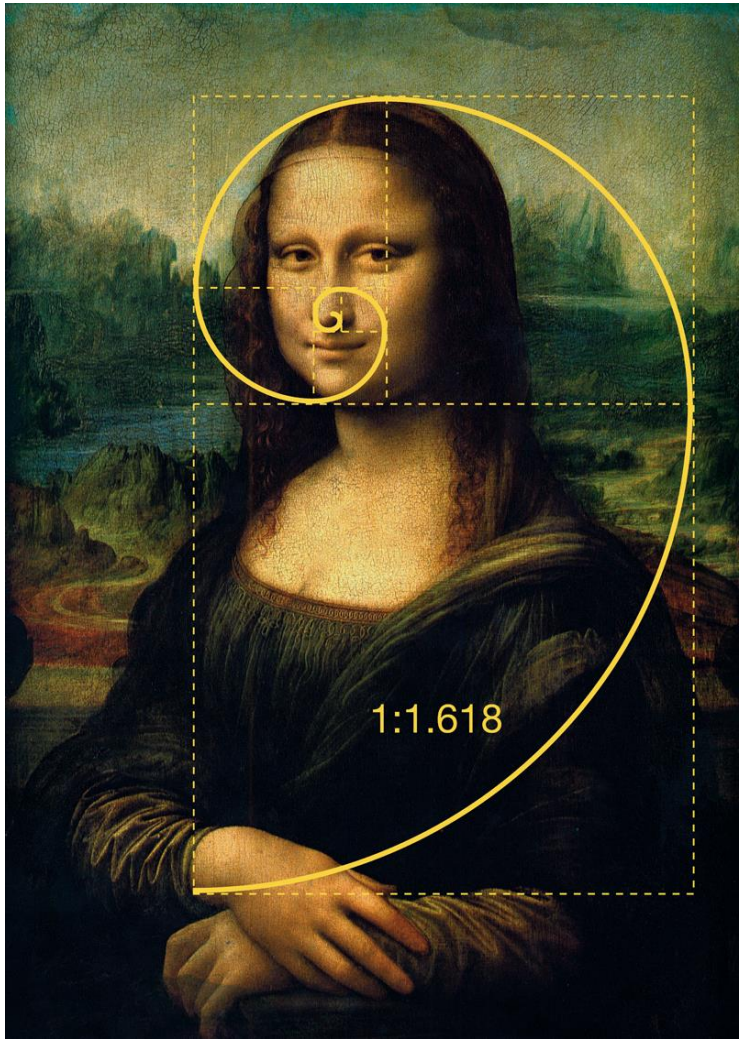


动态规划就是用一个表来记录所有已解决的子问题

动态规划法的求解过程

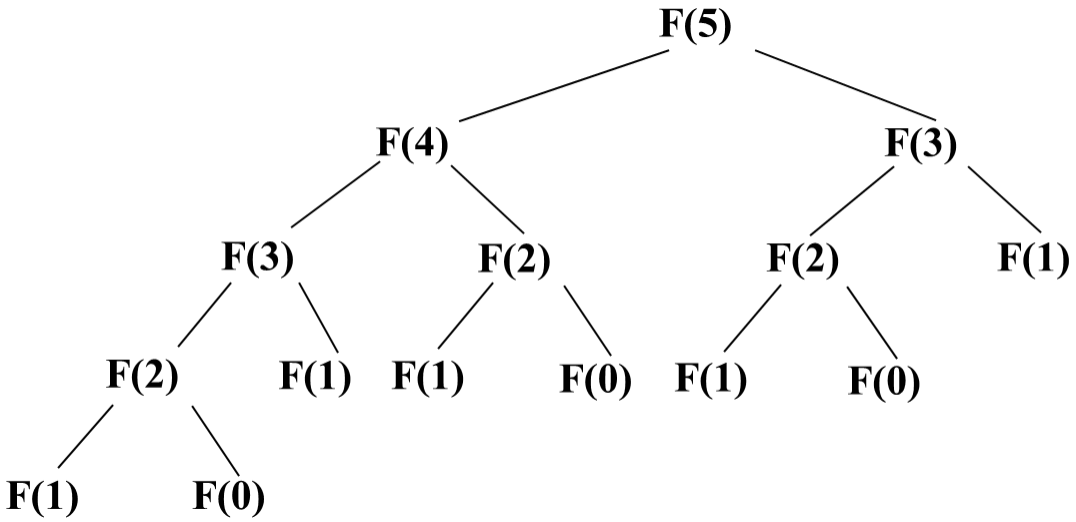


例：计算Fibonacci数列



$$F(n)=\begin{cases}1 & ,\ n=0\\ 1 & ,\ n=1\\ F(n-1)+F(n-2) & ,\ n>1\end{cases}$$

***n*=5时分治法计算斐波那契数的过程：**



例：计算Fibonacci数列

分析：注意到，计算$F(n)$是以计算它的两个重叠子问题$F(n\text{-}1)$和$F(n\text{-}2)$的形式来表达的，所以，可以设计一张表填入$n+1$个$F(n)$的值。

动态规划法求解斐波那契数$F(9)$的填表过程：

0	1	2	3	4	5	6	7	8	9
0	1	1	2	3	5	8	13	21	34

动态规划法注意事项

- 用动态规划法求解的问题具有特征：
  - 能够分解为相互重叠的若干子问题；
  - 满足最优性原理（也称最优子结构性质）：该问题的最优解中也包含着其子问题的最优解。
- 用反证法分析问题是否满足最优性原理：
  - 先假设由问题的最优解导出的子问题的解不是最优的；
  - 然后再证明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。

动态规划法基本步骤

动态规划法设计算法一般分成三个阶段：

1. 分段：将原问题分解为若干个相互重叠的子问题；
2. 分析：分析问题是否满足最优性原理或者优化原则，找出动态规划函数的递推式；
3. 求解：利用递推式自底向上计算，实现动态规划过程。

记住：动态规划法利用问题的最优性原理，以自底向上的方式从子问题的最优解逐步构造出整个问题的最优解。

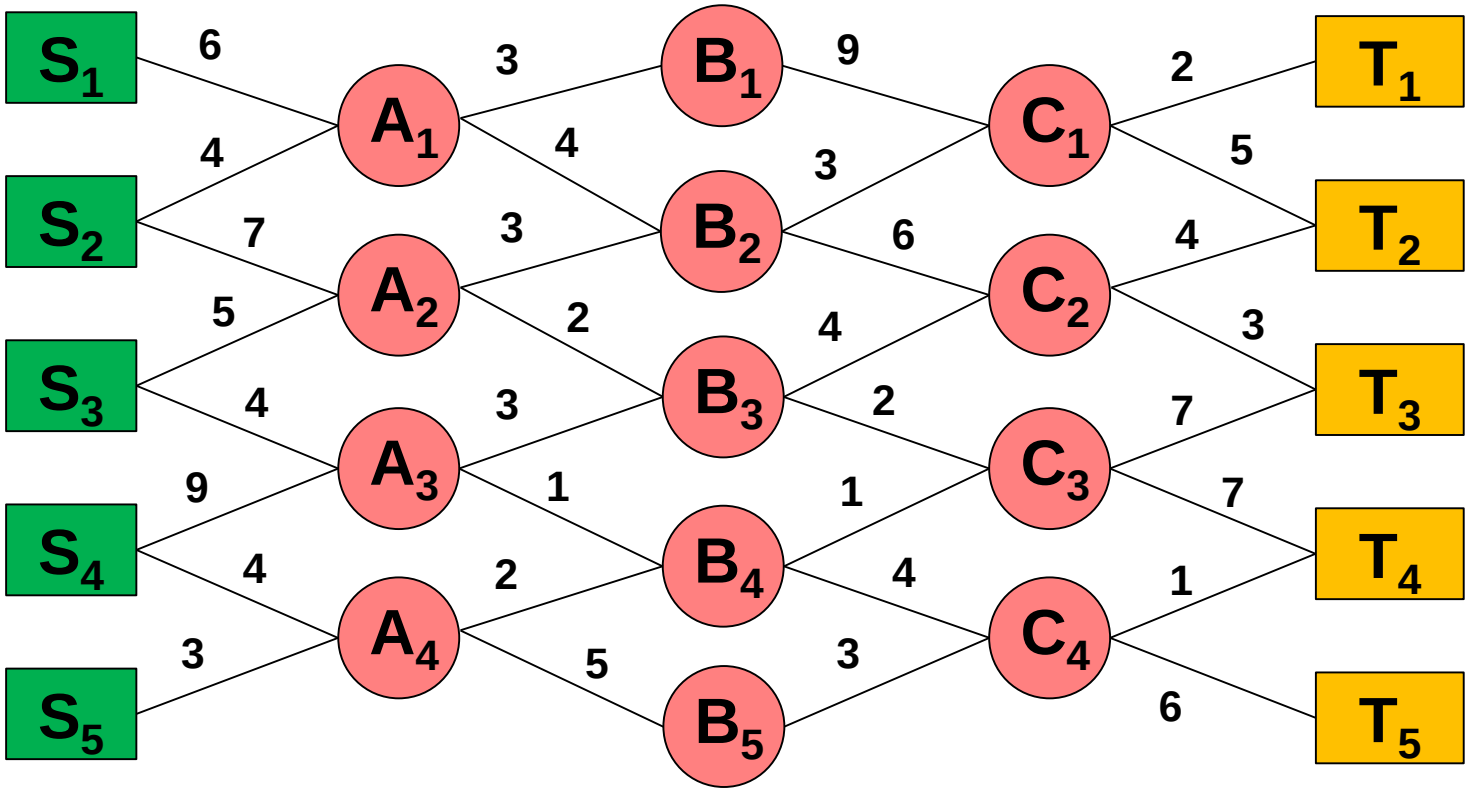
例：最短路径问题

- **问题：** 找任意起点到任意终点的一条最短路径

- **输入：**
  - 起点集合$\{S_1,S_2,\ldots,S_n\}$
  - 终点集合$\{T_1,T_2,\ldots,T_m\}$
  - 中间结点集合，边集，对于任意边e有长度
- **输出：** 一条从起点到终点的最短路

