Ⅳ 高级设计与分析技术

- 动态规划
- 贪心法
- 平摊分析

1

Chapter 15. 动态规划

Dynamic Programming

2

思想，就像幽灵一样……在它自己解释自己之前，必须先告诉它些什么。

——查尔斯.狄更斯

3

Dynamic Programming作为一种使多阶段决策过程最优的通用方法，在20世纪50年代由美国数学家理查德.贝尔曼提出

在应用数学中用来解决某类最优问题的重要工具，而且在计算机领域中被当做一种通用的算法设计技术来使用

4

Ch.15 动态规划

动态规划主要用于优化问题求解，即求出问题的最优(最大/小)解，当有多个最优解时一般是求一个即可

- 与分治法异同
  - ❖相同点：都是通过combining子问题的解来解决整个问题的解
  - ❖不同点 (Page 204 in our text book 第一段)

1)分治法是将问题划分为独立的子问题(互不相交)，递归地求解子问题，然后combine

5

Ch.15 动态规划

2)当分解问题非独立时，即他们共享子子问题时，可采用动态规划

因为此时分治法将重复地解这些共同的子子问题，形成重复计算，而动态规划对每一子子问题只做一次计算，然后将答案存储在一表中(这就是programming含义，像节目单一样)，故可避免重复计算 (critical issue: 一般方法是自底向上)

Note: Fibonacci数递归式求解

6

Ch.15 动态规划

Step1、2、3是基础

- 四个步骤 (课本P204)
  - Step1：描述最优解的结构特征
  - Step2：递归地定义一个最优解的值
  - Step3：自底向上计算一个最优解的值
  - Step4：从已计算的信息中构造一个最优解

若要构造最优解，则step3中应维护附加info，如求最短路径

钢条切割问题自学

7

15.1 矩阵链乘问题 (课本Page 210)

- 给定n个矩阵的序列$< A_1, A_2, ..., A_n >$，需要计算其积 $A_1 A_2 \cdots A_n$
- 计算多个矩阵连乘积可用括号来决定计算次序，每一个括号内的矩阵相乘调用标准的矩阵乘法
- Concept：矩阵积的完全括号化 递归定义
  - 它是单个矩阵
  - 或是两个完全括号化的矩阵积被包括在一个括号里
    - $(A_1(A_2A_3))$ 或 $((A_1A_2)(A_3A_4))$ 计算次序无二义性

8

15.1 矩阵链乘

- 矩阵乘法满足结合律，故所有不同的完全括号化方案产生同样积

例：A_1~A_4积有五种不同的完全括号化方式

$(A_1(A_2(A_3A_4)))$　$(A_1((A_2A_3)A_4))$

$((A_1A_2)(A_3A_4))$　$((A_1(A_2A_3))A_4)$

$((A_1A_2)A_3)A_4)$

Note：
观察分裂点位置

9

15.1 矩阵链乘

- 不同的括号化方式产生不同的计算成本

两矩阵相乘 $A_{pq} \cdot B_{qr}$ 的标量乘法次数为p*q*r

例：设 A_1, A_2, A_3 的维数分别为 $10\times100, 100\times5, 5\times50$

$((A_1A_2)A_3)$：
$A_1A_2 — 10\times100\times5 = 5000$
$(A_1A_2)A_3 — 10\times5\times50 = 2500$
Total:5000 + 2500 = 7500

$(A_1(A_2A_3))$：
$A_2A_3 — 100\times5\times50 = 25000$
$A_1(A_2A_3) — 10\times100\times50 = 50000$
Total:50000 + 25000 = 75000

10

15.1 矩阵链乘

- 矩阵链乘实质上是一个最优括号化问题
  - 给定$< A_1, A_2, ..., A_n >$，A_i 的维数 $p_{i-1}\times p_i (1 \le i \le n)$，在 $A_1 A_2 \cdots A_n$ 的积中插入括号使其完全括号化，且使得数量乘法次数最少
  - 计算括号数目

P(n)表示n个矩阵序列中可选括号化方案的数量，将该序列从k和k+1间划分为两子序列，然后独立地将其括号化，用穷举法产生的括号数：

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

11

15.1 矩阵链乘

其解是一个catalan数，是指数阶：$\Omega(4^n/n^{1.5})$ ，不如直接计算结果。

用动态规划可获得多项式解

- Step1：最优的括号化结构 (即描述最优解的结构特征)
  - 将$A_iA_{i+1}\cdots A_j$积简记为$A_{i..j}$，其中1≤i≤j≤n
  - 设$A_iA_{i+1}\cdots A_j$的最优括号化是在A_k和A_{k+1}之间进行分裂(i≤k≤j-1，要求i<j)
  - 对某个k，先计算$A_{i..k}$和$A_{k+1..j}$，然后将这两个积相乘产生积$A_{i..j}$

12

15.1 矩阵链乘

最优括号化的成本是：

计算$A_{i..k}$的成本+计算$A_{k+1..j}$的成本+两个积相乘成本

关键：

$A_iA_{i+1}\cdots A_j$的最优括号化亦要求前后缀子链$A_{i\cdots k}$和$A_{k+1\cdots j}$是最优括号化。可用反证法证明，若$A_{i\cdots k}$括号化不是最优，则可找到一个成本更小的方法将其括号化，代入到$A_{i\cdots j}$的最优括号化表示中，得到的计算成本比最优解小，矛盾！

