算法设计与分析

第3章 动态规划
(1)

引子 - 多数问题

- 问题描述
  - 给定一个数组A，包含n个元素,只用“=”测试找出大数元素（majority element ）(其在A中出现的次数多于 n/2次)
- 例如, 给定 (2, 3, 2, 1, 3, 2, 2), 则2 是大数元素，因为4>7/2.
- Trivial solution:
  - counting (计数) is O(n^2).

```
Majority(A[1, n])
    for(i = 1 to n)
        M = 1
        for(j = 1 to n)
            if (i != j and A[i]==A[j])    M++
        end
        if (M>n/2) return "A[i] is the majortiy"
    end
    return "No majortity"
```

引子 - 多数问题

- 分治法

```
Majority(A[1, n]){
   if n=1 then
      return A[1]
   else{
      m1=Majority(A[1, n/2])
      m2=Majority(A[n/2+1, n])}
    test if m1 or m2 is the majority for A[1, n]
    return majority or no majority.
}
```

复杂度分析 (Counting):

$T(n) = 2T(n/2)+O(n) = O(n\log n)$

但是，该问题有线性时间算法.

A=(2, 1, 3, 2, 1, 5, 4, 2, 5, 2)

frequency[1] = 2

frequency[2] = 4

frequency[3] = 1

frequency[4] = 1

frequency[5] = 2

```
for(i=1 to n)   ++frequency[ A[i] ]
M = Max(frequency[ A[i] ] )
if (M > n/2)
     check( M = = frequency[ A[j] ] )
     return "A[j] is the majority"
```

- **故事的Moral (寓意)?**

引子 - 多数问题

- 分治法

```
Majority(A[1, n]){
    if n=1, then
        return A[1]
    else{
        m1=Majority(A[1, n/2])
        m2=Majority(A[n/2+1, n])}
     test if m1 or m2 is the majority for A[1, n]
     return majority or no majority. }
```

复杂度分析 (Counting):

$$T(n) = 2T(n/2)+O(n)$$
$$= O(n\log n)$$

- 故事的寓意: 分治法并不总是能给出最好的解决方案!

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

  （1）最优子结构性质

  （2）重叠子问题性质
- 掌握设计动态规划算法的步骤。

  （1）找出最优解的性质，并刻划其结构特征

  （2）递归地定义最优值

  （3）以自底向上的方式计算出最优值

  （4）根据计算最优值时得到的信息，构造最优解

学习要点

- 通过应用范例学习动态规划算法设计策略

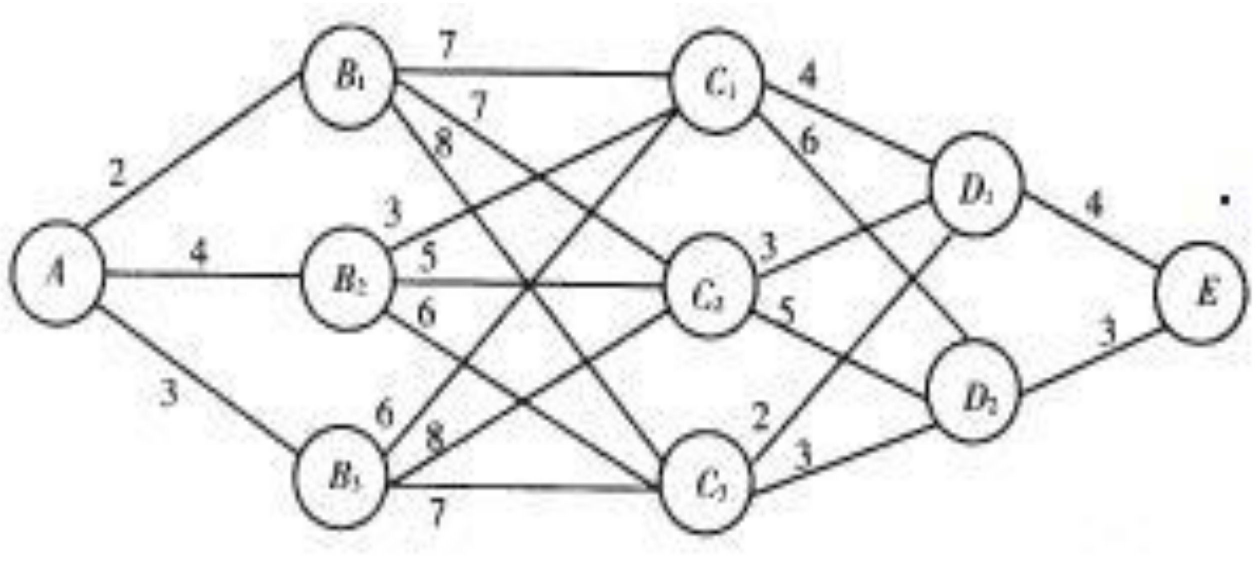
（1）矩阵连乘问题
（2）最长公共子序列
（3）凸多边形最优三角剖分
（4）0/1背包问题
（5）最优二叉搜索树

引言

- 物流供应链中的供应商/快递公司选择

- 最短路径问题

哈尔滨



广州

图 8—1 多阶段决策过程图

解(solution)：怎样走？

值value：路径长度最短？

引言

- 一个英国小伙子计划周末从谢菲尔德回到自己在伦敦附近的家里
  - 两地相距大约300公里
  - 坐火车需要三小时
  - 可最便宜的车票也要47英镑，约合人民币440元。

路程最短

- 另辟蹊径
  - 先坐廉价航空到柏林再转机到伦敦
  - 来回2000公里
  - 机票加起来才21英镑，约合人民币两百多。

价格最低

引言

- 在50年代,Richard Bellman等人提出了解决多阶段决策问题的“最优性原理”，并创建了最优化问题的一种新的求解方法--动态规划（Dynamic programming）。
  - 1957年，出版“动态规划”
- 优点：
  - 对许多问题，比线性规划或非线性规划更有效。
- 弱点：
  - （1）得出函数方程后，尚无统一处理方法；
  - （2）维数屏蔽:变量个数（维数）太大时无法解决。

线性规划是研究多变量函数在变量具有约束条件下的最优化问题，

动态规划没有一个标准表达式，而是不同的具体问题就有不同的、具体的数学表达式；
动态规划没有统一处理格式，必须依据问题本身特性，利用灵活的数学技巧来解决，属于灵活动态的数学方法。

引言

- 线性规划例子

$$\max z = x_1 + x_2 + 3x_3 - x_4$$

$$\text{s.t.} \quad \begin{aligned} & x_1 + 2x_3 <= 18 \\ & 2x_2 - 7x_4 <= 0 \\ & x_1 + x_2 + x_3 + x_4 = 9 \\ & x_2 - x_3 + 2x_4 >= 1 \\ & x_i >= 0 (i=1,2,3,4) \end{aligned}$$

解为$(x_1,x_2,x_3,x_4)=(0,3.5,4.5,1)$, 最优值=16

引言

Why?

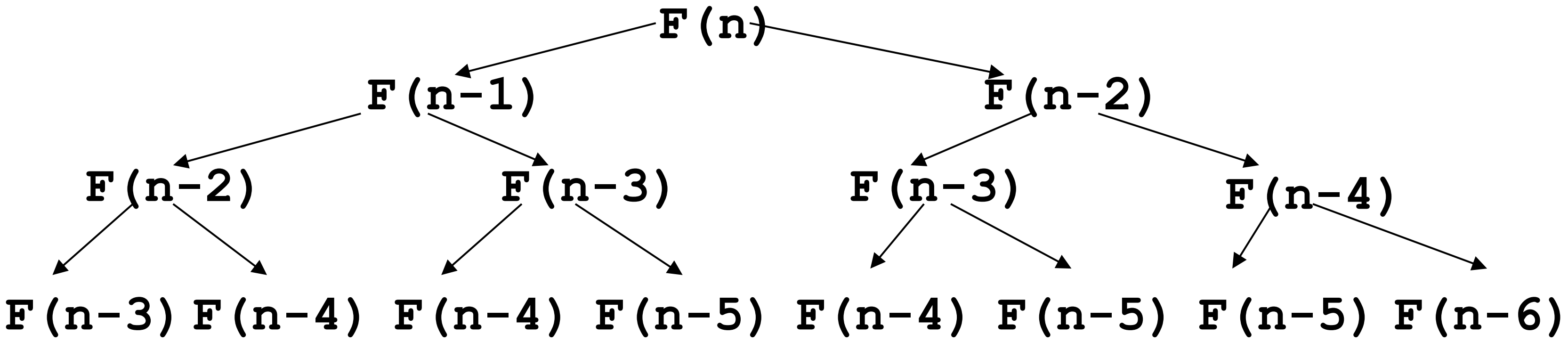
What?