例：最短路径问题

- 一个实例

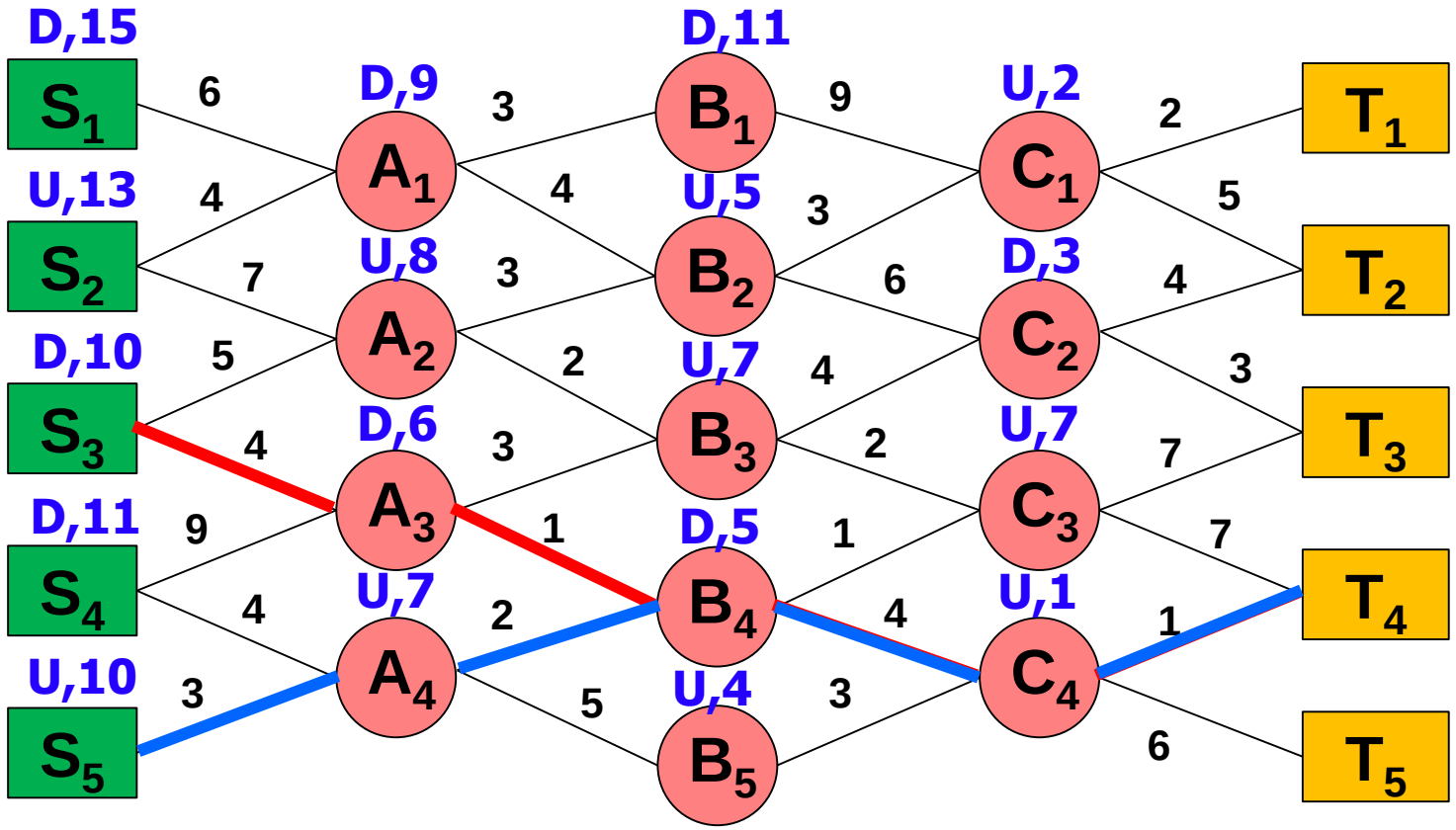


例：最短路径问题

- **蛮力算法/穷举法：**考察每一条从某个起点到某个终点的路径，计算长度，从中找出最短路径。
- 在上述实例中，如果网络的层数为k，那么路径条数将接近与2^k
- **动态规划算法：**多阶段决策过程。每一步求解的问题是后面阶段求解问题的子问题。每步决策将依赖于以前步骤的决策结果。