13

15.1 矩阵链乘

■ **Step2：递归解 (递归地定义一个最优解的值)**

怎样用子问题的最优解递归地定义原问题的最优解(一般是最优解的值)？对矩阵链乘，子问题的最优解的值是：

确定$A_{i..j}$括号化的最小代价(即按最优括号化计算的成本)。

设m[i,j]是计算$A_{i..j}$所需乘法的最小次数(最优解的值)，则$A_{1..n}$的最小计算成本是m[1,n]。

1)若i=j，m[i,i]=0，$1\le i\le n$，链上只有一个A_i，无需乘法

14

15.1 矩阵链乘

2)若i<j，由Step1中的最优解结构可知：

假定最优括号化的分裂点为k($i\le k\le j-1$)，则：

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

其中$m[i,k]$:子积$A_{i..k}$的最小代价，

p_{i-1}:A_i的行，p_k:A_k的列，p_j:A_j的列

在不知道k的取值的情况下，可在j-i个值中选取最优者，所以$A_{i..j}$的最小计算成本为：

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

若要构造最优解，定义S[i,j]记录分裂点k

15

15.1 矩阵链乘

- $(A_1\ A_2\ A_3\ A_4\ A_5\ A_6)$

$m[1,3]$ $m[4,6]$

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

$p_0\times p_3$ $p_3\times p_6$

$m[1,3]+m[4,6]+p_0p_3p_6$

15.1 矩阵链乘

■ **Step3：计算最优解的值**

若简单地用递归算法计算积$A_{1..n}$的最小成本m[1,n]，则算法时间仍为指数阶。但是，满足 $1\le i\le j\le n$ 的i和j总共只有

$$\binom{n}{2}+n=\Theta(n^2)\ //\ i<j\text{有}\binom{n}{2}\text{个，}i=j\text{有n个}$$

即子问题个数并非指数。递归算法在其递归树的不同分支上要重复计算每个子问题，这是动态规划应用的另一特征(即重叠子问题)，故自底向上计算m的值。

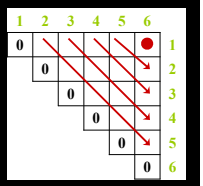
❖算法

17

15.1 矩阵链乘

Instead, use a dynamic program to fill in a table *m[i,j]*:

--Start by setting *m[i,i]*=0 for *i* = 1,…,n.
--Then compute *m*[1,2], *m*[2,3],…,*m*[*n-1*,*n*].
--Then *m*[1,3], *m*[2,4],…,*m*[*n-2*,*n*],…
--… so on till we can compute *m*[1,*n*].



The input a sequence $p=\langle p_0, p_1, \ldots, p_n\rangle$, we use an *auxiliary table* 辅助表 *s[1..n,1..n]* that records which index of *k* achieved the optimal cost in computing *m[i,j]*.

18

15.1 矩阵链乘

- 算法：按链长j-i+1递增序计算m[i,j]

输入：

$p=\langle p_0,p_1,\ldots,p_n\rangle$,其中 $p_{i-1}\times p_i$是A_i的维数

$m[1..n,1..n]$记录成本

$S[1..n,1..n]$记录相应分裂点k

19

```
MatrixChainOrder(p){
    n ← length[p]−1;   // p是维度向量，长度为n+1
    for i ←1  to  n  do{
        m[i,i] ← 0;
    }
    for l ←2  to  n  do{ // A_{i..j}链长 l = j−i+1
        // 第一次计算m[i,i+1],第二次计算 m[i,i+2]等
        for i ←1  to  n−l+1  do{
                // 1≤i≤n−l+1,   i+l−1≤j≤n
            j ← i+l−1; // A_{i..j}长度为 l, j−i+1=l
            m[i,j] ← ∞;
```

20

```
            for k ← i  to  j−1  do{ // 分裂点k
                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;
                } //endif
            } //endfor k
        } //endfor i
    } //endfor l
    return  m & S;
}
```

21

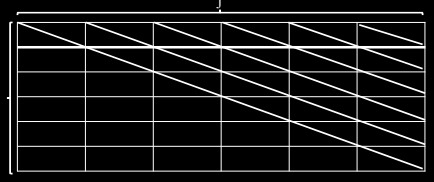
$l=2$：先计算$m[1,2],m[2,3],\ldots,m[n-1,n]$

$l=3$：先计算$m[1,3],m[2,4],\ldots,m[n-2,n]$

- 例：$A_{30\times35}\,A_{35\times15}\,A_{15\times5}\,A_{5\times10}\,A_{10\times20}\,A_{20\times25}$

$p=<30,35,15,5,10,20,25>$

$\because i\le j,\quad \therefore m[i,j]$是上三角阵



$T(n)=O(n^3)$，非指数。

$S(n)=\Theta(n^2)$

22

15.1 矩阵链乘

- Step4：构造一个最优解

∵S[i,j]=k记录了$A_{i..j}$最优括号化分裂点为k，

∴设k_1=S[1,n]，则$A_{1..n}$ 的计算次序是$A_{1..k_1}\cdot A_{k_1+1..n}$

而 $A_{1..k_1}$ 的计算次序应为：$A_{1..k_2}\cdot A_{k_2+1..k_1}, k_2=S[1,k_1]$

$A_{k_1+1..n}$的计算次序应为：

$$A_{k_1+1..k_3}\cdot A_{k_3+1..n}, k_3=S[k_1+1,n]$$

一般地， $A_{i..j}$ 的分裂点为S[i,j]=k

$$A_{i..j}\Rightarrow A_{i..k}\cdot A_{k+1,j}$$

23

```
MatrixChainMultiply(A,S,i,j){
    //求A_{1..n}需用参数1,n
    if j > i  then{
        X ← MatrixChainMultiply(A,S,i,S[i, j]);
        //求子积A_{i..k}
        Y ← MatrixChainMultiply(A,S,S[i,j]+1,j);
        //求子积A_{k+1..j}
        return MatrixMutiply(X,Y);
        //子积相乘，通常乘法
    }else  return A_i;   //i = j
}
```

24

Construct an Optimal Solution

PRINT-OPTIMAL-PARENS

```
PRINT-OPTIMAL-PARENS(s,i,j)
1  if i=j
2     then print "A"i
3     else print "("
4          PRINT-OPTIMAL-PARENS(s,i,s[i,j])
5          PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
6          print ")"
```

- 由于动态规划大多数应用都是求解最优化问题，因此我们需要指出这类应用的一个一般性法则，理查德贝尔曼称其为“**最优化法则**”(Principle of optimality)
  - 该法则认为最优化问题任一实例的最优解都是由其子实例最优解构成的。
  - 最优化法则在大多数情况下成立，少数情况例外！

26

15.2 动态规划要素

- 什么样的优化问题适合使用动态规划?
  - 最优子结构 *分治，DP，Greedy*
  - 重叠子问题 *DP is the best！*

  利用重叠子问题特性可导出动态规划的变种方法：memoization方法 (记忆，备忘录)
- **Wiki explanation:** In computing, memoization (a.k.a, memoisation) is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