How?

引言

why?

- 分治技术的问题
  - 子问题是相互独立的
  - 如果子问题不是相互独立的，分治方法将重复计算公共子问题，效率很低，甚至在多项式量级的子问题数目时也可能耗费指数时间



- 解决方案：动态规划
  - 用表来保存所有已解决子问题的答案
  - 不同算法的填表格式是相同的

引言

- Fibnacci算法的改进
  - 优化这些重复的调用：
    - 用**表的形式**将计算出的fibnacci数fib(n)存储起来
    - 在函数递归调用前，先在表中查找是否存在，如果没有就调用递归函数；否则不用计算，直接将表中存储的fibnacci数返回即可

```
memo={}  #字典数据结构
def fib2(n):
    if n in memo:
        return memo[n]
    else:
        if n<=2:
            f=1
        else:
            f=fib2(n-1)+fib2(n-2)
    memo[n]=f
    return f
```

查表　O(1)

自顶向下求解

分析：
求解fib(n)时，每一个fibnacci数只存在一次递归调用，共有n次调用。
查表时间为O(1).
故时间复杂度时O(n)

引言

- 进一步改进
  - 用自底向上的方法来实现递归， 就是动态规划算法

```
def fib_bottom_up(n):
    fib={}  # 存储结果的字典
    for k in range(n+1):
        if k<=2:
            f=1
        else:
            f=fib[k-1]+fib[k-2]
        fib[k]=f
    return fib[n]
```

自底向上求解
复杂度： O(n)

引言

why?

- 最优化问题
  - 可能有多个可行解，每个解对应一个值，找出最优值的解
  - 可以分为多个子问题，子问题解被重复使用
- 描述
  - 给定一组约束条件和一个代价函数，在解空间中搜索具有最小或最大代价的最优解
  - 例如： 坐飞机从哈尔滨到广州，请找出花费最少的旅行路线？

引言

what?

- 动态规划
  - 不是专用算法(algorithm)，而是类似分治法的一种技术(technique)
  - programming表示列表方法（tabular method），而不是指编程（not computer programming.）
  - 用于优化问题
    - 不仅要解决问题，而且还要以最优的方式解决问题

引言

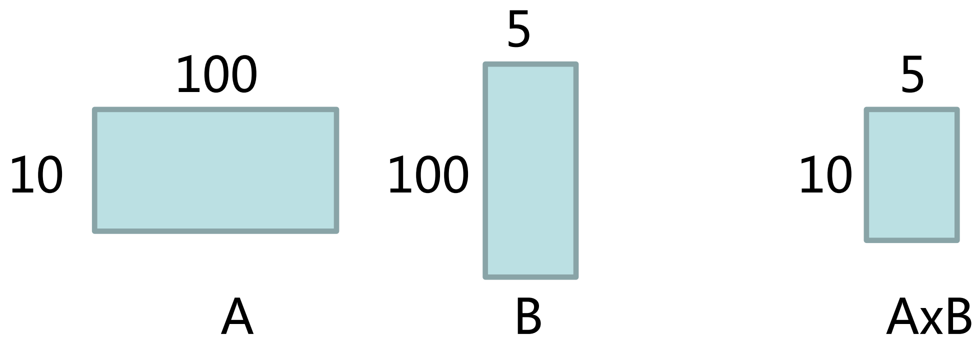
What?

和分治法有
什么不同?

- 动态规划特点
  - 把原始问题划分成一系列子问题
  - 求解每个子问题仅一次，并将其结果保存在一个表中，以后用到时直接存取。不重复计算，节省计算时间
  - 自底向上地计算最优值
- 适用范围
  - 一类优化问题：可分为**多个**相关子问题，子问题的解被重复使用

矩阵连乘问题

- 问题：n 个矩阵$<A_1, A_2, ..., A_n>$相乘，称为‘矩阵连乘’，如何求积？
  - 输入： $<A_1,A_2,...,A_n>$，A_i 是矩阵(i=1,...,n)
  - 输出：计算$A_1A_2...A_n$的**一种**最小代价方法
- 分析：
  - 矩阵乘法的代价/复杂性： 乘法的次数
  - 若A 是p ×q 矩阵，B 是q ×r 矩阵，则A ×B 的代价是O(pqr)
  - 例如
    - A是10 ×100， B是100 ×5，C是5 ×50
    - 则A ×B 的代价是： 10*100*5



矩阵连乘问题

- 分析1：代价取决于乘法次数
- 分析2: 两个矩阵(维度分别为p×q, q×r)相乘的乘法次数= pqr
  - 首先考虑两个矩阵的乘法（仅当矩阵A和B相容时，A和B能相乘）
  - If A is p×q, B is q×r, then C is p×r.
  - 求得矩阵 C 的计算时间主要由 line7 的标量相乘所决定，相乘次数为pqr

$$
\begin{pmatrix} \cdots\cdots\ \cdot\ \cdot\cdot \\ \cdots\cdots c_{ij}\cdot\cdot \\ \cdots\cdots\ \cdot\ \cdot\cdot \\ \cdots\cdots\ \cdot\ \cdot\cdot \end{pmatrix} = \begin{pmatrix} \cdots\cdots\cdots \\ **** \\ \cdots\cdots\cdots \\ \cdots\cdots\cdots \end{pmatrix} \begin{pmatrix} \cdots\cdots * \cdot\cdot \\ \cdots\cdots * \cdot\cdot \\ \cdots\cdots * \cdot\cdot \\ \cdots\cdots * \cdot\cdot \end{pmatrix}
$$

MATRIX-MULTIPLY(*A*, *B*)

1 if *columns*[*A*] ≠ *rows*[*B*]
2 then return “error: incompatible dimensions”
3 else for *i* ← 1 to *rows*[*A*]
4 for *j* ← 1 to *columns*[*B*]
5 *C*[*i*, *j*] ← 0
6 for *k* ← 1 to *columns*[*A*]
7 *C*[*i*, *j*]←*C*[*i*, *j*]+*A*[*i*, *k*]·*B*[*k*, *j*]
8 return *C*

矩阵连乘问题

- For $A_{p\times q}$, $B_{q\times r}$, C=AB is p×r. 乘法次数是 pqr
- 考虑三个矩阵连乘 $\langle A_1, A_2, A_3 \rangle$
- 假设 A_1: 10×100; A_2: 100×5; A_3: 5×50
  - If $A = ((A_1A_2)A_3)$
    a) $C = A_1A_2$, 乘法次数 $10 \cdot 100 \cdot 5 = 5000$ $C_{10\times 5}$
    b) $A = CA_3$,乘法次数 $10 \cdot 5 \cdot 50 = 2500$ $A_{10\times 50}$
    then, 总乘法次数= 7500
  - If $A = (A_1(A_2A_3))$
    a) $C_{100\times 50} = A_2A_3$,乘法次数 $100 \cdot 5 \cdot 50 = 25{,}000$
    b) $A_{10\times 100} = A_1C$,乘法次数 $10 \cdot 100 \cdot 50 = 50{,}000$
    then,总乘法次数75,000 .
  - 第1种情况快第2种情况 10倍.

矩阵连乘问题

- 分析3：n个矩阵连乘，乘法次数由不同的加括号方式决定
  - 假设：给定n个矩阵{A_1,A_2,… A_n}，其中A_i与A_{i+1}是可乘的，i=1,2,…,n-1。考察这n个矩阵的连乘积： $A_1A_2 \ldots A_n$
  - 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　完全加括号形式
  - 对所有加括号的方式，矩阵连乘的积相同

(A1 (A2 (A3 A4)))，(A1 ((A2 A3) A4))，
((A1 A2) (A3 A4))，((A1 (A2 A3)) A4)，
(((A1 A2) A3) A4)

矩阵连乘问题

- 分析3：**不同的加括号方式**决定乘法次数，可导致不同的、甚至极其富有戏剧性差别的乘法开销
  - 设有四个矩阵A,B,C,D，维数分别是：
    $A=50\times10$ $B=10\times40$ $C=40\times30$ $D=30\times5$

    有五种完全加括号的方式:

结合规则	代价
(A((BC)D))	16000
(A(B(CD)))	10500
((AB)(CD))	36000
(((AB)C)D)	87500
((A(BC))D)	34500

矩阵连乘问题

- 分析4: 问题转化为求n个矩阵的**完全加括号形式**

是顺序，不是值

- 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

最小计算单元?