例：最短路径问题

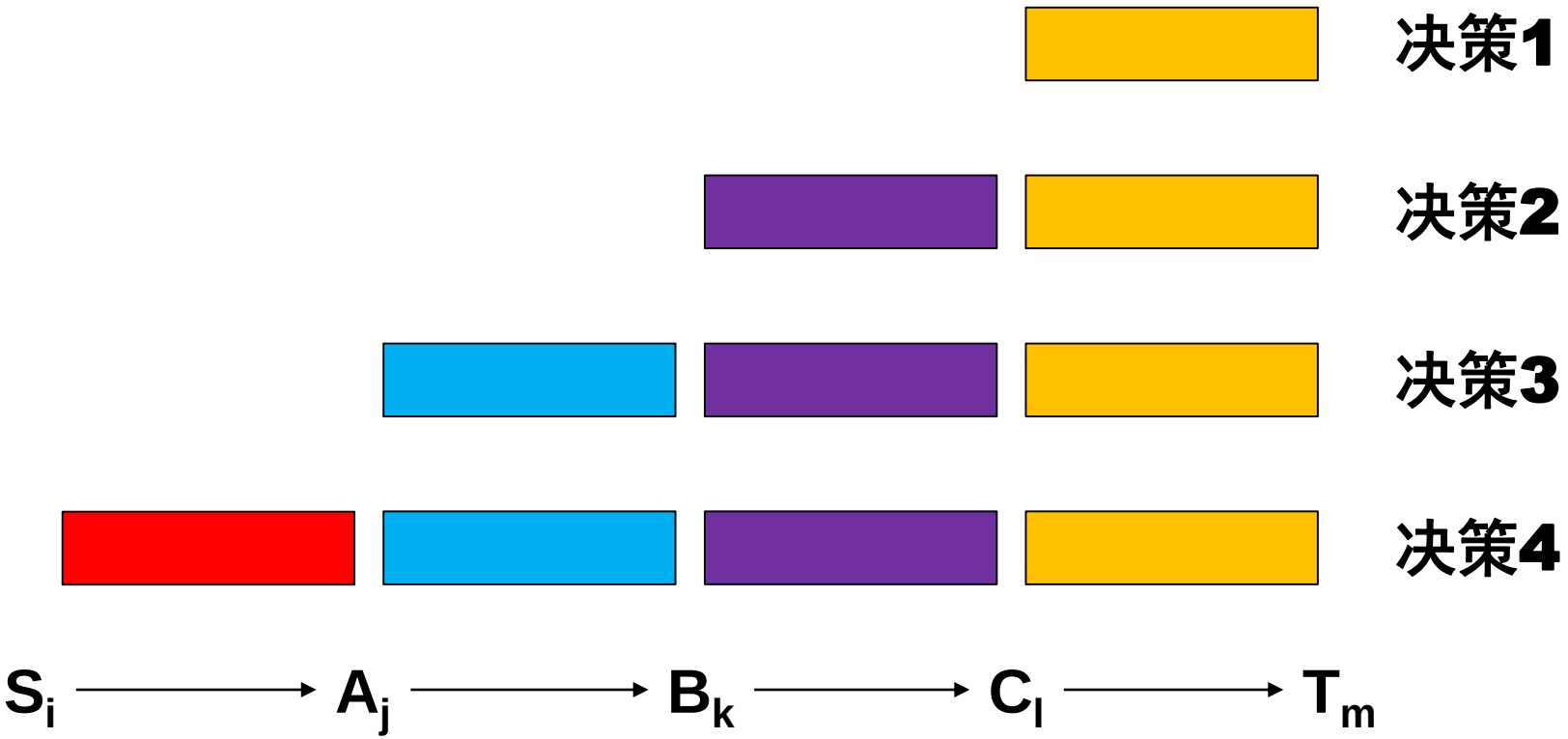
- 动态规划求解



例：最短路径问题

- 子问题界定

后边界不变，前边界前移



例：最短路径问题

- 最短路径的依赖关系

$$F(C_l)=\min_m\{C_lT_m\}$$ 决策1

$$F(B_k)=\min_l\{B_kC_l+F(C_l)\}$$ 决策2

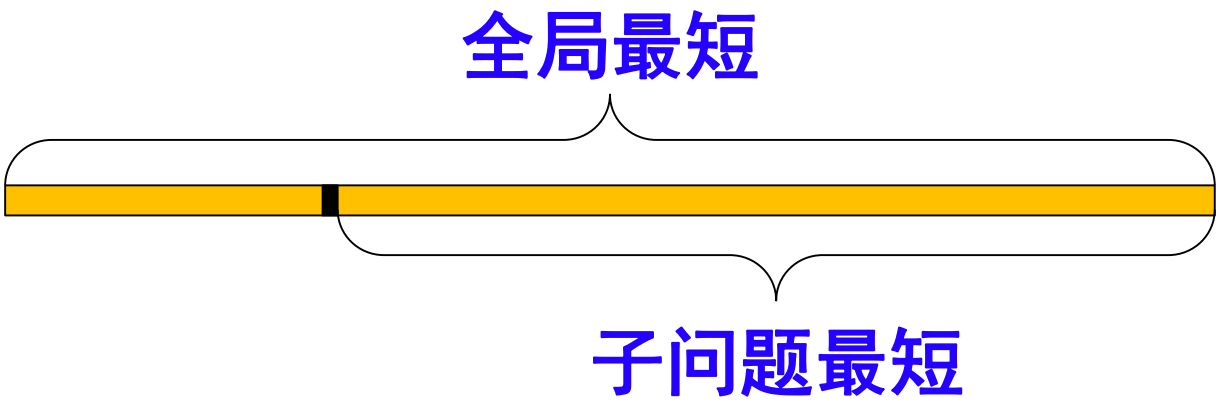
$$F(A_j)=\min_k\{A_jB_k+F(B_k)\}$$ 决策3

$$F(S_i)=\min_j\{S_iA_j+F(A_j)\}$$ 决策4

优化函数值之间存在依赖关系

优化原则：最优子结构性质

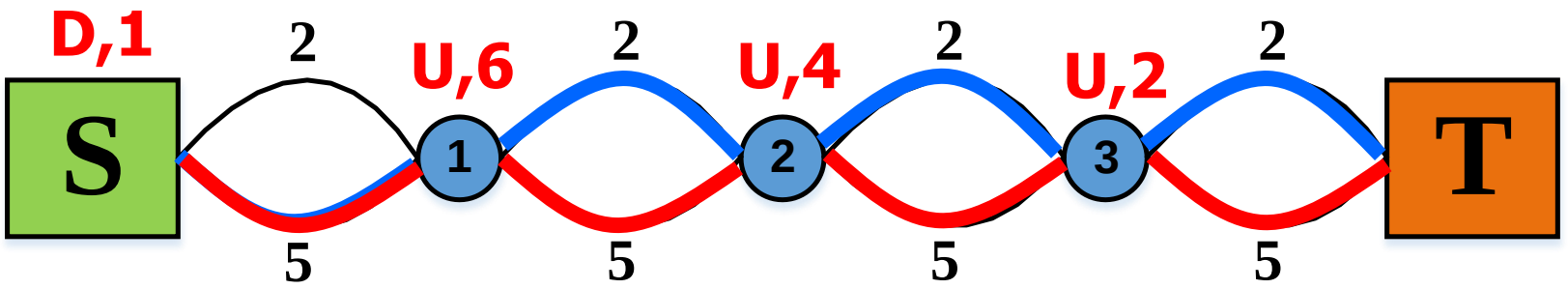
- **优化函数的特点：**任何最短路的子路径相对于子问题始、终点最短



- **优化原则：**一个最优决策序列的任何子序列本身一定是相对于子序列的初始和结束状态的最优决策序列

一个反例

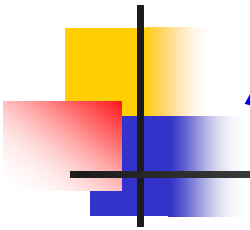
- 求总长模10的最小路径



➔动态规划算法的解：下、上、上、上

- 最优解：下、下、下、下

不满足优化原则，不能用动态规划！！！



小结

动态规划(Dynamic Programming)

- 求解过程是多阶段决策过程，每步处理一个子问题，可用于求解组合优化问题
- 适用条件：问题要满足优化原则或者最优子结构性质，即：一个最优决策序列的任何子序列本身一定是相对于子序列的初始和结束状态的最优决策序列

动态规划算法设计

动态规划设计要素

1. 问题建模，优化的目标函数是什么？约束条件是什么？
2. 如何划分子问题？（边界）
3. 问题的优化函数值与子问题的优化函数值存在着什么依赖关系？（递推方程）
4. 是否满足优化原则/最优子结构性质？
5. 最小子问题怎样界定？其优化函数值，即初值等于什么？

完全加括号的矩阵连乘

完全加括号的矩阵连乘积可递归地定义为：

- **单个矩阵是完全加括号的；**
- **矩阵连乘积*A*是完全加括号的，则*A*可表示为2个完全加括号的矩阵连乘积*B*和*C*的乘积并加括号，即*A*=(*BC*)**

$$a[i][j]=\sum_{k=1}^{n} b[i][k]c[k][j]$$

例子：设有四个矩阵*A*,*B*,*C*,*D*，它们的维数分别是：

$A=50\times10$ $B=10\times40$ $C=40\times30$ $D=30\times5$

总共有五种完全加括号的方式

$(A((BC)D))$ **16000**

$(A(B(CD)))$ **10500**

$((AB)(CD))$ **36000**

$(((AB)C)D)$ **87500**

$((A(BC))D)$ **34500**

矩阵连乘背景

- 给定*n*个矩阵$\{A_1, A_2, \ldots, A_n\}$，其中$A_i$与$A_{i+1}$是可乘的，$i = 1, 2, \ldots, n-1$。考察这*n*个矩阵的连乘积

$$A_1 A_2 \ldots A_n$$

- 由于矩阵乘法满足结合律，所以计算矩阵的连乘可以有许多不同的计算次序。这种计算次序可以用加括号的方式来确定。
- 若一个矩阵连乘积的计算次序完全确定，也就是说该连乘积已完全加括号，则可以依此次序反复调用2个矩阵相乘的标准算法计算出矩阵连乘积