27

15.2 动态规划要素

- Optimal Substructure
- 细节
- 重叠子问题
- 重构最优解
- Memoization

28

Optimal Substructure

- 若一个问题存在最优解，其内部包含的所有子问题解也必须最优，则该问题呈现了“最优子结构”，具有此结构特征的问题可能会使用动态规划。
  - 例如：矩阵$A_iA_{i+1}\cdots A_j$的最优括号化问题蕴含着两个子问题$A_i\cdots A_k$和$A_{k+1}\cdots A_j$的解也必须是最优的
  - 在动态规划中，可用子问题的最优解来构造原问题的最优解

29

Optimal Substructure

- 如何发现最优子结构呢(Principle)?
  - 说明问题的解必须进行某种选择，这种选择导致一个或多个待解的子问题
  - 对一给定问题，假定导致最优解的选择已给定，即无须关心如何做出选择，只须假定它已给出
  - 给定选择后，决定由此产生哪些子问题，如何最好地描述子问题结构(空间)
  - 证明用在问题最优解内的子问题的解也必须是最优的。方法是“*cut-and-paste*”技术和反证法。假定在最优解对应的子问题的解非最优，删去它换上最优解，得到原问题的解非最优，矛盾！

30

Optimal Substructure

- 如何描述子问题空间(子问题结构,不同的子问题个数)

  尽可能使其简单，然后再考虑有没有必要扩展. 例：

  $A_{1..n} \Rightarrow A_{1..k} \cdot A_{k+1..n} \Rightarrow (A_{1..k_1} \cdot A_{k_1+1..k})(A_{k+1..k_2} \cdot A_{k_2+1..n})$

  由此可见，最合适的子问题空间描述为：$A_i A_{i+1} \cdots A_j$

- 最优子结构有关的两方面问题
  - 用在最优解中有多少个子问题 (*i.e.*, 每次做出选择会产生多少个子问题)
  - 用在最优解中的子问题有多少种选择

    例如，矩阵链乘 $A_i A_{i+1} \cdots A_j$

    ——两个子问题，j-i种选择

31

Optimal Substructure

- 动态规划算法的运行时间
  - *子问题总数*
  - *对每个子问题涉及多少种选择*

  例如:

  矩阵链乘共要解$\Theta(n^2)$个子问题: 1≤i≤j≤n

  求解每个子问题至多有n-1种选择

  最终的运行时间为$\Theta(n^3)$
- 动态规划求解方式
  - 自底向上

32

细节

当心不要随便假定最优子结构的应用

例如有向无权图中，求最短/长路径的问题(指简单路径)

- 最短路径含有最优子结构

  设从u到v的最短路径是P，并设中间点为w，则

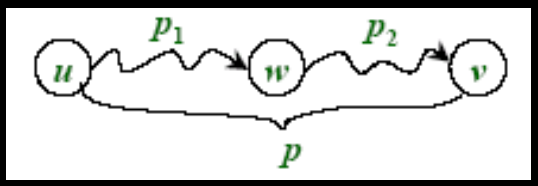
  $u \xrightarrow{P} v \quad \Rightarrow \quad u \xrightarrow{P_1} w \xrightarrow{P_2} v$

  显然P_1和P_2也必须是最优(短)的。

33

细节

- Shortest path has optimal substructure
  - Proof. If there's a shorter path from *u* to *w*, call it p'_1, then $p'_1 p_2$ contains a shorter path than $p_1 p_2$ from *u* to *v*, which contradicts the assumption.



34

细节

- 最长路径不具有最优子结构

  设P是从u到v的最长路径，w是中间某点，则

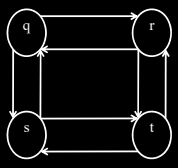
  $u \xrightarrow{P} v \Rightarrow u \xrightarrow{P_1} w \xrightarrow{P_2} v$

  但P_1不一定是从u到w的最长路径，P_2也不一定是从w到v的最长路径。 例：

  考虑一条最长路径 $q \to r \to t$

  但q到r的最长路径是：$q \to s \to t \to r$

  r到t的最长路径是：$r \to q \to s \to t$



35

细节

- 为什么两问题有差别？
  - 最长路径的子问题不是独立的。所谓独立指一个子问题的解不能影响另一个子问题的解。但第一个子问题中使用了s和t，第二个子问题又使用了，使得产生的路径不再是简单路径。

    从另一个角度看一个子问题求解时使用的资源(顶点)不能在另一个子问题中再使用。
  - 最短路径问题中，两子问题没有共享资源，可用反证法证明之。

    举例：如矩阵链乘 $A_{i..j} \Rightarrow A_{i..k} \cdot A_{k+1..j}$

    显然两子链不相交，没有资源共享，是相互独立的两子问题

36

重叠子问题

当用递归算法解某问题时，重复访问(计算)同一子问题

- 分治法与动态规划的比较
  - 当递归每一步产生一个新的子问题时，适合使用分治法
  - 当递归中较多出现重叠子问题时，适合使用动态规划，即对重叠子问题只求解一次，然后存储在表中，当需要使用时常数时间内查表。若子问题规模是多项式阶的，动态规划特别有效。

37

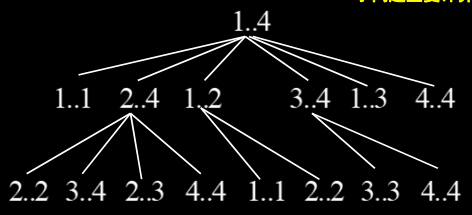
重叠子问题

- 例：用自然递归算法求解 (算法见P219 算法导论)