- 完全加括号的矩阵连乘积可递归地定义为：
  - （1）单个矩阵是完全加括号的；
  - （2）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)

矩阵连乘问题

- 矩阵连乘问题：
  - 给定一个矩阵链<$A_1, A_2, \ldots, A_n$>，其中矩阵 A_i 的维数为 $p_{i-1} \times p_i$ ，寻找一种矩阵连乘完全加括号方式，使得矩阵连乘积的乘法次数最少

解

值

- 该问题中，我们的目的是仅仅寻求有最少乘法个数的矩阵相乘顺序，而不是实际进行矩阵相乘

$A_1 A_2 A_3 A_4 A_5$:　$(A_1 A_2 A_3)(A_4 A_5)$?　$(A_1 A_2)(A_3 A_4 A_5)$?　……

$A_1 (A_2 A_3)$?　$(A_1 A_2) A_3$?

矩阵连乘问题

- 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乘法次数，从中找出**最少**的一种
- 算法复杂度分析：
  - 对于n个矩阵的连乘积，设其不同的所有可能完全加括号方式的个数为P(n)
  - 由于每种加括号方式都可以分解为两个矩阵子序列的加括号问题：
  - P(n)的递推式如下：

$$\underbrace{(A_1 \dots A_k)}_{P(k)}\underbrace{(A_{k+1} \dots A_n)}_{P(n-k)}$$

$$\boxed{P(n)=\begin{cases} 1 & n=1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n>1 \end{cases} \Rightarrow P(n)=\Omega(4^n/n^{3/2})}$$

矩阵连乘问题

有没有更好的方法?

矩阵连乘问题

- 动态规划法基本步骤
  - 1 找出最优解的性质，并刻划其**结构特征**
  - 2 **递归**地定义最优值
  - 3 以**自底向上**的方式计算出最优值
  - 4 根据计算最优值时得到的信息，**构造最优解**

矩阵连乘问题

- 改进策略
  - 考虑矩阵连乘积的中缀 $A_i\ A_{i+1}\ \ldots\ A_j$， A_i: p_{i-1}*p_i，**先求最少乘法次数再找加括号形式**
  - 设最优计算次序在A_k和A_{k+1}之间断开：

$$(A_i\ A_{i+1}\ \ldots A_k\ )(A_{k+1}\ldots\ A_j)$$

  - 定义： **m[i,j]**表示$A_i\ldots A_j$的最少乘法次数

则有：$m[i,j]=m[i,k]+m[k+1,\ j]\ +p_{i-1}p_kp_j$

$$(A_i\quad A_{i+1}\quad \ldots\quad A_k\quad)\ (\ A_{k+1}\quad \ldots\quad A_j\ )$$

$$p_{i-1}p_i\qquad \ldots\ p_{k-1}p_k\qquad p_kp_{k+1}\qquad \ldots\qquad p_{j-1}p_j$$

矩阵维度： $p_{i-1}p_k$ p_kp_j 乘法次数$p_{i-1}p_kp_j$

$$p_{i-1}p_j$$

矩阵连乘问题-改进策略

- $m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$
- 问题：
  - (1) k怎么找？ 找一个最优的k，使得m[i,j]最小
  - (2) 找到了k，A[i,k]和A[k+1,j]次序是不是最优的呢？

$$(A_i \quad A_{i+1} \quad \dots \quad A_k)\ (A_{k+1} \quad \dots \quad A_j)$$

矩阵连乘问题-改进策略

动态规划基本要素1：
最优子结构：问题的最优解包含着其子问题的最优解。

问题：$\mathbf{A_1 \ldots A_n}$

		最优解
原问题	$A_iA_{i+1} \ldots A_j$	$(A_i \ldots A_k)(A_{k+1} \ldots A_j)$
子问题	$\mathbf{A_i \ldots A_k}$	$(A_i \ldots A_k)$
	$\mathbf{A_{k+1} \ldots A_j}$	$(A_{k+1} \ldots A_j)$

要证明：$(A_i \ldots A_k)$是子问题$\mathbf{A_i \ldots A_k}$ **的最优完全加括号形式**

矩阵连乘问题

- 动态规划法基本步骤
  - 1 找出最优解的性质，并刻划其结构特征
  - 2 递归地定义最优值
  - 3 以自底向上的方式计算出最优值
  - 4 根据计算最优值时得到的信息，构造最优解

步骤1：分析最优解的结构

- 将矩阵连乘积$A_iA_{i+1}\ldots A_j$，简记为A[i:j] ，这里$i\leq j$, A_i是$p_{i-1}\times p_i$的矩阵
- 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，$i\leq k<j$，则其相应完全加括号方式为$(A_iA_{i+1}\ldots A_k)(A_{k+1}A_{k+2}\ldots A_j)$
- 计算量：
  - A[i:k]的计算量加上A[k+1:j]的计算量，再加上A[i:k] 和A[k+1:j]相乘的计算量

$$m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$$

步骤1：分析最优解的结构

- 最优子结构 ✦
  - 设矩阵连乘的最优完全加括号将矩阵连乘分成 $A_{i..k}$ 和 $A_{k+1..j}$ 两部分之积

    $(A_i A_{i+1} \dots A_k)(A_{k+1} \dots A_{j-1} A_j)$
  - $A_{i..j}$ 的最优全括号中的 $A_{i..k}$ 的全括号必定是 $\mathbf{A_{i..k}}$ 的最优全括号

    $(A_i (A_{i+1} \dots) \dots A_k)(A_{k+1} \dots A_{j-1} A_j)$

证明方法：反证+构造

- 证明
  - 最优全括号 $A_{i..j} = (A_i \dots A_k)(A_{k+1} \dots A_j) = M \cdot N$，$M = (A_i A_{i+1} \dots A_k)$. 若有一个**更优**的完全加括号形式 $A_i \dots A_k = P$, 用 P 代替M, 则 P·N 将产生一个新的$A_{i..j}$加括号形式，其计算量比M·N的计算量还小.产生矛盾.
  - 同理，上述结论对$A_{k+1}A_{k+2} \dots A_j$也成立

> 动态规划基本要素1：
> 最优子结构：问题的最优解包含着其子问题的最优解。

步骤1：分析最优解的结构

- 矩阵连乘积计算次序问题的最优解包含着其子问题的最优解。这种性质称为最优子结构性质
  - 基于最优子结构，可以从子问题的最优解构造原问题的最优解
- 矩阵连乘问题的任何最优解必包含其子问题的最优解
  - 将问题分为两个子问题 $(A_iA_{i+1} \ldots A_k)$and $(A_{k+1}\ A_{k+1} \ldots\ A_j)$
  - 求子问题的最优解
  - 合并子问题的最优解
- 问题的最优子结构性质是该问题可用动态规划算法求解的显著特征

步骤1：分析最优解的结构

- 问题：找到一个k，使得m[i,j]最优

$m[i,j]=m[i,k]+m[k+1, j] +p_{i-1}p_kp_j$

k在哪里？？

如何找？？

- 当寻找矩阵连乘分割点时，需要考虑所有分割位置以确保最优解是其中之一

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

- 特征
  - 计算A[i:j]的最优次序所包含的计算矩阵子链 A[i:k]和A[k+1:j]的次序也是最优的
  - 关键过程:选择括号分界线

矩阵连乘问题

- 动态规划法基本步骤
  - 1 找出最优解的性质，并刻划其结构特征。
  - 2 递归地定义最优值,建立递归关系。
  - 3 以自底向上的方式计算出最优值。
  - 4 根据计算最优值时得到的信息，构造最优解。

步骤2：建立递归关系

- 找k的策略
  - 矩阵连乘**子问题的的形式**为：

    $$A_{i}A_{i+1}\dots A_{j}\text{，}\quad \text{for } 1 \le i \le j \le n.$$

    - Not $A_1A_2\dots A_j$, Why?