矩阵连乘问题

问题： 给定n个矩阵$\{A_1,A_2,\ldots,A_n\}$，其中A_i为$P_{i-1}\times P_i$阶矩阵，$i=1，2\ldots，n$。

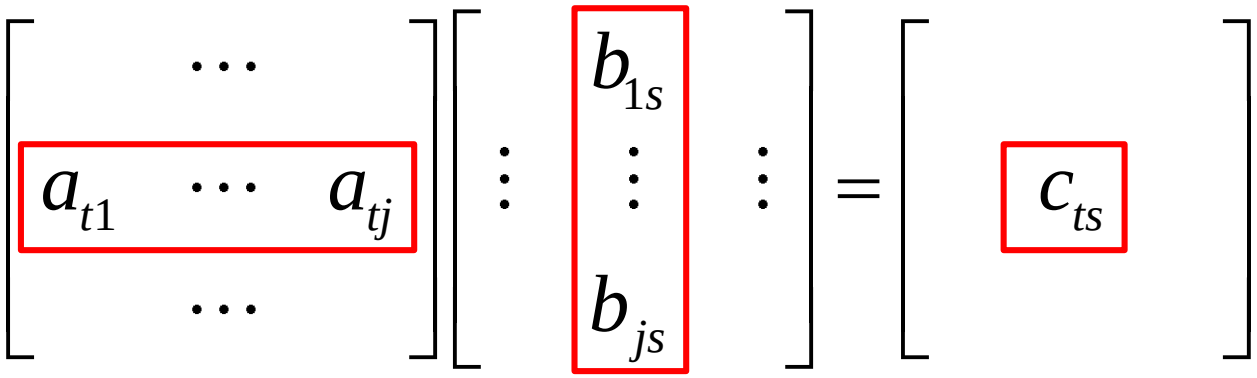
试确定计算矩阵连乘积的计算次序，使得矩阵链相乘需要的总次数最少。

输入： 向量$P=<P_0,P_1,\ldots,P_n>$，其中P_0，P_1，…，P_n为n个矩阵的行数与列数。

输出： 矩阵链乘法加括号的位置。

矩阵相乘基本运算次数

- 矩阵A：*i*行*j*列，B：*j*行*k*列，以元素相乘作基本运算，计算AB的工作量



$$c_{ts} = a_{t1} \times b_{1s} + \cdots + a_{tj} \times b_{js}$$

- AB：*i*行*k*列的矩阵，计算每个元素需要作*j*次乘法，总计乘法次数为：*ik*×*j*=*ijk*

矩阵连乘问题

实例： $P=<10, 100, 5, 50>$

A_1:10×100, A_2:100×5, A_3:5×50。

乘法次序：

$(A_1A_2)\,A_3$: 10×100×5+10×5×50=7500

$A_1(A_2A_3)$: 10×100×50+100×5×50=75000

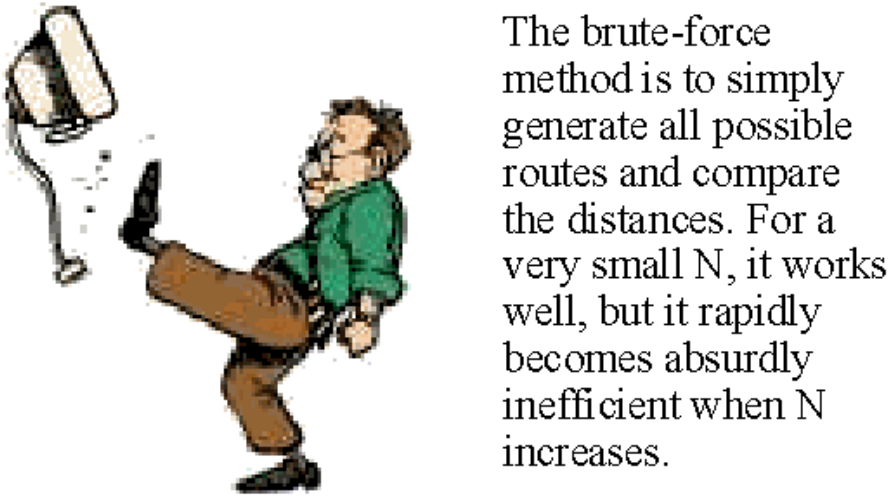
第一种次序计算次数最少。

矩阵连乘问题

实例： $P=<25, 2, 40, 15, 30>$

A:25×2, B:2×40, C:40×15, D:15×30。

穷举法：列举出所有可能的计算次序，并计算出每一种计算次序相应需要的数乘次数，从中找出一种数乘次数最少的计算次序。



矩阵连乘问题

实例： $P=<25, 2, 40, 15, 30>$

A:25×2, B:2×40, C:40×15, D:15×30。

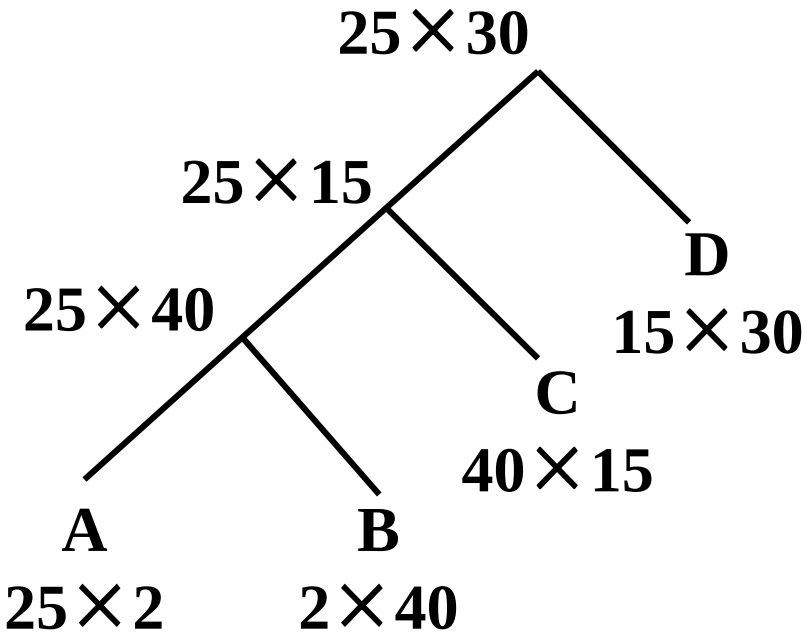
① (((AB)C)D)

= 25×2×40

+ 25×40×15

+ 25×15×30

= 28 250



矩阵连乘问题

实例： $P=<25, 2, 40, 15, 30>$

A:25×2, B:2×40, C:40×15, D:15×30。

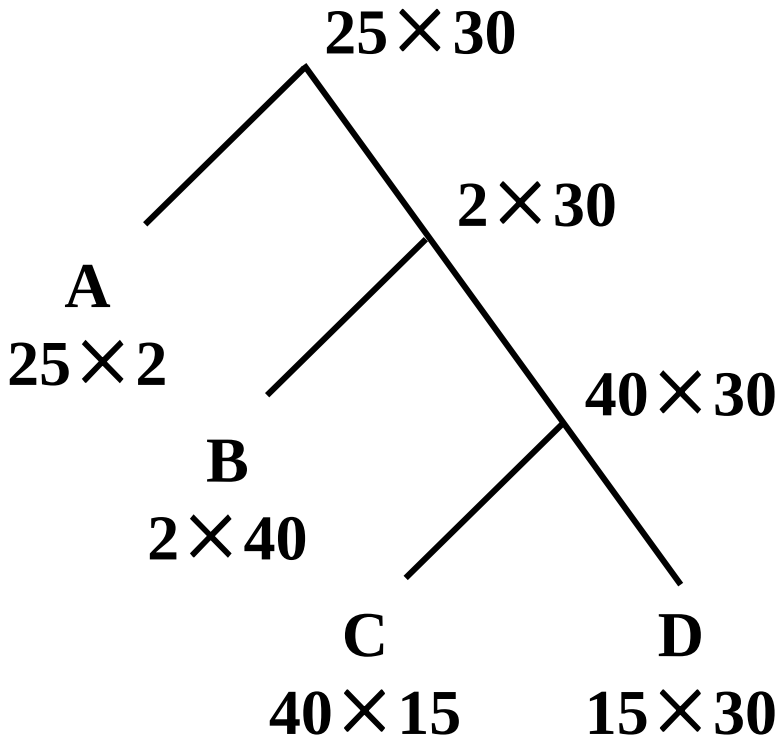
② (A(B(CD)))

= 40×15×30

+ 2×40×30

+ 25×2×30

= 21 900



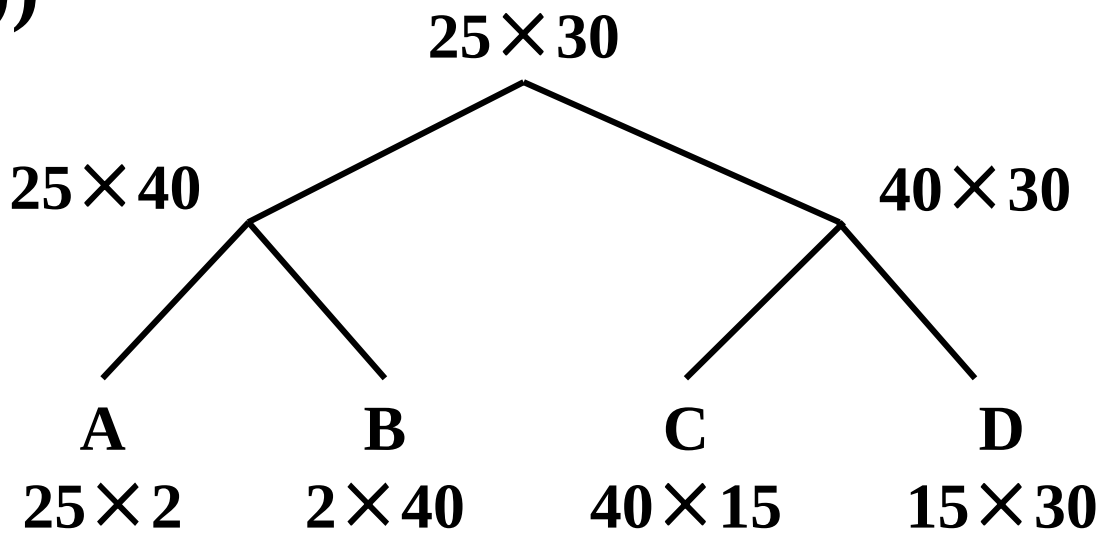
矩阵连乘问题

实例： P=<25, 2, 40, 15, 30>

A:25×2, B:2×40, C:40×15, D:15×30。

③ ((AB)(CD))

= 50 000



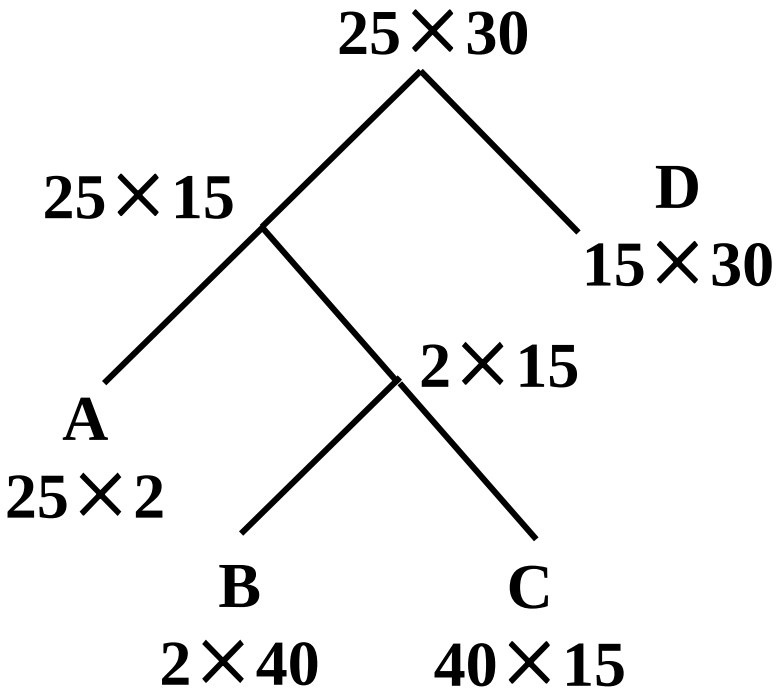
矩阵连乘问题

实例： $P=<25, 2, 40, 15, 30>$

A:25×2, B:2×40, C:40×15, D:15×30。

④ ((A(BC))D)