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

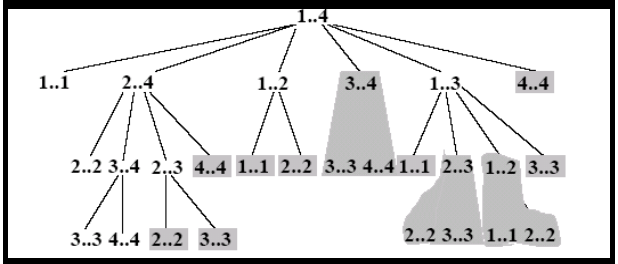
时 $A_{1\ldots4}$ 的递归树为：

子问题重复计算问题



38

Overlapping subproblems



重叠子问题

阴影部分是重叠子问题，递归算法须重复计算 (P219)。

递归式时间：

$$\begin{cases}T(1)\ge 1\\ T(n)\ge 1+\sum_{k=1}^{n-1}(T(k)+T(n-k)+1)\end{cases}\Rightarrow T(n)\ge 2\sum_{i=1}^{n-1}T(i)+n$$

用代入法可证为 $\Omega(2^n)$

请同学课下自行证明

- 结论：当自然递归算法是指数阶，但实际不同的子问题数目是多项式阶时，可用动态规划来获得高效算法

40

重构最优解

- 用附加表保存中间选择结果能节省重构最优解的时间

41

Memoization Version for DP

- 动态规划：分析是自顶向下，实现是自底向上
- 可采用备忘(记忆)型版本。它是动态规划的变种，效率和动态规划相似，但采用自顶向下实现，故是一个记忆型递归算法。
- 和动态规划类似，将子问题的解记录在一个表中，但填表的控制结构更像递归算法，其特点是：
  - 每个子问题的解对应一表项
  - 每个表目初值唯一，特殊值表示尚未填入
  - 在递归算法执行过程中第一次遇某子问题时，计算其解并填入表中，以后再遇此子问题时，将表中值简单地返回(不重复计算)，截断递归。
- 该方法的前提
  - 原有可能的子问题参数集合是已知的
  - 可在表位置和子问题间建立某种关系

42

```
MemoizedMatrixChain(p){
    n ← length[p]−1;
    for i = 1 to n do
        for j = i to n do
            m[i, j] ← ∞;
            //表目初值，上三角
    return LookupChain(p,1,n);
}
```



```
LookupChain(p,i,j){
    if m[i, j] < ∞ then //已计算过
        return m[i, j]; //截断递归
    //第一次遇到子问题 A_{i..j}，计算之
    if i = j then
        m[i, j] = 0;
    else
        for k ← i to j−1 then{
            q ← LookupChain(p,i,k) +
                LookupChain(p,k+1,j) + p_{i−1} p_k p_j;
            if q < m[i, j] then m[i, j] ← q;
        }
    return m[i, j]; //先计算，后返回
}
```


Memoized ver. of Recursive-Matrix-Chain

MEMOIZED MATRIX-CHAIN

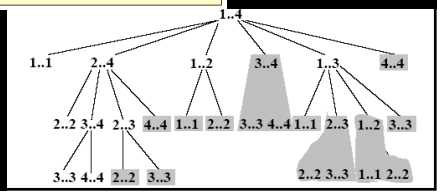
```
MEMOIZED-MATRIX-CHAIN(p)
1  n ← length[p]-1
2  for i ← 1 to n
3      do for j ← i to n
4           do m[i,j] ← ∞
5  return LOOKUP-CHAIN(p,1,n)
```

LOOKUP-CHAIN

```
LOOKUP-CHAIN(p,i,j)
1  if m[i,j] < ∞
2    then return m[i,j]
3  if i=j
4    then m[i,j] ← 0
5    else for k ← i to j-1
6           do q ← LOOKUP-CHAIN(p,i,k)
                  +LOOKUP-CHAIN(p,k+1,j)+ P_{i-1} P_k P_j
7              if q < m[i,j]
8                then m[i,j] ← q
9  return m[i,j]
```

阴影子树表示查表获得而未通过递归方式计算的子问题

树的遍历方式?



Top-Down Memoi. VS Bottom-Up DP

- Bottom-up dynamic programming
  - all subproblems must be solved
  - regular pattern of table access can be exploited to reduce time or space (利用一些特定的表访问模式可以进一步降低算法的时间以及空间开销)
- Top-down + memoization
  - solve only subproblems that are definitely required
  - recursion overhead
- Both methods solve the matrix-chain multiplication problem in $O(n^3)$ and take advantage of the overlapping-subproblems property.

初始化$\Theta(n^2)$,$\Theta(n^2)$个表目每个仅计算一次，但计算一个表目时需要$O(n)$时间，故总共$\Theta(n^3)$

小结

- 若所有子问题须至少解一次，自底向上的动态规划时间常数因子较优(不需要递归开销，维护表的开销较小)
- 若子问题空间有些不需要计算，则备忘型递归具有只需计算需要的子问题的优点。
- Ex 15.3-3

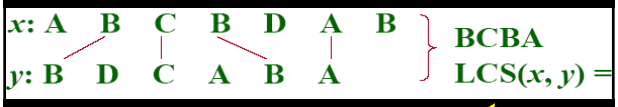
矩阵链乘问题属于突出阶段性的动态规划实例，每一个阶段都要在全面考虑各种情况下，做出必要的决策（e.g.，矩阵链乘在决定第一次分裂点时，我们其实是考虑了所有j-i种可能）

下面的例子是另一类动态规划问题，设计角度是从递推角度出发，设计过程不太强调阶段性，只需要找出大规模问题与小规模问题(子问题)之间的递推关系，当然每一个子问题是一个比原问题简单的优化问题。

Dynamic programming for LCS

Longest Common Subsequence (LCS)
-- Given two sequences x[1..m] and y[1..n], find a longest subsequence common to them both.

not "the"



Functional notation, but not a function

15.3 最长公共子序列(LCS)

- 子序列

  B是A的子列，若B是A中删去某些元素(亦可不删)所得的序列。即B不一定是由A的连续元素构成的子列。

  - 定义

    给定序列 $X=<x_1, x_2,\ldots, x_m>$，序列 $Z=<z_1, z_2,\ldots, z_k>$ 是X的一个子序列须满足：若X的索引中存在一个严格增的序列 $i_1 < i_2 <\ldots< i_k$ 使得对所有的 $j(1\leqslant j\leqslant k)$，均有 $x_{i_j} = z_j$ 成立。

50

15.3 最长公共子序列(LCS)

- 两个序列的公共子序列(CS)

  若Z是X的子列，又是Y的子列，则Z是序列X和Y的公共子序列

- 最长公共子序列(LCS)