  - 设计算A[i:j]，1≤i≤j≤n，所需要的最少数乘次数**m[i,j]**，则**原问题**（计算 $A_{1..n}$ ）的最优值为m[1,n]

  - 当**i=j**时，A[i:j]=A_i，只有一个矩阵A_i ，没有矩阵元素相乘，因此，

    m[i,i]=0，i=1,2,…,n

步骤2：建立递归关系

- 当 **i<j**，设矩阵连乘 $A_{i..j}$ 最优全括号的分割点位于A_k 与A_{k+1} 之间，即k为最佳断开位置：

$$(A_i\ A_{i+1}\ \dots A_k\ )(A_{k+1}\ \dots A_j\ )$$

  - m[i, j] = 计算$A_{i..k}$的最小代价
    +计算$A_{k+1..j}$的最小代价
    + $A_{i..k}$ and $A_{k+1..j}$相乘的代价
  - A_i 维数 $p_{i-1} \times p_i$，则计算 $A_{i...k}\ A_{k+1..j}$ 需要数乘次数为：$p_{i-1}p_kp_j$ ，有：
    $m[i, j]=m[i, k]+m[k+1, j]+p_{i-1}p_kp_j$

k: 最优断开位置

  - k的位置可以为 {i, i+1, ..., j-1}，共j-i种可能，**遍历所有**可以找到最小

步骤2：建立递归关系

- 递归地定义m[i,j]为：

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

How?

	原问题	子问题
	A[1,n]	A[i,j]
最少乘法次数	m[1,n]	m[i,j]

矩阵连乘问题

- 动态规划法基本步骤
  - 1 找出最优解的性质，并刻划其结构特征。
  - 2 递归地定义最优值。
  - 3 以自底向上的方式计算出最优值。
  - 4 根据计算最优值时得到的信息，构造最优解。

步骤3：计算最优值

- 矩阵连乘的递归算法

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

```
   RecursiveMatrixChain(i,j){
1.      if(i=j) return 0;
2.      m[i ,j] ←∞;
3.      for(int k=i; k<j; k++){
4.          t←RecursiveMatrixChain(i,k)
                 +RecursiveMatrixChain(k+1,j)
                  +p[i-1]*p[k]*p[j];
5.          if(t<m[i ,j]) then m[i ,j] ←t;
        }
6.      return m[i ,j];
   }
```

步骤3：计算最优值

- 矩阵连乘的递归算法

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

```
RecursiveMatrixChain(i,j){                                  T(n)
      if(i=j) return 0;
      m[i ,j] ←∞;                                           O(1)
      for(int k=i; k<j; k++){
          t←RecursiveMatrixChain(i,k)                       T(k)
                 +RecursiveMatrixChain(k+1,j)               T(n-k)
                  +p[i-1]*p[k]*p[j];
          if(t<m[i ,j]) then m[i ,j] ←t;                    O(1)
      }
      return m[i ,j];
}
```

步骤3：计算最优值

- 分析
  - 采用递归算法需要指数运算时间，与 brute-force 方法一样，递归算法并不是好的方法

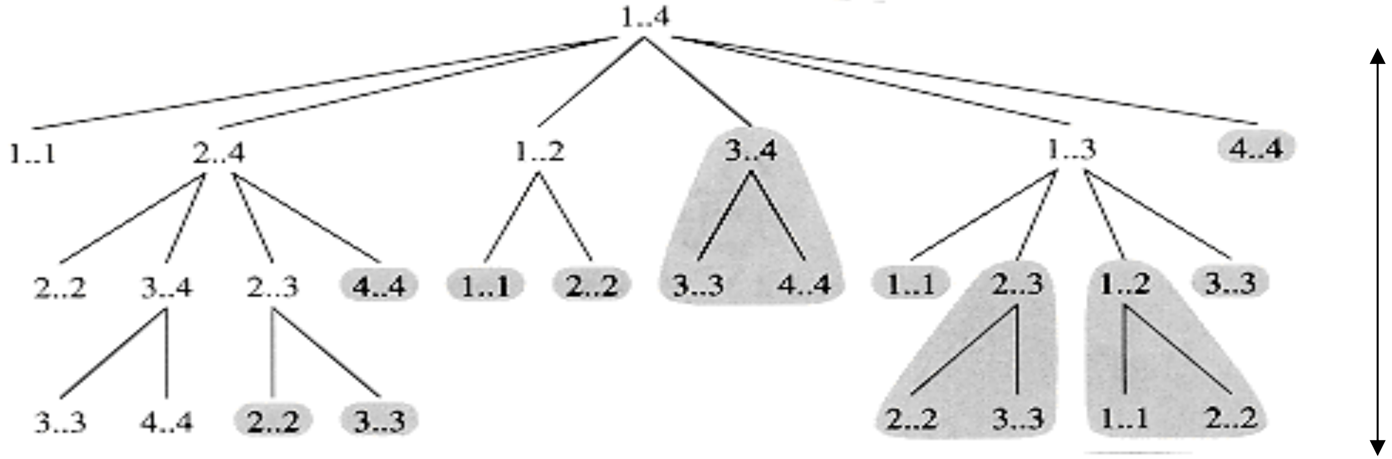
$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \geq 2 \sum_{i=1}^{n-1} T(i) + n$$

可证得：　$T(n) = \Omega(2^n)$

  - 根据方程递归求解 m[1, n]
    - 高度=j-i+1 ⇒ 指数时间复杂度
    - 原因：重复计算

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

递归树：i=1，j=4



步骤3：计算最优值

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

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

多项式级的子问题个数

$$1 \le i < j \le n,\ C_n^2;\ 1 \le i = j \le n,\ C_n^1 = n$$

- 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。

动态规划基本要素2：

重叠子问题：递归算法自顶向下解问题时，有些子问题被反复计算多次。

步骤3：计算最优值

- 矩阵连乘的备忘录算法
  - 在求解完一个子问题后，把答案存在表里，在下次需要解此子问题时，只需要到表中查看，而无需重新计算

```
MemoizedMatrixChain(a){
    for i←1 to n
        for j ←1 to n
            do m[i ,j] ←∞;
    return lookupChain(a,1,n);
}
```

时间复杂度：$O(n^3)$

（ n^2 个子问题，每个问题填入时间$O(n)$ ）

空间复杂度：$O(n^2)$

```
lookupChain(a,i,j){
    if (m[i ,j]<∞) then
        return m[i ,j];
    if(i=j) return 0;
    for(int k=i;k<j;k++){
        t←lookupChain(a,i,k)
            +lookupChain(a,k+1,j)
            +p[i-1]*p[k]*p[j];
        if(t<m[i ,j]) then m[i ,j] ←t;
    }
    return m[i ,j];
}
```

步骤3：计算最优值

- 矩阵连乘的**动态规划算法**
  - 基本思想：
    - 不用递归方法，而采用列表方式、自底向上的方法计算最优解
  - 可依据其**递归式**以自底向上的方式进行计算。
    - 在计算过程中，保存已解决的子问题答案。
    - 每个子问题只计算一次，而在后面需要时只要简单查一下，从而避免大量的重复计算，最终得到多项式时间的算法

步骤3：计算最优值

- 当i=j时, $A[i:j]=A_i$，有$m[i,i]=0$，$i=1,2,\ldots,n$
- 当i<j时，$m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$
- 例如： A1*A2*A3*A4*A5

m[1,1] **m[1,2]** **m[1,3]** **m[1,4]** **m[1,5]**

m[2,5]

m[3,5]

m[4,5]

m[5,5]

步骤3：计算最优值

- 当i=j时, $A[i:j]=A_i$，有$m[i,i]=0$，$i=1,2,\ldots,n$
- 当i<j时，$m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$
- 例如： A1*A2*A3*A4*A5

m[1,1]	m[1,2]	m[1,3]	m[1,4]	m[1,5]
	m[2,2]	m[2,3]	m[2,4]	m[2,5]
		m[3,3]	m[3,4]	m[3,5]
			m[4,4]	m[4,5]
				m[5,5]

步骤3：计算最优值

j

1	2	3	4	5	i
m[1,1]	m[1,2]	m[1,3]	m[1,4]	m[1,5]	1
	m[2,2]	m[2,3]	m[2,4]	m[2,5]	2
		m[3,3]	m[3,4]	m[3,5]	3
			m[4,4]	m[4,5]	4
				m[5,5]	5