= 13 200



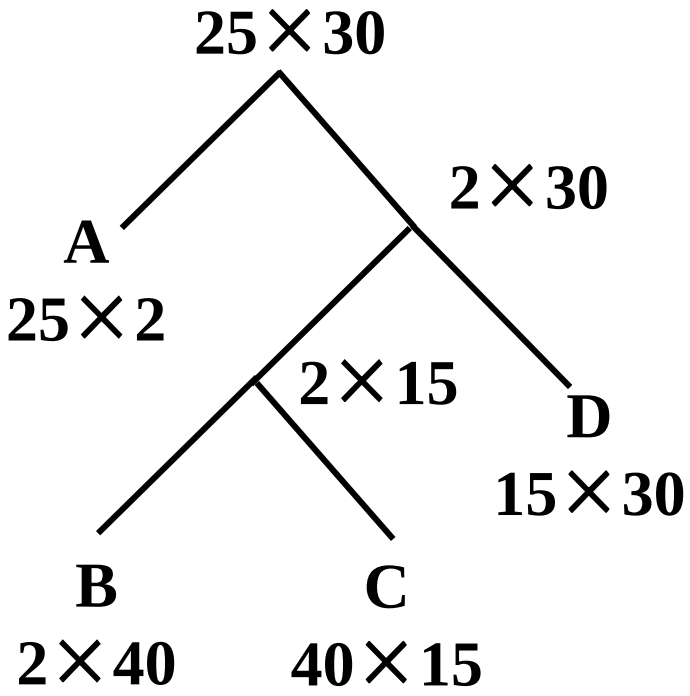
矩阵连乘问题

实例： P=<25, 2, 40, 15, 30>

A:25×2, B:2×40, C:40×15, D:15×30。

⑤ (A((BC)D))

= 3600



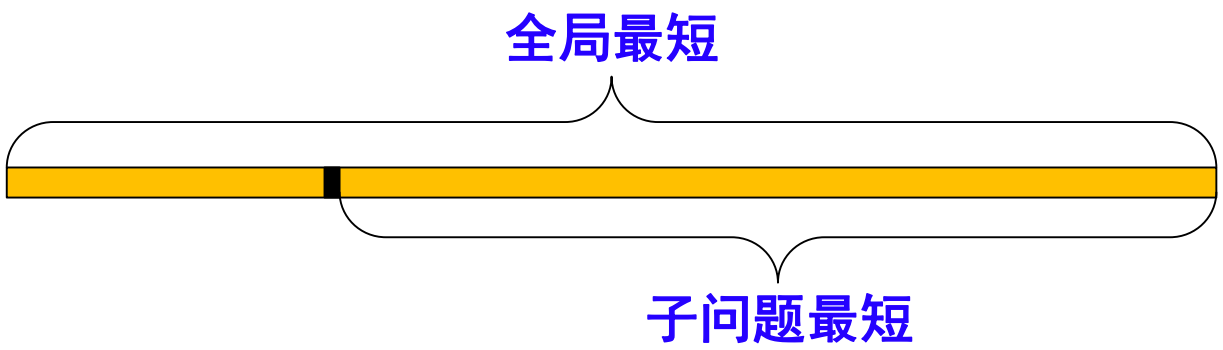
算法分析与设计

Analysis and Design of Algorithm

Lesson 08

要点回顾

- 动态规划概述
  - 最优化问题
    - 概念：约束条件、可行解、目标函数、最优解
    - 例子：POS机找零钱
  - 最优性原理/优化原则
    - 最优子结构性质
    - 反例：总长模10的最小路径



要点回顾

- 动态规划算法的设计要点
  - 建模：目标函数、约束条件
  - 分段：确定子问题的边界
  - 分析：原始问题与子问题之间的依赖关系
  - 判断：最优子结构性质
  - 求解：先定最小子问题（初值），自底向上求解（疑问）
- 实例：矩阵链相乘问题
  - 目标：加括号求出矩阵链相乘的最小乘法次数
  - 穷举法（又名暴力算法，二叉树求解）

矩阵连乘问题穷举法

复杂度分析

- 对于n个矩阵的连乘积，设其不同的计算次序为$P(n)$。
- 由于每种加括号方式都可以分解为两个子矩阵的加括号问题：$(A_1...A_k)(A_{k+1}...A_n)$可以得到关于$P(n)$的递推式如下：

$$P(n)=\begin{cases}1 & n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & n>1\end{cases}$$

Catalan数　即：$P(n)=C(n-1)=\frac{1}{(n-1)+1}\binom{2(n-1)}{n-1}$

矩阵连乘问题穷举法

- Stirling公式计算Catalan数
- 不失一般性，假设：

$$T(n)=\frac{1}{n+1}\times\binom{2n}{n}=\Omega\left(\frac{1}{n+1}\times\frac{(2n)!}{n!\times n!}\right)$$

$$=\Omega\left(\frac{1}{n+1}\times\frac{\sqrt{2\pi 2n}\left(\frac{2n}{e}\right)^{2n}}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}\times\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}}\right)=\Omega\left(\frac{2^{2n}}{n^{\frac{3}{2}}}\right)$$

复杂度是指数量级！！！

矩阵连乘问题动态规划法

1. 分段：子问题划分

将矩阵连乘积 $A_iA_{i+1}\ldots A_j$ 简记为$A[i:j]$，这里$i\leqslant j$
输入向量：$<P_{i-1},P_i,\ldots,P_j>$
其最好划分的运算次数：$m[i,j]$

2. 分析：子问题的依赖关系

考察计算$A[i:j]$的最优计算次序。设这个计算次序在矩阵A_k和A_{k+1}之间将矩阵链断开，$i\leqslant k<j$，即最优划分最后一次相乘发生在矩阵k的位置，则其相应完全加括号方式为 $(A_iA_{i+1}\ldots A_k)(A_{k+1}A_{k+2}\ldots A_j)$

矩阵连乘问题动态规划法

递推方程：

$m[i, j]$：得到$A[i:j]$的最少的相乘次数。可递归定义为：

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+P_{i-1}P_kP_j\} & i<j\end{cases}$$

$(A_iA_{i+1}\cdots A_k)$ $(A_{k+1}A_{k+2}\cdots A_j)$

$P_{i-1}\times P_k$ $P_k\times P_j$

该问题满足优化原则！

递归实现动态规划的部分伪码

- 算法1： RecurMatrixChain(P, i, j)

```
1.  m[i,j]←∞
2.  s[i, j]←i
3.  for k←i to j-1 do
4.      q←RecurMatrixChain(P, i, k)
          + RecurMatrixChain(P, k+1, j) + P_{i-1}P_kP_j
5.      if q < m[i,j]
6.      then m[i,j]←q
7.           s[i,j]←k
8.  return m[i,j]
```

划分位置k

子问题i:j

找到了更好的解

这里没有写出算法的全部描述（递归边界）

算法分析

- **时间复杂度的递推方程**

$$T(n) \geq \begin{cases} O(1) & n=1 \\ \sum_{k=1}^{n-1}(T(k)+T(n-k)+O(1)) & n>1 \end{cases}$$

$$T(n) \geq O(n) + \sum_{k=1}^{n-1} T(k) + \sum_{k=1}^{n-1} T(n-k)$$

$$T(n) \geq O(n) + 2\sum_{k=1}^{n-1} T(k)$$

可以证明还是一个指数函数，即

- $T(n) \geqslant 2^{n-1}$

矩阵连乘问题动态规划法

递归实现动态规划算法的问题：

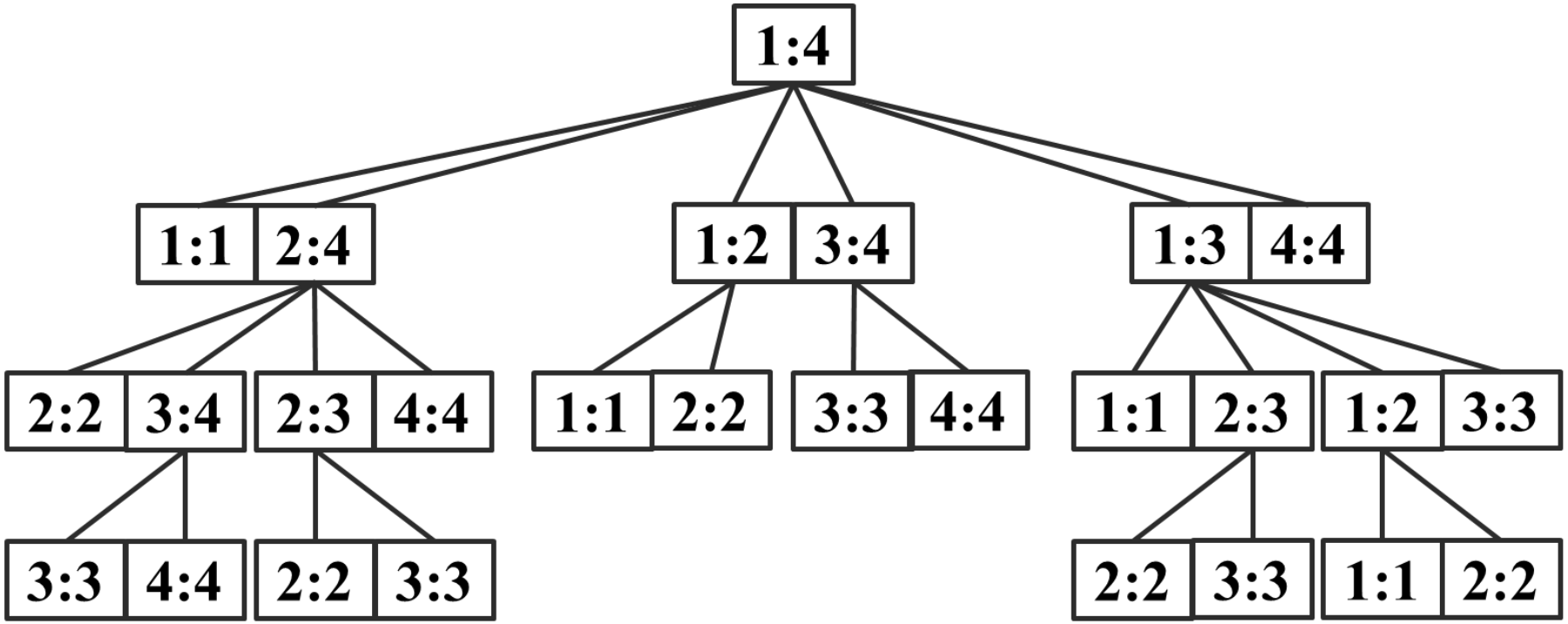
- 对于$1 \leqslant i \leqslant j \leqslant n$不同的有序对$(i,j)$对应于不同的子问题。因此，不同子问题的个数最多只有