  X和Y的公共子序列中长度最大者

- 如何求解两个给定序列的LCS
  - Step1：刻划LCS结构特征
  - Step2：子问题的递归解
  - Step3：计算最优解
  - Step4：构造一个LCS

51

如何求两给定序列的LCS

- Step1：刻划LCS结构特征
  - 穷举无效

    枚举X的所有子列，检查其是否亦为Y的子列，若|X|=m，则子序列数为2^m-1

  - LCS问题有一个最优子结构特性(Th15.1 P223)

    定义序列X的前 i 个元素构成的前缀：

    $X_i = <x_1, x_2,\ldots, x_i>$，$1\leqslant i\leqslant m$，$X_0 = \Phi$

52

如何求两给定序列的LCS

Th15.1(一个LCS的最优子结构)

设$X=<x_1, x_2,\ldots, x_m>$和$Y=<y_1, y_2,\ldots, y_n>$是序列，$Z=<z_1, z_2,\ldots, z_k>$是X和Y的任意一个LCS.

(1)若$x_m = y_n$，

则$z_k = x_m = y_n$且Z_{k-1}是X_{m-1}和Y_{n-1}的一个LCS **Case 1**

(2)若$x_m \neq y_n$，且$z_k \neq x_m$，

则Z是X_{m-1}和Y的一个LCS **Case 2**

(3)若$x_m \neq y_n$，且$z_k \neq y_n$，

则Z是X和Y_{n-1}的一个LCS

53

pf(反证法):

(1)若$z_k \neq x_m$(则$z_k \neq y_n$)，则将x_m加入Z尾，

于是获得X和Y的长度为$k+1$的CS,与Z是X和Y的LCS矛盾！ **对应则**

$\because z_k = x_m = y_n$

$\therefore$ 前缀Z_{k-1}是X_{m-1}和Y_{n-1}的公共子序列(长度为$k-1$)

若Z_{k-1}不是X_{m-1}和Y_{m-1}的LCS,

则存在一个X_{m-1}和Y_{n-1}的公共子序列W,其长度$>k-1$,

于是将z_k附加于W，则产生的公共子序列长度$>k$,

这与Z是X和Y的LCS(长度为k)矛盾！ **对应且**

54

(2)若$x_m \neq y_n$，且$z_k \neq x_m$，
则Z是X_{m-1}和Y的一个LCS
若$z_k \neq x_m$，z_k必等于x_m之前的某个元素，
则Z是X_{m-1}和Y的一个CS,现需证Z是一个LCS
若Z不是X_{m-1}和Y的LCS，则存在一长度$>k$的子序列W
它显然也是X和Y的CS，这与Z是X和Y的LCS矛盾！
(3)与(2)对称。

该定理表示：LCS问题有“最优子结构”性质：两个序列的一个LCS包含了两个序列的前缀子序列的一个LCS。

蕴含的选择：当$x_m \neq y_n$时，我们事先并不知道Z的长度，只能是在X_{m-1}和Y的LCS以及在X和Y_{n-1}的LCS中取最大者。

55

Dynamic-programming hallmark(特点)#1

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

z=LCS(x,y), then any prefix of z is an LCS of a prefix of x and a prefix of y.

56

如何求两给定序列的LCS

- Step2：子问题的递归解

定理15.1找X和Y的1个LCS，可分解为1个(或2个)子问题：

(1) If $x_m=y_n$ then 需解1个子问题：找X_{m-1} 和Y_{n-1} 的1个LCS

(2) If $x_m \neq y_n$ then 需解2个子问题：找X_{m-1}和Y的1个LCS，找X和Y_{n-1}的1个LCS，最长者为X和Y的LCS

❖LCS里重叠子问题特性

找X和Y的1个$LCS \Rightarrow$找X_{m-1} & Y以及X & Y_{n-1}的LCS，它们都包含子子问题：找X_{m-1}和Y_{n-1}的1个LCS，还有许多其它共享子子问题。

57

如何求两给定序列的LCS

❖用C记录最优解的值

$C[i,j]$定义为X_i和Y_j的一个LCS长度，$0 \le i \le m, 0 \le j \le n$

$$C[i,j]=\begin{cases} 0 & if\ i=0\ or\ j=0, X和Y有一个为空序列，则LCS长度为0 \\ C[i-1,j-1]+1 & if\ i,j>0, x_i=y_j \quad //case1 \\ \max\{C[i,j-1], C[i-1,j]\} & if\ i,j>0, x_i \neq y_j \quad //case2 \end{cases}$$

至此，可以给出直接递归版本的实现

为何引入第0行和第0列？

因为C[i,j]是当前行和前一行元素构成，或由当前列和前一列元素构成，故可起哨兵作用。

58

Recursive algorithm for LCS

RECURSIVE for LCS

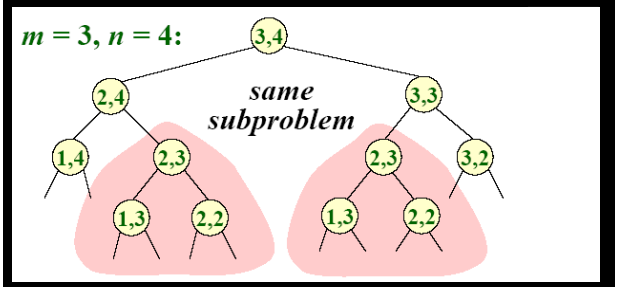
```
LCS(x,y,i,j)
1  if x[i] = y[j]
2     then c[i,j] ← LCS(x,y,i-1,j-1)+1
3     else c[i,j] ← max{LCS(x,y,i-1,j)
4                      LCS(x,y,i,j-1)}
```

Worst-case?

x[i] ≠y[j], in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

59

Recursive Tree 平凡递归算法对应的递归树



Height = m + n → work potentially exponential, since we're solving subproblems already solved

60

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 m and n is only mn.