步骤3：计算最优值

A1	A2	A3	A4	A5
30×35	35×15	15×5	5×10	10×20
$p_0 \times p_1$	$p_1 \times p_2$	$p_2 \times p_3$	$p_3 \times p_4$	$p_4 \times p_5$

m[1,1]	m[1,2]	m[1,3]	m[1,4]	m[1,5]
	m[2,2]	m[2,3]	m[2,4]	m[2,5]
		m[3,3]	m[3,4]	m[3,5]
			m[4,4]	m[4,5]
				m[5,5]

维度　个数　值表　解表

```
MatrixChain(int *p， int n， int **m， int **s)
{
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int r = 2; r <= n; r++)
      for (int i = 1; i <= n - r+1; i++) {
        int j=i+r-1;
        m[i][j] = m[i][i]+m[i+1][j]+ p[i-1]*p[i]*p[j];
        s[i][j] = i;
        for (int k = i+1; k < j; k++) {
          int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
          if (t < m[i][j]) { m[i][j] = t; s[i][j] = k;}
        }
      }
}
```

r:矩阵链长度

i:矩阵链起始位置

j:矩阵链终止位置

k:矩阵链断开位置

s:记录最优断开位置

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

步骤3：计算最优值

```
MatrixChain(p, n, m, s)
  n ← length[p] – 1
  for i ← 1 to n
      do m[i, i] ← 0
  for l←2 to n                    // l is the chain length.
      do for i ← 1 to n - l + 1
             do j ← i + l – 1
                m[i, j] ← ∞
                for k ← i to j - 1
                    do q←m[i, k]+m[k+1, j]+p_{i-1}p_k p_j
                       if q < m[i, j]
                          then m[i, j] ← q
                               s[i, j] ← k
  return m and s
```

A_i : 维数 $p_{i\text{-}1} \times p_i$

输入: $p = <p_0, p_1, ..., p_n>$, n, m, s

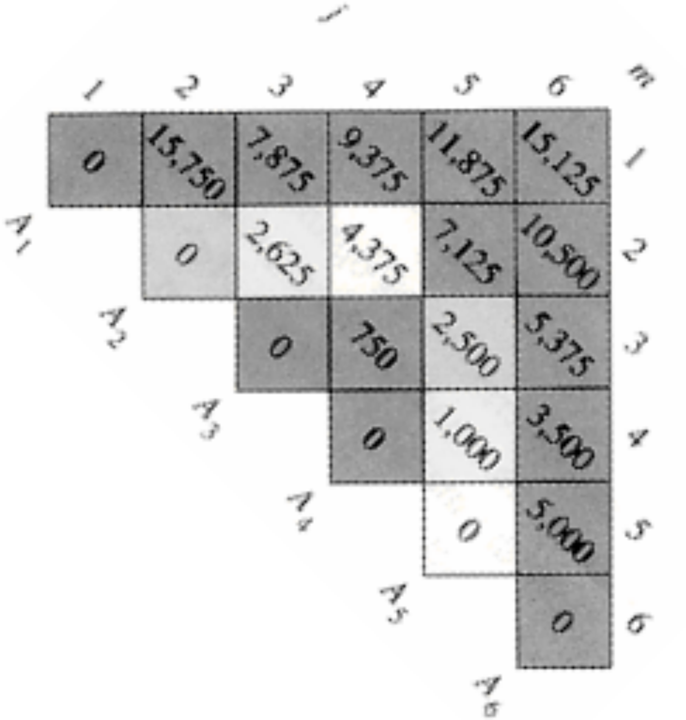
表格 $m[1..n, 1..n]$ storing the $m[i, j]$ costs;

计算 $m[i, j]$ 时，用辅助表项 $s[i, j]$ 来记录最佳位置 k 的值

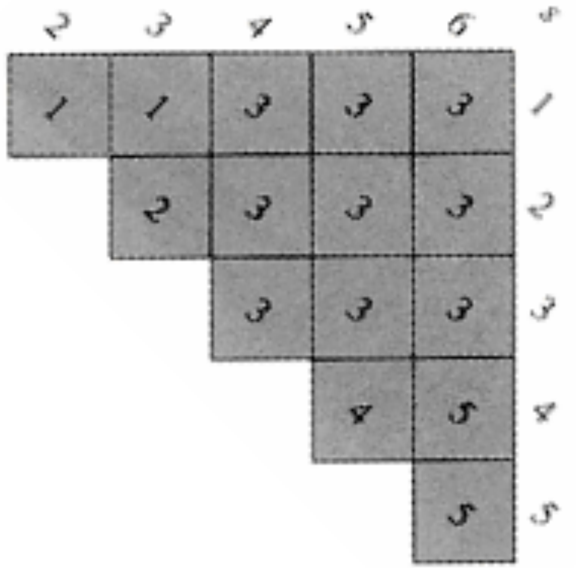
步骤3：计算最优值

```
MatrixChain(p, n, m, s))
n ← length[p] – 1
for i ← 1 to n
    do m[i, i] ← 0
for l←2 to n   // l is the chain length.
    do for i ← 1 to n - l + 1
            do j ← i + l - 1
                m[i, j] ← ∞
                for k ← i to j - 1
                    do q←m[i, k]+m[k+1, j]+p_{i-1}p_k p_j
                        if q < m[i, j]
                            then m[i, j] ← q
                                s[i, j] ← k
return m and s
```