$$\binom{n}{2} + n = \Theta(n^2)$$

- 由此可见，在递归计算时，许多子问题被重复计算多次。这也是该问题可用动态规划算法求解的又一显著特征。

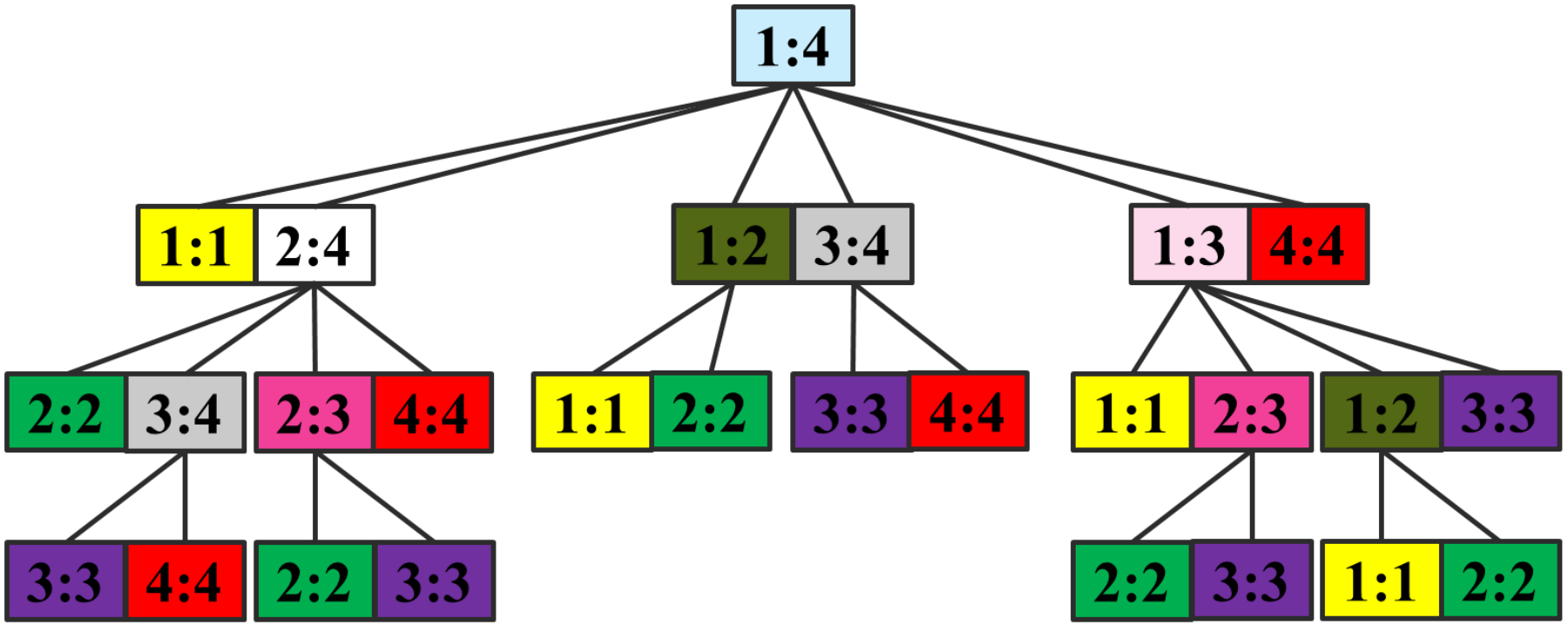
子问题的产生

用上述算法RecurMatrixChain(1,4)计算A[1:4]的递归树：



子问题的产生

用上述算法RecurMatrixChain(1,4)计算A[1:4]的递归树：



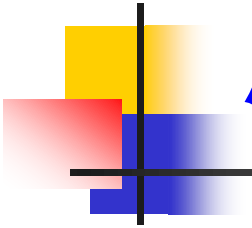
子问题的计数

边界	次数	边界	次数	边界	次数
1:1	4	1:2	2	1:3	1
2:2	5	2:3	2	2:4	1
3:3	5	3:4	2		
4:4	4			1:4	1

边界不同的子问题：**10**个

递归计算的子问题：**27**个

当n=5的时候，上面两个数值分别是15和81



小结

- 与穷举法相比较，矩阵连乘问题由于具有最优子结构性质和重复子问题的性质，可以用动态规划法
- 动态规划算法的直接递归实现效率并不高，原因在于同一子问题多次重复出现，每次出现都需要重新计算一遍

- 还有没有改进的空间？

动态规划算法的迭代实现

- **思想：**采用空间换时间策略，记录每个子问题首次计算结果，后面再用时就直接取值，每个子问题只计算一次！

迭代计算的关键

- 每个子问题只计算一次
- 迭代过程
  - 从最小的子问题算起
  - 考虑计算顺序，以保证后面用到的值前面已经计算好了
  - 存储结构保存计算结果——备忘录
- 解的追踪
  - 设计标记函数标记每步的决策
  - 考虑根据标记函数追踪解的算法

矩阵链乘法不同子问题

- **长度1：** 只含1个矩阵，有n个子问题（不需要计算）
- **长度2：** 含2个矩阵，n-1个子问题
- **长度3：** 含3个矩阵，n-2个子问题

…

- **长度n-1：** 含n-1个矩阵，2个子问题
- **长度n：** 原始问题，只有1个

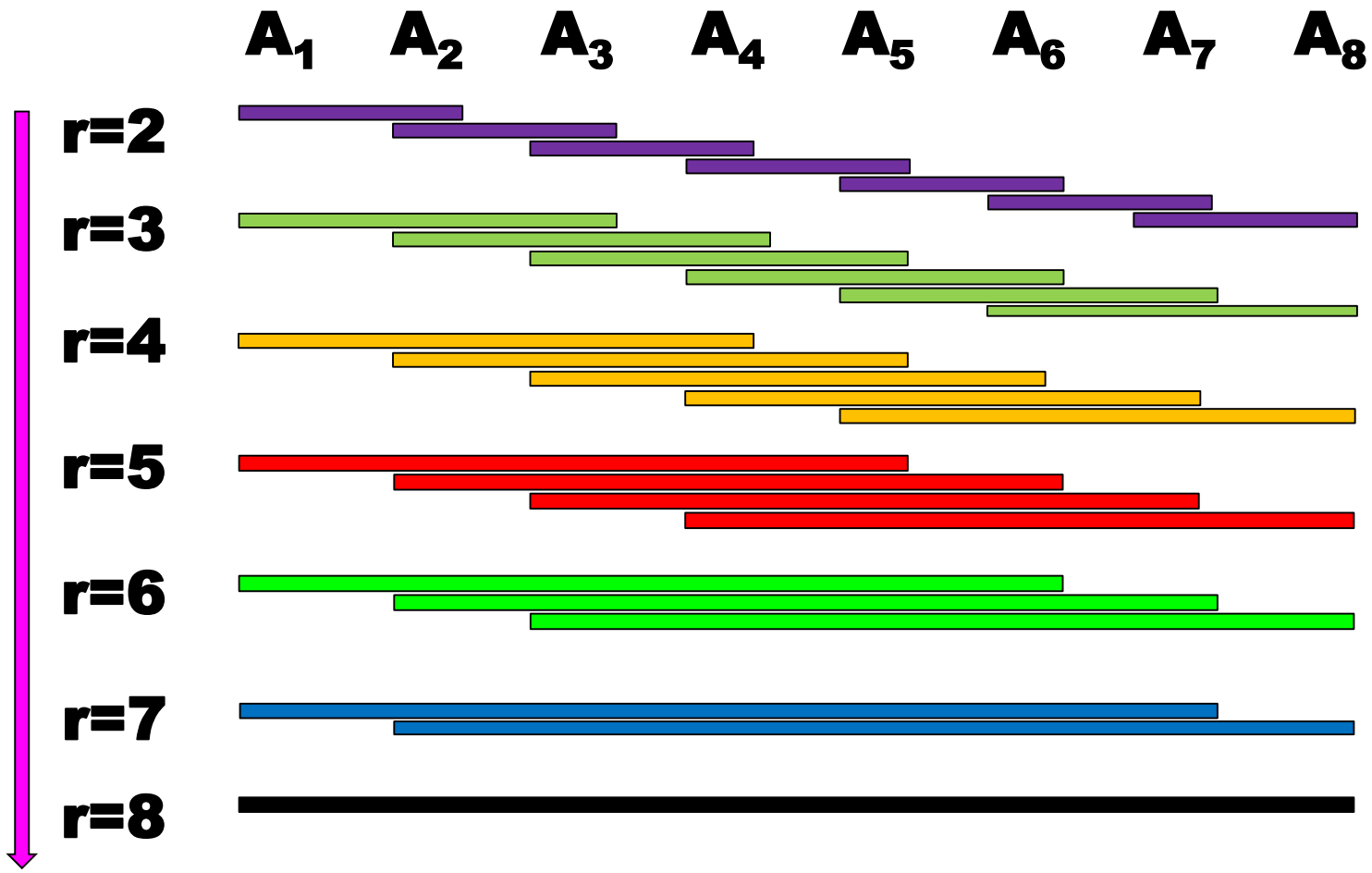
矩阵链乘法迭代顺序

- 长度为1：初值，$m[i,i]=0$
- 长度为2：1:2, 2:3, 3:4, …, n-1:n
- 长度为3：1:3, 2:4, 3:5, …, n-2:n