61

如何求两给定序列的LCS

- Step3：计算最优解
  - 由递归关系式易写自然递归算法(指数阶)，但子问题空间规模为O(mn)。
  - 输入：$X=<x_1,x_2,...,x_m>$ $Y=<y_1,y_2,...,y_n>$
  - 输出：$C[0..m,0..n]$—计算次序行主序

    $b[1..m,1..n]$—解矩阵（空串无需表示出解）

$$b[i,j]=\begin{cases}\nwarrow & if\ C[i,j]由C[i-1,j-1]确定\\ \uparrow & if\ C[i,j]由C[i-1,j]确定\\ \leftarrow & if\ C[i,j]由C[i,j-1]确定\end{cases}$$

  - 故当构造解时，从b[m,n]出发，上溯至i=0或j=0为止，当b[i,j]包含“↖”时打印出x_i即可。

62

```
LCS_length(X,Y){
    m ← length[X];
    n ← length[Y];
    for i ← 1 to m do
        C[i,0] ← 0;//0th列
    for j ← 1 to n do
        C[0,j] ← 0;//0th行
    for i ← 1 to m do
    //依次考虑X1,X2,..,Xm的前缀子列
      for j ← 1 to n do
```

63

```
        if xi = yj then{
            C[i,j] ← C[i-1,j-1]+1;
            b[i,j] ← "↖";
        }else//xi ≠ yj
            if C[i-1,j] ≥ C[i,j-1] then{
                C[i,j] ← C[i-1,j];//前一行,Xi-1和Yj决定
                b[i,j] ← "↑";
            }else{
                C[i,j] ← C[i,j-1];//前一列,Xi和Yj-1决定
                b[i,j] ← "←";
            }
    return b&C;
}// 时间Θ(mn)，空间Θ(mn)
```

64

如何求两给定序列的LCS

- Step4：构造一个LCS

  从b[m,n]开始据箭头回溯至i=0或j=0即可。
  - 当b[i,j]= “↖”时，有$x_i=y_j$，打印之。
  - 用循环则打印元素为逆序，可用递归算法使之逆置

65

```
Print_LCS(b,X,i,j){
    if i=0 or j=0 then return;
    if b[i,j]="↖" then{//case1
        Print_LCS(b,X,i-1,j-1);
        Print xi;//相当于后序遍历
    }else//case2
        if b[i,j]="↑" then//前1行
            Print_LCS(b,X,i-1,j);
        else Print_LCS(b,X,i,j-1);
}
```

66

如何求两给定序列的LCS

初始调用：

$Print\text{-}LCS(b, X, length[X], length[Y])$

时间$O(m+n)$。因为i, j每递归一次至少有其一要减1，直至其中一个为0。

67

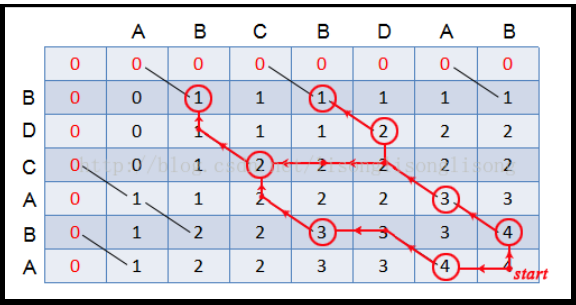
15.3 最长公共子序列(LCS)

- 改进代码

时空常数因子改变，但时间渐近性能不变

- 可省略b
  - ∵C[i,j]来自于C[i-1,j-1]， C[i-1,j]， C[i,j-1]
  - ∴可在O(1)判定是由谁计算产生的，故可在O(m+n)重构LCS。
  - T不变，S减少
- C只需要两行：当前行和前一行
  - 可是这样不能构造出解，但空间由O(mn)变为O(n)

68



$if\ C[i-1,j] \geq C[i,j-1]\quad then\{$

$C[i,j] \leftarrow C[i-1,j]$;//前一行，$X_{i-1}$和$Y_j$决定

$b[i,j] \leftarrow$"↑";

作业

- Ex 15.4-3，15.4-5

70

15.4 多边形的最佳三角剖分

- 三角剖分
- 最优的三角剖分
- 与矩阵链乘最优括号化问题一致
- 求解最优三角剖分问题

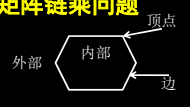
71

15.4 多边形的最佳三角剖分

1、三角剖分

凸多边形的最佳三角剖分类似于矩阵链乘问题

- 凸多边形



多边形：无交叉边

凸多边形定义：边界上或内部的任意两点连线上的点均在边界上或内部

- 表示

顶点逆时针列表，$P_n = < v_0, v_1, \ldots v_{n-1} >$

有n条边 $\overline{v_0v_1}, \overline{v_1v_2}, \ldots, \overline{v_{n-1}v_n}$，这里$v_0 = v_n$

顶点号是顶点数模除结果

72

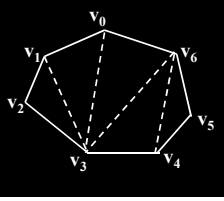
15.4 多边形的最佳三角剖分

- 弦(chord)：若v_i，v_j是不相邻点，则线段$\overline{v_iv_j}$是一弦，它将多边形划分为两个多边形：

$$< v_i, v_{i+1}, \ldots, v_j > 和 < v_j, v_{j+1}, \ldots, v_i >$$

- 一个多边形的三角剖分

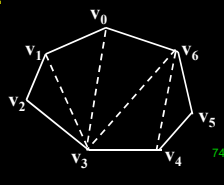
  它是多边形的一个弦集T，将多边形划分为若干个不相交的三角形



73

15.4 多边形的最佳三角剖分

- 一些结论
  - 在一个三角剖分中，没有弦相交(端点除外)且弦集合T最大：每条不在T中的弦和T中的某弦相交
  - 三角剖分产生的三角形的边只可能是剖分中的弦或多边形的边
  - n个顶点的凸多边形的每个三角剖分有n-3条弦，划分所得的三角形个数为n-2