A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

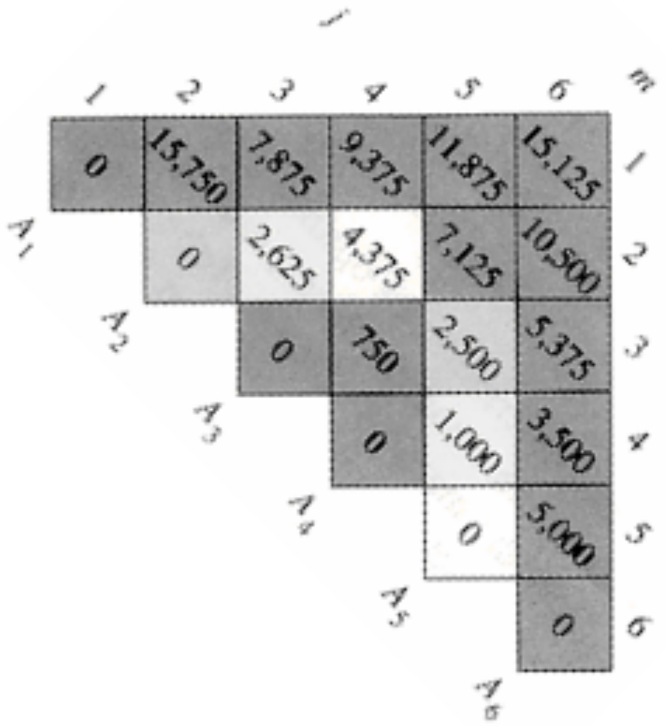


m

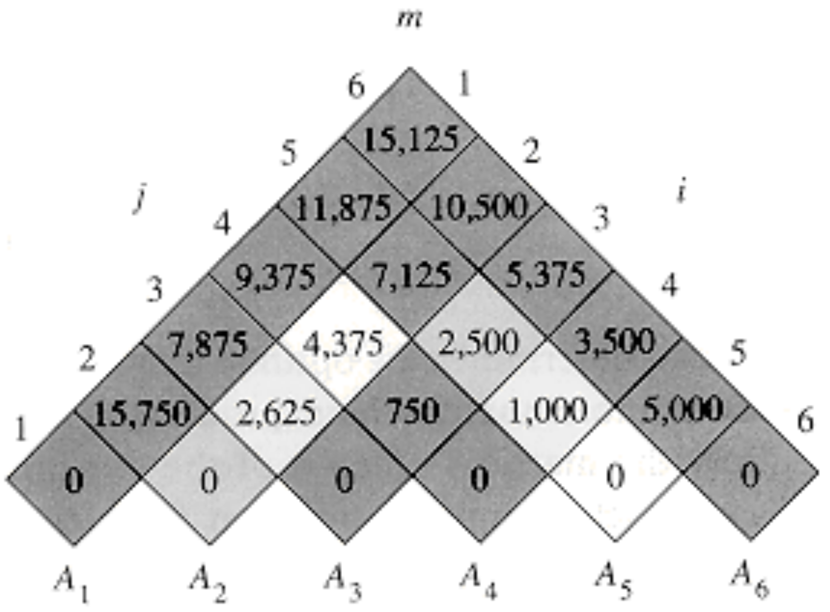


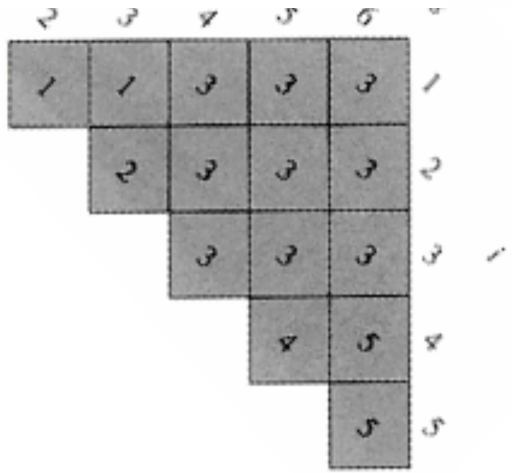
s

步骤3：计算最优值

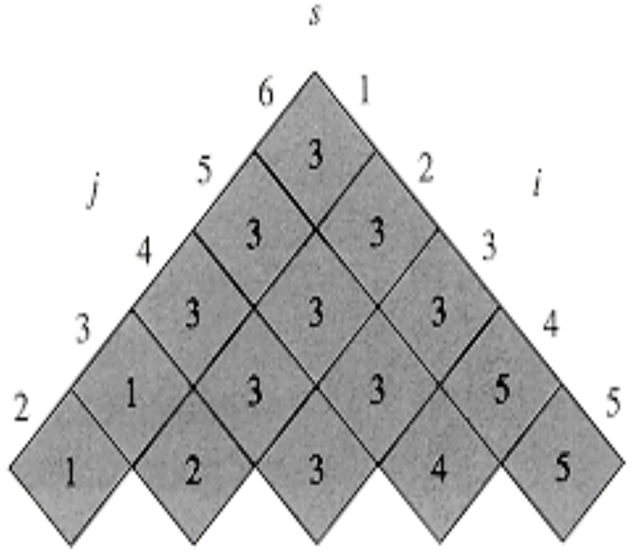


rotate 45^0





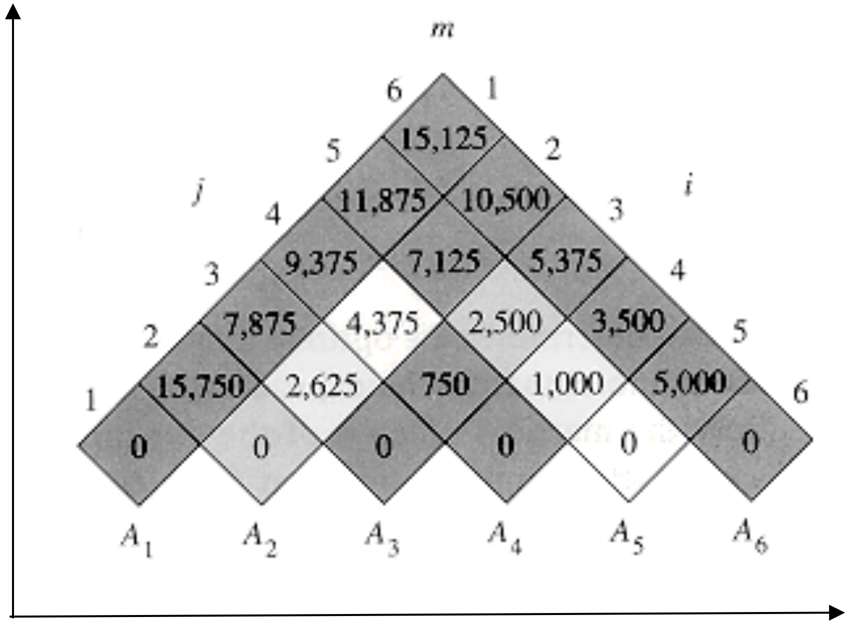
rotate 45^0

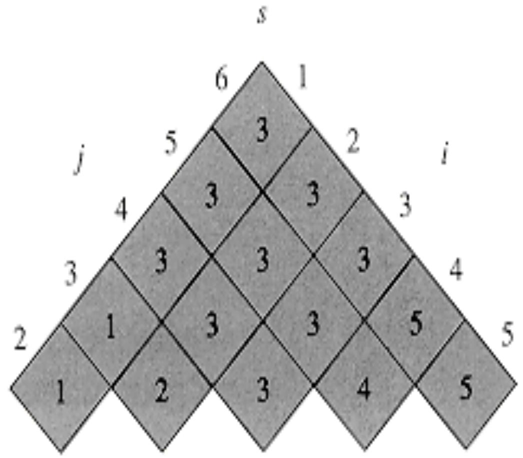


步骤3：计算最优值

```
MatrixChain(p)
n ← length[p] – 1
for i ← 1 to n
    do m[i, i] ← 0
for l←2 to n   //  l is the chain length.
    do for i ← 1 to n - l + 1
          do j ← i + l - 1
             m[i, j] ← ∞
             for k ← i to j - 1
                do q←m[i, k]+m[k+1, j]+p_{i-1}p_k p_j
                   if q < m[i, j]
                      then m[i, j] ← q
                           s[i, j] ← k
return m and s
```

A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25





步骤3：计算最优值

```
MatrixChain(p)
n ← length[p] – 1
for i ← 1 to n
    do m[i, i] ← 0
for l←2 to n      // l is the chain length.
    do for i ← 1 to n - l + 1
           do j ← i + l - 1
              m[i, j] ← ∞
              for k ← i to j - 1
                  do q←m[i, k]+m[k+1, j]+p_{i-1}p_k p_j
                     if q < m[i, j]
                        then m[i, j] ← q
                             s[i, j] ← k
return m and s
```

时间复杂度：$O(n^3)$

空间复杂度：$\Theta(n^2)$

步骤3：计算最优值

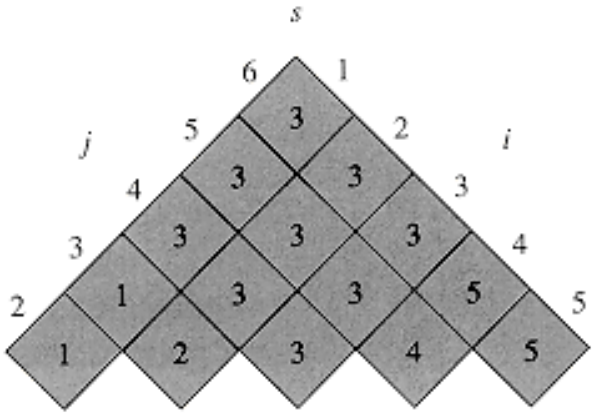
- 算法复杂度分析：
  - 算法的主要计算量取决于算法中对r，i和k的3重循环。循环体内的计算量为O(1)，而3重循环的总次数为O(n^3)。
  - 因此算法的计算时间上界为O(n^3)。算法所占用的空间显然为O(n^2)。

矩阵连乘问题

- 动态规划法基本步骤
  - 1 找出最优解的性质，并刻划其结构特征。
  - 2 递归地定义最优值。
  - 3 以自底向上的方式计算出最优值。
  - 4 根据计算最优值时得到的信息，构造最优**解**。

步骤4：求最优解

- MatrixChain 给出了如何求解最佳乘法次数 m[i,j]，但对于按什么顺序来相乘各矩阵，没有给出具体方法.
- 从 表 s[1.. n, 1.. n]中构建最优解
  - s[i, j] 记录值 k，表示在矩阵连乘 $A_iA_{i+1}\cdots A_j$的最优全括号中，分割点位于 A_k 和 A_{k+1} 之间
  - 因此，矩阵连乘 $A_{1..n}$ 的最优分割方式为
- $(A_1\ A_2\ \ldots\ A_{s[1,n]})(A_{s[1,n]+1}\cdots\ A_n)$.
- 矩阵乘法可以递归回溯得到
  - s[1, s[1, n]] → splits $A_{1..s[1,n]}$
  - s[s[1, n] + 1, n]→ splits $A_{s[1,n]+1..n}$
- PrintOptimalParents(s, i, j)



步骤4：求最优解

- 给定s表，PrintOptimalParents(i, j, s) 能够递归打印出 $<A_i, A_{i+1}, \ldots, A_j>$的最优完全加括号形式. 初始调用时 i=1, j=n

```
PrintOptimalParens(i, j,s) {
  IF j=i
     THEN
          Print "A"i;
     ELSE
          Print "(";
          PrintOptimalParens(i, s[i, j],s);
          PrintOptimalParens(s[i, j]+1, j,s);
          Print ")";
}
```