…

- 长度为n-1：1:n-1, 2:n
- 长度为n：1:n

n=8的子问题计算顺序



部分伪码

二维数组*m*与*s*为备忘录

- 算法MatrixChain(*P*, *n*)

```
1.  令所有的m[i,j]初值为0
2.  for r←2 to n do                    //r为链长
3.      for i←1 to n-r+1 do            //左边界i
4.          j← i+r-1                   //右边界j
5.          m[i,j]← m[i+1,j]+P_{i-1}P_iP_j      //k=i
6.          s[i,j]←i                   //记录k
7.          for k←i+1 to j-1 do        //遍历k
8.              t← m[i,k]+ m[k+1,j]+P_{i-1}P_kP_j
9.              if t< m[i,j]
10.             then m[i,j]←t          //更新解
11.                 s[i,j]←k
```

遍历长度为*r*子问题

遍历所有划分

时间复杂度

1、根据伪码：行2，3，7都是$O(n)$，循环执行$O(n^3)$次，内部为$O(1)$

$$T(n)= O(n^3)$$

2、根据备忘录：估计每项工作量，求和。子问题有$O(n^2)$个，确定每个子问题的最少乘法次数需要对不同划分位置比较，需要$O(n)$时间。

$$T(n)= O(n^3)$$

追踪解工作量$O(n)$，总工作量$O(n^3)$

实例

- **输入：** P=<30, 35, 15, 5, 10, 20>

  n=5

- **矩阵链：** $A_1 A_2 A_3 A_4 A_5$，其中

  A_1:30×35, A_2:35×15, A_3:15×5,

  A_4:5×10, A_5:10×20

- **备忘录：** 存储所有子问题的最小乘法次数

  标记函数： 得到这个值的划分位置，用于追踪解

备忘录 $m[i, j]$

- P=<30, 35, 15, 5, 10, 20>

r=1	m[1,1]= 0	m[2,2]=0	m[3,3]=0	m[4,4]=0	m[5,5]=0
r=2	m[1,2]= 15750	m[2,3]= 2625	m[3,4]= 750	m[4,5]=1000	
r=3	m[1,3]= 7875	m[2,4]= 4375	m[3,5]= 2500		
r=4	m[1,4]= 9375	m[2,5]= 7125			
r=5	m[1,5]= 11875				

标记函数 $s[i, j]$

r=2	s[1,2]=1	s[2,3]=2	s[3,4]=3	s[4,5]=4
r=3	s[1,3]=1	s[2,4]=3	s[3,5]=3	
r=4	s[1,4]=3	s[2,5]=3		
r=5	s[1,5]=3			

解的追踪：s[1,5]=3➔$(A_1 A_2 A_3)(A_4 A_5)$

s[1,3]=1➔$A_1(A_2 A_3)$

输出

计算顺序为：$(A_1(A_2 A_3))(A_4 A_5)$

最少的乘法次数：m[1,5]=11875

两种实现的比较

- 递归实现：时间复杂度高，空间开销小
- 迭代实现：时间复杂度低，空间消耗多
  - 原因：递归实现子问题多次重复计算，子问题计算次数呈指数增长。迭代实现每个子问题只计算一次
- 动态规划时间复杂度：
  - 备忘录各项计算量之和+追踪解工作量
  - 追踪解的工作量通常是问题规模的多项式函数，一般不会超过计算的工作量

组合问题中的动态规划法

最长公共子序列

子序列： 若给定序列$X=\{x_1,x_2,\ldots,x_m\}$，则另一序列$Z=\{z_1,z_2,\ldots,z_k\}$，是X的子序列是指：存在一个**严格递增**下标序列$\{i_1,i_2,\ldots,i_k\}$使得对于所有$j=1,2,\ldots,k$有：$z_j=x_{i_j}$。

例： 序列$Z=\{B,C,D,B\}$是$X=\{A,B,C,B,D,A,B\}$的子序列，相应的递增下标序列为$\{2,3,5,7\}$。

给定两个序列X和Y，当另一序列Z既是X的子序列又是Y的子序列时，称Z是序列X和Y的**公共子序列**。

最长公共子序列

- **问题：** 给定两个序列$X=\{x_1,x_2,\ldots,x_m\}$和$Y=\{y_1,y_2,\ldots,y_n\}$，找出X和Y的最长公共子序列(Longest Common Subsequence, LCS)
- **实例：**

  X: A B C B D A B

  Y: B D C A B A

  最长公共子序列：B C B A，长度4

➔不是唯一的，长度相等情况下，可能有多个不同公共子序列，只需给出一个即可

穷举法

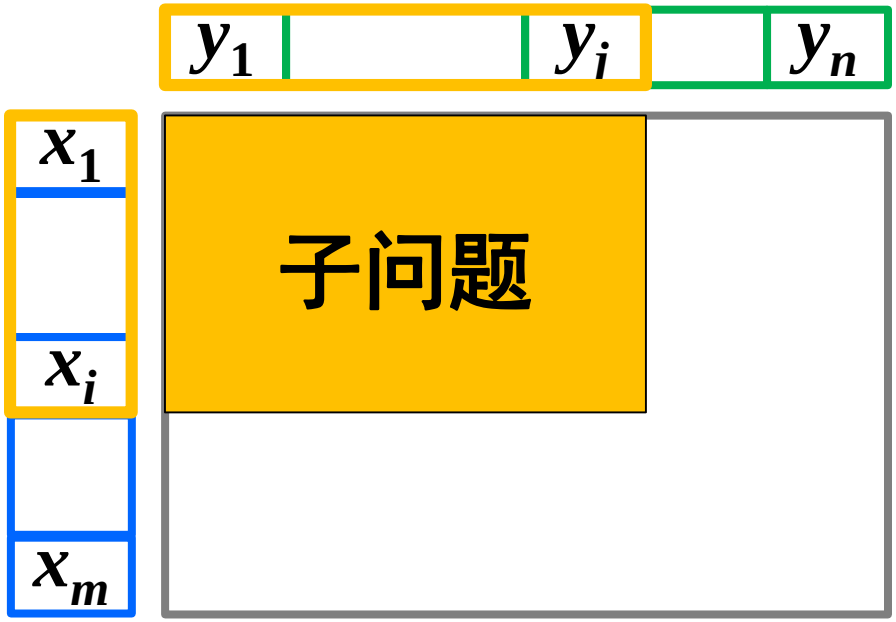
- 不妨设$m \leqslant n$,$|X|=m$, $|Y|=n$
- 算法：依次检查X的每个子序列在Y中是否出现

- **时间复杂度：**
  - X有2^m个子序列
  - 每个子序列$O(n)$时间

最坏情况下时间复杂度：$O(n2^m)$

子问题界定

- 参数 i 和 j 界定子问题
- X的终止位置是 i，Y的终止位置是 j
- $X_i=\{x_1,x_2,\ldots,x_i\}$，$Y_j=\{y_1,y_2,\ldots,y_j\}$



子问题间的依赖关系

- 假设两个序列X={$x_1,x_2,\ldots,x_m$}，Y={$y_1,y_2,\ldots,y_n$}，Z={$z_1,z_2,\ldots,z_k$}为X和Y的LCS，那么
  - 若$x_m = y_n$➔ $z_k = x_m = y_n$，且Z_{k-1}是X_{m-1}与Y_{n-1}的LCS
  - 若$x_m \neq y_n$，$z_k \neq x_m$➔ Z是X_{m-1}与Y的LCS
  - 若$x_m \neq y_n$，$z_k \neq y_n$➔ Z是X与Y_{n-1}的LCS

满足最优子结构性质和子问题重叠性

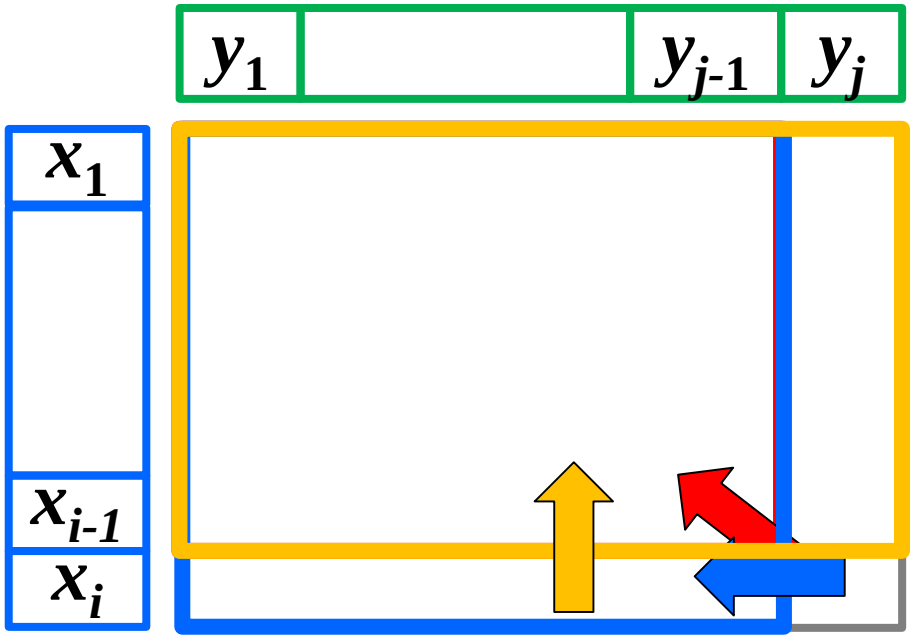
优化函数的递推方程

- 令X和Y的子序列
- $X_i=\{x_1,x_2,\cdots,x_i\}$，$Y_j=\{y_1,y_2,\cdots,y_j\}$
- $C[i,j]$：X_i与Y_j的LCS的长度

$$C[i,j]=\begin{cases}0 & \text{若}i=0\text{或}j=0\\ C[i-1,j-1]+1 & \text{若}i,j>0,\ x_i=y_j\\ \max\{C[i,j-1],C[i-1,j]\} & \text{若}i,j>0,\ x_i\neq y_j\end{cases}$$