74

15.4 多边形的最佳三角剖分

2、最优的三角剖分

Def：给定多边形及三角形的权值函数W，找到权值之和最小的三角剖分。

- 三角形权值：视具体问题而定，例如：

$$W(\Delta v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|$$

$|V_i V_j|$是V_i和V_j的欧氏距离

- 三角剖分的权：各三角形权值之和
- 最优三角剖分：权值之和最小的三角剖分

75

15.4 多边形的最佳三角剖分

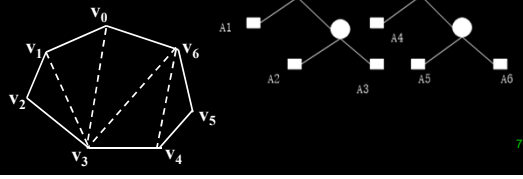
3、与括号化一致

通过树来解释三角剖分与表达式括号化的一致性

表达式的完全括号化 ⇔一棵丰满树(无度为的节点)

语法分析树

$((A_1(A_2A_3))(A_4(A_5A_6)))$ ⇔

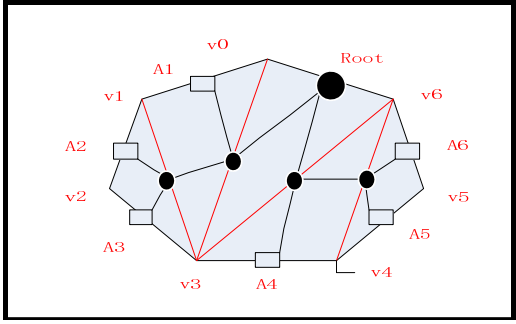


76

15.4 多边形的最佳三角剖分

凸多边形P_{n+1} ⇔语法分析树 (n片叶子)

$\overline{v_0v_n}$表示根，弦是内点，其余边是叶子，叶子 A_i表示边 $\overline{v_{i-1}v_i}$



77

15.4 多边形的最佳三角剖分

- n个矩阵的完全括号化⇔n个叶子的parse tree ⇔n+1个顶点的多边形的三角剖分
  - 矩阵$A_i(p_{i-1} \times p_i)$对应边 $\overline{v_{i-1}v_i}$
  - 矩阵积$A_{i+1..j}$对应弦 $\overline{v_iv_j}(i+1<j)$
- 最优矩阵链乘可看作是最优三角剖分的特例

矩阵链乘 ⇒一个最优三角剖分问题

链积$A_1A_2\ldots A_n$ ⇔$(n+1)$顶点多边形P_{n+1} $=< v_0, v_1, .., v_n >$

A_i维数$p_{i-1} \times p_i$ ⇔顶点$v_{i-1}v_i$

78

15.4 多边形的最佳三角剖分

❖最优矩阵链乘看作是最优三角剖分的特例(续)

定义三角剖分的权函数： $W(\Delta v_i v_j v_k) = p_i p_j p_k$

该权函数下的P_{n+1}的最优三角剖分就给出了 $A_1 A_2 .. A_n$ 的一个最优括号化。但反之不然，因为三角剖分问题的权函数更一般化，但可对链乘算法稍作修改即可求最优的三角剖分。

79

15.4 多边形的最佳三角剖分

4、求解最优三角剖分问题

- 用$v_0, v_1, \ldots, v_n$代替$p_0, p_1, \ldots, p_n$
- 修改MatrixChainOrder为

$$q \leftarrow m[i,k] + m[k+1, j] + W(\Delta v_{i-1} v_k v_j)$$

m[1,n]包含了P_{n+1}的一个最优三角剖分的权值，为什么？

80

15.4 多边形的最佳三角剖分

(1)一个最优三角剖分的子结构

设T是多边形 $P_{n+1} = < v_0, v_1, .., v_n >$ 的一个最佳三角剖分，它包括某个三角形 $\Delta v_0 v_k v_n (1 \le k \le n-1)$，即T的权

$$W(\Delta v_0 v_k v_n) + W(< v_0, v_1, \ldots, v_k >) + W(< v_k, v_{k+1}, \ldots, v_n >)$$

最优，要求子多边形

$< v_0, v_1, \ldots, v_k >$ 和 $< v_k, v_{k+1}, \ldots, v_n >$

的三角剖分也是最优的。因为，这两个子问题没有共享资源，相互独立，可用cut-and-paste证明。

81

15.4 多边形的最佳三角剖分

(2)一个递归解

- 设t[i,j](1≤i≤j≤n)是多边形<v_{i-1}，v_i，…，v_j>的一个最优解的值(即最优三角剖分的权)，则多边形P_{n+1}的最优三角剖分的权为t[1,n]。
- t[i,j]的递归定义如下：
  - 若i=j，多边形退化为<v_{i-1}，v_i> ，可定义权为0. 即t[i,i]=0, 1≤i≤n
  - 若i<j，则<v_{i-1}，v_i,…，v_j> 至少有3个顶点，设最佳剖分点为k(i≤k≤j-1)，则剖分结果：

$\Delta v_{i-1} v_k v_j$ $< v_{i-1}, v_i, \ldots, v_k >$ $< v_k, v_{k+1}, \ldots, v_j >$

82

15.4 多边形的最佳三角剖分

$$t[i,j] = \begin{cases} 0 & if\ i = j \\ \min_{i \le k \le j-1} \{t[i,k] + t[k+1, j] + W(\Delta v_{i-1} v_k v_j)\} & if\ i < j \end{cases}$$

比较此式与矩阵链乘完全一致，只是权函数不一样，所以只需修改MatrixChainOrder算法中一条语句即可

时间为 $\Theta(n^3)$，空间为 $\Theta(n^2)$

83

15.5 0-1背包问题

- 0-1背包问题
- 算法改进
- 一个例子
- 算法改进
- 一个例子
- 算法复杂度分析

84

15.5 0-1背包问题

1、0-1背包问题

- 给定n种物品和一背包。物品i的重量是w_i，其价值为v_i，背包的容量为C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大?
- 0-1背包问题是一个特殊的整数规划问题。

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

85

15.5 0-1背包问题

设所给0-1背包问题的子问题

$$\max \sum_{k=i} v_k x_k$$

$$\begin{cases} \sum_{k=i}^{n} w_k x_k \le j \\ x_k \in \{0,1\}, i \le k \le n \end{cases}$$

的最优值为m(i,j)，即m(i,j)是背包容量为j，可选择物品为i，i+1，…，n时0-1背包问题的最优值。由0-1背包问题的最优子结构性质，可以建立计算m(i,j)的递归式如下：

$$m(i,j) = \begin{cases} \max\{m(i+1,j), m(i+1,j-w_i)+v_i\} & j \ge w_i \\ m(i+1,j) & 0 \le j < w_i \end{cases}$$