PrintOptinalParents(1,6, s)

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25

↓

((A1(A2A3))((A4A5)A6))

步骤4：求最优解

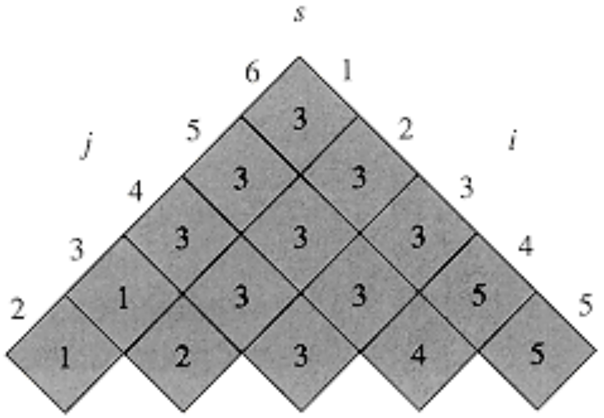
PrintOptinalParents(1,6, s)

```
PrintOptimalParents(i, j, s)
if i = j
    then print "A"_i
    else print "("
        PrintOptimalParents(i, s[i, j], s)
        PrintOptimalParents(s[i, j] + 1, j, s)
        print ")"
```

$A_1 A_2 A_3 A_4 A_5 A_6$： $((A_1\ A_2\ A_3)\ (A_4\ A_5)\ A_6)$?

$A_1\ (A_2\ A_3)$?　　$(A_1\ A_2)\ A_3$?

A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25



s(1,6)

(s(1,3) s(4,6))

(s(1,1) s(2,3))

(s(2,2) s(3,3))

$((A_1(A_2A_3))\ ((A_4A_5)A_6))$　?

矩阵连乘问题

- 动态规划法基本步骤
  - 1 找出最优解的性质，并刻划其结构特征。
  - 2 递归地定义最优值。
  - 3 以自底向上的方式计算出最优值。
  - 4 根据计算最优值时得到的信息，构造最优解。

矩阵连乘问题

- **动态规划法**基本步骤
  - 1 找出最优解的性质，并刻划其结构特征。
  - 2 递归地定义最优值。
  - 3 以自底向上的方式计算出最优值。
  - 4 根据计算最优值时得到的信息，构造最优解。

找子问题

写出最优值计算的递归表达式

填表

递归回溯

动态规划算法的变形

- 备忘录方法
  - 用一个表格来保存***已解决***的子问题的答案,在下次需要解此问题时,只要简单地查看该问题的解答,而不必重新计算.
  - 备忘录方法的控制结构与直接递归方法的控制结构相同
    - 递归方式: 自顶向下
  - 使直接递归算法的计算时间从Ω (2^n)降至O(n^3)
- 备忘录方法是动态规划算法的一个变形
  - 备忘录方法与直接递归方法的比较
    - 相同：控制结构相同
    - 区别：备忘录方法为每个*解过*的子问题建立了备忘录以备需要时查看，避免了相同子问题的重复求解。

动态规划算法的变形

- 备忘录方法与动态规划法比较
  - 都利用子问题重叠性质

	动态规划法	备忘录法
求解次数	所有子问题都至少要解一次	部分子问题可不必求解
结构形式	自底向上	自顶向下
查表方式	先填后查	先查后填

- 两种方法的适用条件
  - 当子问题都至少要解一次时,更适合用动态规划法；当子问题空间中的部分问题可不必求解时,用备忘录方法较有利

矩阵连乘问题 - 小结

- **动态规划法**基本步骤
  - 1 找出最优解的性质，并刻划其结构特征。
  - 2 递归地定义最优值。
  - 3 以自底向上的方式计算出最优值。
  - 4 根据计算最优值时得到的信息，构造最优解。

- 难点：
  - 改进策略：先求最优值，再求最优解
  - 动态规划算法两个基本要素
    - 最优子结构性质
    - 重叠子问题

动态规划算法的基本要素

How?

- 使用动态规划的条件
  - 最优子结构
    - 当一个问题的最优解包含了子问题的最优解时，说这个问题具有最优子结构

1)缩小子问题集合，降低复杂性
2)能自底而上地完成求解过程

  - 重叠子问题
    - 在问题的求解过程中，很多子问题的解将被多次使用

动态规划算法的基本要素

How?

- 最优子结构
  - 问题的最优解包含着其子问题的最优解。这种性质称为最优子结构性质。
- 在分析问题的最优子结构性质时，所用的方法具有普遍性
  - 首先假设由问题的最优解导出的子问题的解不是最优的，然后再设法说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。
- 最优子结构是问题能用动态规划算法求解的前提.利用问题的该性质, 以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。

同一个问题可以有多种方式刻划它的最优子结构，有些表示方法的求解速度更快（空间占用小，问题的维度低）

动态规划算法的基本要素

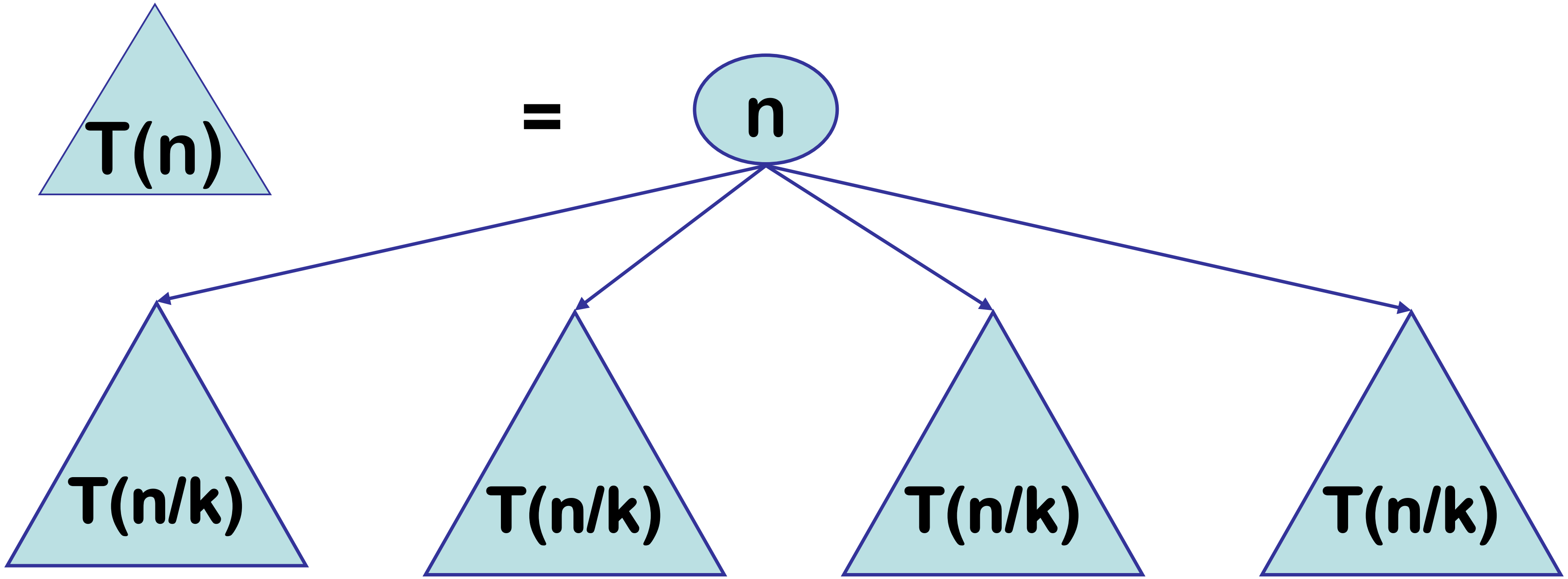
How?

- 重叠子问题
  - 递归算法求解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次。这种性质称为子问题的重叠性质。
  - 动态规划算法，对每一个子问题只解一次，而后将其解保存在一个表格中，当再次需要解此子问题时，只是简单地用常数时间查看一下结果。
  - 通常不同的子问题个数随问题的大小呈多项式增长。因此用动态规划算法只需要多项式时间，从而获得较高的解题效率。

动态规划算法的基本要素

How?