标记函数

- 标记函数： $B[i,j]$，值为↖、←、↑
- $C[i,j]= C[i-1,j-1]+1$： ↖
- $C[i,j]= C[i,j-1]$： ←
- $C[i,j]= C[i-1,j]$： ↑



算法的伪码

- 算法 LCS(X, Y, *m*,*n*)

```
1.  for i←1 to m do  C[i,0] ← 0
2.  for i ← 1 to n do  C[0,i] ← 0
3.  for i ← 1 to m do
4.      for j ← 1 to n do
5.          if X[i]=Y[j]
6.          then  C[i,j] ← C[i-1,j-1]+1
7.                B[i,j] ←"↖"
8.          else if C[i-1,j]≥C[i,j-1]
9.               then C[i,j] ← C[i-1,j]
10.                   B[i,j] ←"↑"
11.              else C[i,j] ← C[i,j-1]
12.                   B[i,j] ←"←"
```

初值

子问题

追踪解

- 算法StructureSequence(B,*i*,*j*)
  - 输入：B[*i*,*j*]
  - 输出：X与Y的最长公共子序列

```
1.  if i=0 or j=0 then return  //序列为空
2.  if B[i,j]="↖"
3.  then  输出X[i]
4.        StructureSequence(B,i-1,j-1)
5.  else if  B[i,j]="↑"
6.        then StructureSequence(B,i-1,j)
7.        else StructureSequence(B,i,j-1)
```

标记函数的实例

- 输入：X=〈 A, B, C, B, D, A, B 〉
  Y=〈 B, D, C, A, B, A 〉

X \ Y	1	2	3	4	5	6
1	B[1,1]=↑	B[1,2]=↑	B[1,3]=↑	B[1,4]=↖	B[1,5]=←	B[1,6]=↖
2	B[2,1]=↖	B[2,2]=←	B[2,3]=←	B[2,4]=↑	B[2,5]=↖	B[2,6]=←
3	B[3,1]=↑	B[3,2]=↑	B[3,3]=↖	B[3,4]=←	B[3,5]=↑	B[3,6]=↑
4	B[4,1]=↑	B[4,2]=↑	B[4,3]=↑	B[4,4]=↑	B[4,5]=↖	B[4,6]=←
5	B[5,1]=↑	B[5,2]=↑	B[5,3]=↑	B[5,4]=↑	B[5,5]=↑	B[5,6]=↑
6	B[6,1]=↑	B[6,2]=↑	B[6,3]=↑	B[6,4]=↖	B[6,5]=↑	B[6,6]=↖
7	B[7,1]=↑	B[7,2]=↑	B[7,3]=↑	B[7,4]=↑	B[7,5]=↑	B[7,6]=↑

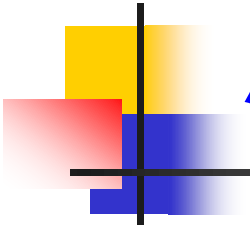
解：X[2], X[3], X[4], X[6], 即B, C, B, A

算法的时空复杂度

- 计算优化函数和标记函数
  - 赋初值，为$O(m)+O(n)$
  - 计算优化、标记函数迭代次$\Theta(mn)$
  - 循环体内常数次运算，时间为$\Theta(mn)$
- 构造解
  - 每步缩小X或Y的长度，时间$\Theta(m+n)$

算法时间复杂度：$\Theta(mn)$

空间复杂度：$\Theta(mn)$



小结

- 最长公共子序列问题的建模
- 子问题边界的界定
- 递推方程及初值，优化原则判定
- 伪码
- 标记函数与解的追踪
- 时空复杂度

背包问题(Knapsack Problem)

- 一个旅行者随身携带一个背包。可以放入背包的物品有n种，每种物品的重量和价值分别为w_i, v_i。如果背包的最大承重限制是b，每种物品可以放多个。怎么样选择放入背包的物品使得背包所装物品价值最大？
- 不妨设上述w_i, v_i, b都是正整数。

实例：$n=4$, $b=10$

$v_1=1$, $v_2=3$, $v_3=5$, $v_4=9$

$w_1=2$, $w_2=3$, $w_3=4$, $w_4=7$

建模

- 解是$\langle x_1,x_2,\ldots,x_n\rangle$，其中$x_i$是装入背包的第$i$种物品个数

目标函数 $\max \sum_{i=1}^{n} v_i x_i$

约束条件 $\sum_{i=1}^{n} w_i x_i \le b, \quad x_i \in N$

线性规划问题： 由线性条件约束的线性函数取最大或最小的问题

整数规划问题： 线性规划问题的变量x_i都是非负整数

子问题界定和计算顺序

- **子问题界定：** 由参数k和y界定
  - k：考虑物品1, 2, ..., k的选择
  - y：背包总重量不超过y
- 原始输入：$k=n, y=b$
- **子问题计算顺序：**
  - $k=1, 2, ..., n$
  - 对于给定的k，$y=1, 2, ..., b$

优化函数的递推方程

$F_k(y)$: 装前 k 种物品，总重不超过 y 背包达到的**最大价值**

$$F_k(y)=\max\{F_{k-1}(y),\ F_k(y-w_k)+v_k\}$$

$$F_0(y)=0,\ 0\leqslant y\leqslant b,\ F_k(0)=0,\ 0\leqslant k\leqslant n$$

$$F_1(y)=\left\lfloor \frac{y}{w_1} \right\rfloor v_1, \qquad F_k(y)=-\infty \quad y<0$$

标记函数

$i_k(y)$: 装前k种物品，总重不超过y，背包达到最大价值时装入物品的**最大标号**

$$i_k(y)=\begin{cases} i_{k-1}(y) & F_{k-1}(y) > F_k(y-w_k)+v_k \\ k & F_{k-1}(y) \le F_k(y-w_k)+v_k \end{cases}$$

$$i_1(y)=\begin{cases} 0 & y < w_1 \\ 1 & y \ge w_1 \end{cases}$$

动态规划求解背包问题

输入： $n=4, b=10$

$v_1=1, v_2=3, v_3=5, v_4=9$

$w_1=2, w_2=3, w_3=4, w_4=7$

$F_k(y)$的计算表如下：

k \ y	1	2	3	4	5	6	7	8	9	10
1	0	1	1	2	2	3	3	4	4	5
2	0	1	3	3	4	6	6	7	9	9
3	0	1	3	5	5	6	8	10	10	11
4	0	1	3	5	5	6	9	10	10	12

追踪解

k \ y	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	1	1	1	1	1	1
2	0	1	2	2	2	2	2	2	2	2
3	0	1	2	3	3	3	3	3	3	3
4	0	1	2	3	3	3	4	3	4	4

$i_4(10)=4$ ➔ $x_4 \geqslant 1$

$i_4(10- w_4) = i_4(3) = 2$ ➔ $x_2 \geqslant 1$, $x_4=1$, $x_3=0$

$i_2(3- w_2) = i_2(0) = 0$ ➔ $x_2=1$, $x_1=0$

解： $x_1=0$, $x_2=1$, $x_3=0$, $x_4=1$，价值12

追踪算法

- **算法 TrackSolution**
- **输入：$i_k(y)$表，k=1, 2, ..., n，y=1, 2, ..., b**
- **输出：$x_1,x_2,...,x_n$，n种物品的装入量**

```
1.  for j ← 1 to n do x_j ← 0
2.  y ← b, j ← n
3.  j ← i_j(y)
4.  x_j ← 1
5.  y ← y - w_j
6.  while i_j(y)=j do
7.      y ← y - w_j
8.      x_j ← x_j +1
9.  if i_j(y)≠ 0 then goto 3.
```

初始追踪位置

继续放j种物品

继续追踪下一种

时间复杂度

根据公式

$$F_k(y)=\max\{F_{k-1}(y), F_k(y-w_k)+v_k\}$$

备忘录需计算nb项，每项常数时间，计算时间为$O(nb)$

伪多项式时间算法：时间为参数b和n的多项式，不是输入规模的多项式。参数b是整数，表达b需要$\log b$位，输入规模以n和$\log b$为参数。

背包问题的推广

- **物品数受限**背包：第i种物品最多用n_i个
- **0-1背包**问题：x_i =0,1，i=1, 2, ..., n
- **多背包**问题：m个背包，背包j装入最大重量B_j, j=1, 2, ..., m。在满足所有背包重量约束条件下使装入物品价值最大。
- **二维背包**问题：每件物品有重量w_i和体积t_i，i=1, 2, ..., n，背包总重不超过b，体积不超过V，如何选择物品以得到最大价值。

第三章小结

- 理解动态规划算法的概念
- 掌握动态规划算法的基本要素
  - 最优子结构性质
  - 重叠子问题性质
- 掌握设计动态规划算法的步骤
  - 建模→分段→分析→判断→求解
- 动态规划法应用实例
  - 最短路径、矩阵连乘（递归、迭代）
  - 最长公共子序列、背包问题