86

15.5 0-1背包问题

$$m(n,j) = \begin{cases} v_n & j \ge w_n \\ 0 & 0 \le j < w_n \end{cases}$$

算法复杂度分析：

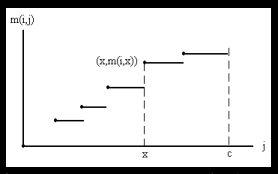
从m(i,j)的递归式容易看出，算法需要O(nc)计算时间。当背包容量c很大时，算法需要的计算时间较多。例如，当$c>2^n$时，算法需要$\Omega(n2^n)$计算时间。

87

15.5 0-1背包问题

2、算法改进

由m(i,j)的递归式容易证明，在一般情况下，对每一个确定的i(1≤i≤n)，**函数m(i,j)是关于变量j的阶梯状单调不减函数**。跳跃点是这一类函数的描述特征。在一般情况下，函数m(i,j)由其全部跳跃点唯一确定。如图所示。



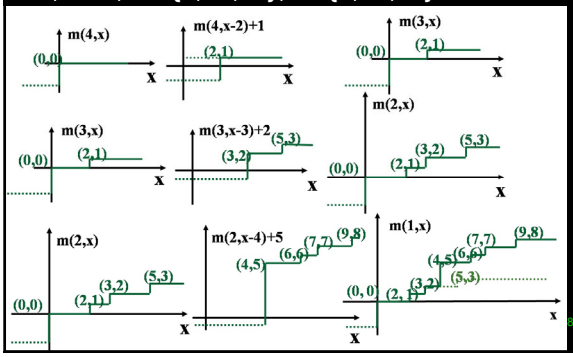
对每一个确定的i(1≤i≤n)，用一个表p[i]存储函数m(i,j)的全部跳跃点。表p[i]可以计算m(i,j)的递归式递归地由表p[i+1]计算，初始时p[n+1]={(0,0)}。

88

15.5 0-1背包问题

3、一个例子

n=3，c=6，w={4，3，2}，v={5，2，1}。



89

15.5 0-1背包问题

4、算法改进

- 函数m(i,j)是由函数m(i+1,j)与函数$m(i+1,j-w_i)+v_i$作max运算得到的。因此，函数m(i,j)的全部跳跃点包含于函数m(i+1，j)的跳跃点集p[i+1]与函数$m(i+1，j-w_i)+v_i$的跳跃点集q[i+1]的并集中。易知，(s,t) ∈ q[i+1]当且仅当$w_i \le s \le c$且$(s-w_i,t-v_i) \in p[i+1]$。因此，容易由p[i+1]确定跳跃点集q[i+1]如下

$q[i+1]=p[i+1] \oplus (w_i,v_i)=\{(j+w_i,m(i,j)+v_i) | (j,m(i,j)) \in p[i+1]\}$

- 另一方面，设(a，b)和(c，d)是p[i+1] ∪ q[i+1]中的2个跳跃点，则当c ≥ a且d<b时，(c，d)受控于(a，b)，从而(c，d)不是p[i]中的跳跃点。除受控跳跃点外，p[i+1] ∪ q[i+1]中的其它跳跃点均为p[i]中的跳跃点。
- 由此可见，在递归地由表p[i+1]计算表p[i]时，可先由p[i+1]计算出q[i+1]，然后合并表p[i+1]和表q[i+1]，并清除其中的受控跳跃点得到表p[i]。

90

15.5 0-1背包问题

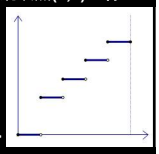
5、一个例子

n=5，c=10，w={2，2，6，5，4}，v={6，3，5，4，6}。

初始时p[6]={(0,0)}，(w5,v5)=(4,6)。因此，q[6]=p[6]⊕(w5,v5)={(4,6)}。

p[5]={(0,0),(4,6)}。

q[5]=p[5]⊕(w4,v4)={(5,4),(9,10)}。从跳跃点集p[5]与q[5]的并集p[5]∪q[5]={(0,0),(4,6),(5,4),(9,10)}中看到跳跃点(5,4)受控于跳跃点(4,6)。将受控跳跃点(5,4)清除后，得到p[4]={(0,0),(4,6),(9,10)}



q[4]=p[4]⊕(6，5)={(6，5)，(10，11)}

p[3]={(0，0)，(4，6)，(9，10)，(10，11)}

q[3]=p[3]⊕(2，3)={(2，3)，(6，9)}

p[2]={(0，0)，(2，3)，(4，6)，(6，9)，(9，10)，(10，11)}

q[2]=p[2]⊕(2，6)={(2，6)，(4，9)，(6，12)，(8，15)}

p[1]={(0，0)，(2，6)，(4，9)，(6，12)，(8，15)}

p[1]的最后的那个跳跃点(8,15)给出所求的最优值为m(1,c)=15。

91

15.5 0-1背包问题

6、算法复杂度分析

- 上述算法的主要计算量在于计算跳跃点集p[i](1≤i≤n)。由于q[i+1]=p[i+1]⊕(w_i，v_i)，故计算q[i+1]需要O(|p[i+1]|)计算时间。合并p[i+1]和q[i+1]并清除受控跳跃点也需要O(|p[i+1]|)计算时间。从跳跃点集p[i]的定义可以看出，p[i]中的跳跃点相应于$x_i,\ldots,x_n$的0/1赋值。因此，p[i]中跳跃点个数不超过2^{n-i+1}。由此可见，算法计算跳跃点集p[i]所花费的计算时间为

$$O\left(\sum_{i=2}^{n}|p[i+1]|\right)=O\left(\sum_{i=2}^{n}2^{n-i}\right)=O(2^n)$$

- 从而，改进后算法的计算时间复杂性为$O(2^n)$。当所给物品的重量$w_i(1\le i\le n)$是整数时，$|p[i]|\le c+1$，$(1\le i\le n)$。在这种情况下，改进后算法的计算时间复杂性为$O(\min\{nc,2^n\})$。

92