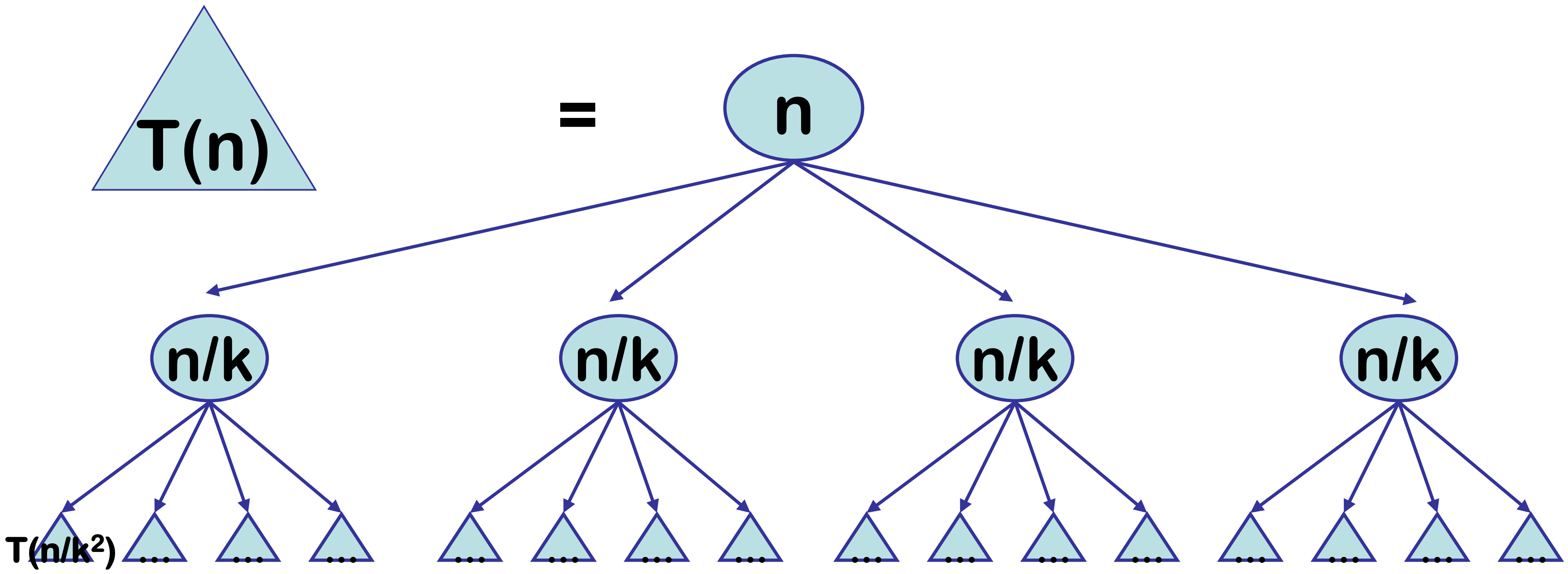
- 动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题



动态规划算法的基本要素

How?

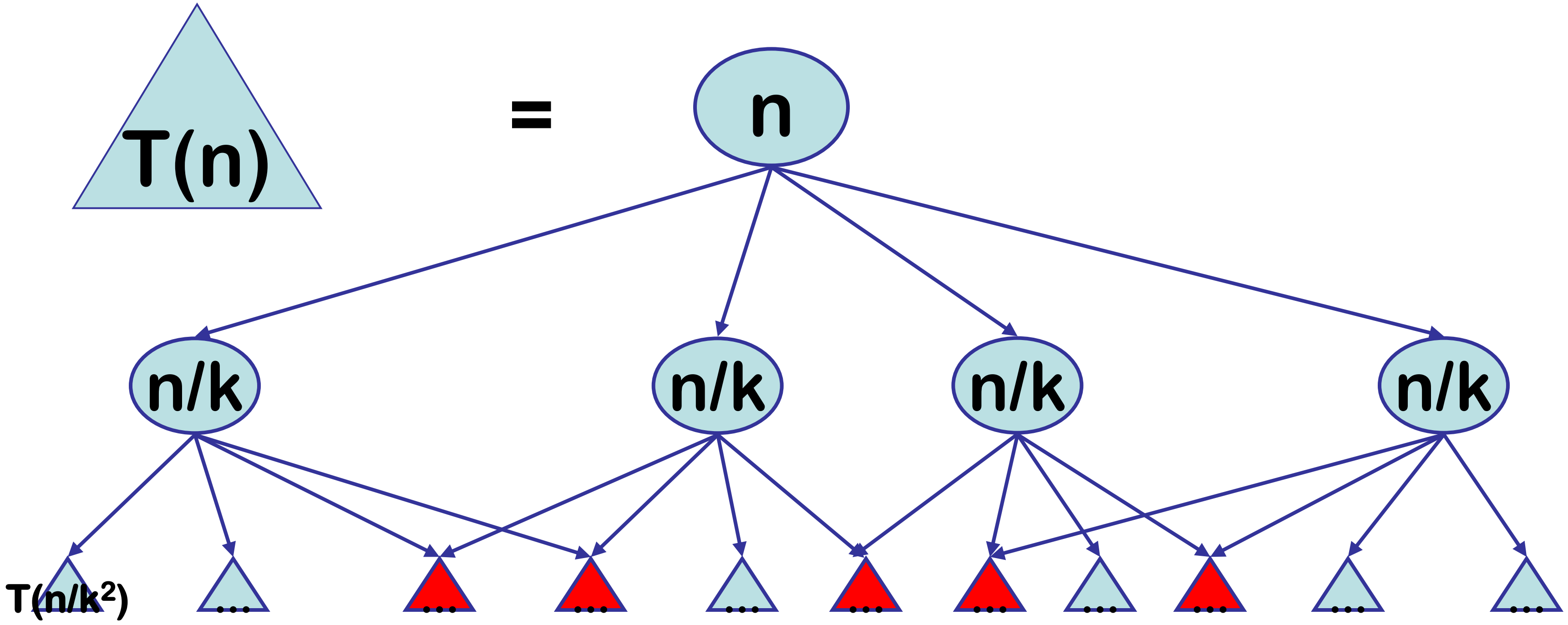
- 但是经分解得到的子问题往往不是互相独立的。不同子问题的数目常常只有**多项式**量级。在用分治法求解时，有些子问题被重复计算了许多次。



动态规划算法的基本要素

How?

- 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。



动态规划算法的基本要素

How?

- 动态规划算法的设计步骤(框架)
  - 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

动态规划的关键

- 动态规划的关键
  - 找出一个问题所包含的子问题及其表现形式

- 子问题的模式
  - 子问题是原问题的前缀
  - 子问题是原问题的中缀
  - 子问题是原问题的子树

动态规划的关键

- 子问题模式1
  - 原问题：$x_1,x_2,\ldots,x_n$
  - 子问题是： $x_1,x_2,\ldots,x_i$
  - 子问题个数：线性的
- 子问题模式2
  - 原问题：$x_1,x_2,\ldots,x_m$,和$y_1,y_2,\ldots,y_n$
  - 子问题是：$x_1,x_2,\ldots,x_i$和$x_1,x_2,\ldots,x_j$
  - 子问题个数：O(mn)

动态规划的关键

- 子问题模式3
  - 原问题：$x_1,x_2,\ldots,x_n$
  - 子问题是： $x_i,x_2,\ldots,x_j$
  - 子问题个数：$O(n^2)$
- 子问题模式4
  - 原问题：树
  - 子问题是：其子树
  - 若树有n个结点，它有多少个子问题呢？

小结

- 重点
  - 动态规划算法的两个要素：最优子结构和重复子问题
  - 动态规划算法的基本步骤
- 难点
  - 子问题的描述
  - 矩阵连乘问题转化为完全加括号的形式个数
  - 动态规划计算最优**值**的递归式的构造

实践能力训练：一维动态规划

- 扑克牌选取问题：假设有n张扑克牌排成一行，可表示为：c[0],c[1],…,c[n-1]。要求不能取相邻的两张牌，以获得累加面值最大的选取子序列。
  - 例如：有以下排列的牌：5，1，2，10，6，2
  - 可以取： 5+2+6=13， 1+10+2=13，最大为：5+10+2=17
- 分析
  - 枚举法：求出所有可行的选取子序列，并求出它们各自的累加和，其中累加和最大的序列就是结果。
  - 每个元素要么在要么不在选取序列中，共有2^n个可行序列
  - 时间复杂度T(n)=O(2^n)